EXPLORATION OF A FRAMEWORK FOR BEHAVIOR-BASED MALWARE DETECTION AND CLASSIFICATION

TING MENG YEAN

NATIONAL UNIVERSITY OF SINGAPORE

2006

EXPLORATION OF A FRAMEWORK FOR BEHAVIOR-BASED MALWARE DETECTION AND CLASSIFICATION

TING MENG YEAN
B.CS (Hons.), Melbourne

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgements

I would like to thank A/P Chi Chi-Hung for his mentorship, the effort he put into our discussions and all his help in revising the thesis. I would also like to acknowledge Dr Ken Sung for all his support. Finally, I would like to thank my parents for supporting me and having faith in my work.

Contents

Summary

One of the greatest security threats that we face today is malwares like worms and viruses. But as current defenses against malwares are fast approaching their limits, we propose a new behavioral approach to combat this threat.

This thesis attempts to study the feasibility of detecting malwares based on behaviors and forms the basis of a new behavior-based detection system. While the final aim of our research is to study the behaviors of malware, the scope of this thesis is limit to malware detection. The reason for this approach is that we believe all malwares share some common behaviors, and malwares within the same families display more similar behaviors.

We will explore a framework that allows the modeling of high-level behaviors from Windows native API system calls. But rather than simply using sequences of API calls to build behavior signatures like many other researches, we built semantically rich behavioral signatures based on context provided the system call and reverse engineering based on descriptions provided by anti-virus companies.

In our analysis, we were successfully in identifying some behaviors common to all or most of our malware samples, but not to the set of normal applications used as baseline; thus showing the capability of our system to detect

for the presence of known malwares and newer malware variants. We were also able to observe some interesting features of the malwares by studying the behavioral information provided by the framework.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Computers today face an onslaught of security threats, from distributed denial-of-service attacks by botnets, to losing passwords and credit card information to keystroke loggers. While it seems that a myriad of security techniques are required to combat these threats, they do have a common cause: *malwares*.

Malwares are considered a high priority in the information security sector. We believe that any improvement in stopping malwares can be very helpful in slowing down the spread of malwares, thus significantly alleviating the security threats faced today.

As the current malware detection technology like the anti-virus systems are fast approaching their limits, we propose a new behavioral approach to combat this threat.

Rather than to attempt the herculean task of stopping malwares, we just seek to slow down the propagation. This can be accomplished just by being

able to detect some classes of novel malwares on certain operating systems.

We hope that by understanding malwares based on their behavior, we can provide another angle of looking at malware threats that can complement current detection technology.

1.2 Malware Introduction

Malware, or malicious software, is a broad category of software designed to cause computers to act in a way not authorized by their owners. Two common classes of malwares will be explored in this thesis based on what they do and how they spread: *viruses* and *worms*.

Viruses and worms have the ability to self-replicate: that is, they can spread copies of themselves within the infected host, or propagate themselves to other hosts. The main difference between viruses and worms is that worms have the ability to spread by themselves. Worms are usually self-contained and carry the propagation mechanism in addition to the exploits and payloads.

Viruses on the other hand, depend on the hosts to spread themselves. The most common propagation strategy is for the virus to embed itself in e-mail as attachment, depending on the recipient to open the viral attachment.

The rate of propagation for these mobile malwares is extremely fast. For example, the "Code-Red version 2" worms infected more than 359,000 hosts in less than 14 hours on July 19, 2001 [8]. It is not inconceivable for a hacker to be able to form a botnet of hundreds of thousands of infected hosts within a short period of time.

The greatest advantage of malwares is their automated, fire-and-forget vector of attack. That is, the hackers do not need to manually monitor the malwares they launched. Worms and viruses will spread by themselves; or be embedded into web pages or trojaned applications, just waiting for unsuspecting users to download and activate them. Malwares are widely believed to be the most pressing security concern for most of the Internet population.

To understand some of the problems caused by malwares, let us take the example of when a flash worm spreads: the process could take up a large amount of the network traffic. This could not only affect servers and hosts so much that legitimate users will experience some degree of denial-of-service, the wastage of the Internet or network bandwidth is also very expensive to Internet service providers.

1.3 Current Defense

Currently, the most common form of detection strategy against malwares is the *misuse-signature* based approach. This approach presumes any behavior in the knowledge base to be malicious, while any behavior not found in that knowledge base are presumed to be normal. We have countless anti-virus systems, spyware hunters, intrusion detection systems and intelligent firewalls utilizing this pattern-matching defense.

Misuse-signature based systems basically does pattern matching: anti-virus systems scans files and memory, and network-based intrusion detection systems scans network packets, for patterns matching known malicious binaries or protocol in its database.

While anti-virus systems have evolved to include heuristics to detect novel viruses, and sandboxing to extract the execution behavior of polymorphic malwares, their basic premise still depends upon a known database of exploit signatures.

Anomaly-statistical based approach, takes the opposite stance. It presumes any behavior in the knowledge base to be normal, but the knowledge base contains trend of past behaviors, as oppose to exact signatures. Any deviation from the behaviors in the knowledge base is classified based on heuristics or probability/statistics, to be abnormal, or possibly malicious.

The greatest strength of misuse-signature based approach is its high probability of correct threat identification. Compared to anomaly-based systems, it has a very low rate of false positives. For exact protocol or binary matches, the intrusion or malware detection is definite, rather than based on some confidence level.

While some might contend that searching through a large database of signature is not practical, hashing algorithms enables the matching of events or binaries to a large number of signatures to be done very efficiently.

The main disadvantage of the misuse-signature system is its inability to detect unknown threats. It is reactive as any new malwares or exploits must be captured before signatures can be created for them. The time lag between getting the malware sample and deployment of created signatures creates a time window for the new malware to spread. In addition, the process of signature creation is very labor and knowledge intensive.

1.4 Behavioral Approach

Our behavior-based approach utilizes high-level behaviors for malware detection. The basic assumptions that we made are that all malware have shared behaviors, and must perform some actions. We will show that it is possible to detect for the presence of malwares using known behaviors.

Another assumption that we made is that malwares within the same family share more similarity than with malwares in other family. If this is true, we will be able to generalize the detection behavior functions to detect novel variants of a malware family. Our framework will allow for the verification of this assumption in future work. This is important because if this assumption does not hold, we will have to explore another malware classification paradigm based on behavioral similarity to help our system detect newer malware variants.

While the final aim of our research is to study the behaviors of malware, the scope of this thesis is limit to malware detection.

1.5 Objectives and Contributions

The objective of this thesis is to show the feasibility of detecting malwares based on their high-level behaviors. We will explore a framework that can be used to help us study malware behaviors. In addition, we will show that the sample malwares shared a number of behaviors, thus showing the ability of this approach to detect unknown malwares based on behaviors collected from known malwares. The data collected is semantically rich enough to allow the identification of known malwares and classification of malwares based the similarity of their behaviors, as will as flexible enough to allow statistical analysis on the detected behaviors.

As this is a proof-of-concept work to explore the framework that can get quantitative proof, we would like to state the following limitations. We will explore the potential of this framework with a limited set of sample malwares and behaviors. The implementation of this work is not in real time, but via offline analysis.

We will show how we solved a series of problems for this research.

- What malware behaviors to use?

  We profiled the behaviors of the more prevalent of malware families from technical descriptions provided by anti-virus companies.
- What kind of sensor data to use?

  We explored various options to get behavioral information from the system, and finally settled on tracing native level system calls. We also explored various experimental issues to allow the malwares to exhibit as many behaviors as possible.
- How to get behaviors from system calls?

  We introduce a pattern matching approach to model behaviors from the system calls, based on the internal workings of Windows and information gained by studying the system call traces.
- Can behaviors be used to detect known malwares?

  We showed that malwares could be detected using certain behavioral functions. These behaviors appear in the majority of the malwares, but do not appear in any of the normal applications tested.
- Can behaviors be used to detect novel malwares?

  We showed that malware behaviors are composed of basic behavior blocks that are shared mainly between malware variants of the same family, and among a small number of malwares in other families. This

means that it is possible to detect a newer malware variant based on generalized behaviors.

For this research, we built a database of behavioral signatures collected from the sample malwares. While some malwares do share the same behavioral signatures, this database is growing as more malwares are added to the experiment. These behavioral signatures combine to form complex behaviors, or new behaviors not mentioned in the technical descriptions provided by anti-virus companies. We will introduce these descriptions in Chapter 4. We believe that a large collection of these behavioral signatures is vital to help us detect newer malwares.

1.6 Structure of Thesis

This thesis is structured into nine chapters, with the current chapter serving to introduce the current malware threat and some relevant background information.

Chapter 2 provides an overview of our behavioral approach, together with the justifications, advantages and disadvantages. The motivation for the approach is discussed, followed by the objectives and potential of this work.

Chapter 3 looks at some other research utilizing various kinds of behaviors for intrusion or malware detection.

In Chapter 4, we first look at the malware behaviors we extracted from technical descriptions provided by the anti-virus companies. We then perform some initial analysis on these behaviors to show that it is feasible to use behaviors to detect newer malwares.

Chapter 5 discusses all the experimental issues, from the choice of sensor to the network configuration.

Chapter 6 explores the methods we use to model high-level behaviors from system calls.

In Chapter 7, we analyze the behaviors captured from the malware samples. We showed that it is possible to detect the presence of malwares based on a small number of complex behaviors, and discuss more about the results.

Finally, Chapter 8 summarizes the whole thesis into a short conclusion and suggests areas in which future research may be performed to extend and improve the framework.

Chapter 2

Behavioral Approach Overview

2.1 Basic Concept

The term behavior has a number of different definitions in the area of intrusion detection research. For host-based signature-based research like anti-virus systems, behavior usually means patterns or sequences of instructions executed by a binary.

For anomaly-based research, behavior usually means the trend of the system's past profile. But as this area of research is very broad, profile could mean a different number of things. For example, the behavior of a network-based intrusion detection system could be the trend of frequency of certain types of network packets. The behavior of an anomaly-based host IDS could be the trend of the system's CPU and memory performance.

Behavior-based detection is significantly different from the general form of signature-based detection. Most signature-based approach looks for fixed patterns or regular expressions in payloads, but our behavioral approach attempts to detect patterns at a much higher level of abstraction.

A few examples of behaviors in the Windows environment will be given to illustrate our definition.

- Adding to registry key to start certain program at boot time;
- Copying files;
- Searching directories;
- Listening at certain network ports;
- Connecting to network shares;
- Initiating network connections to multiple hosts.

2.2 Risk Factor

In addition to the behaviors exhibited by malwares, we are also interested in the risk to normal operations posed by these behaviors. Every action taken contains an element of risk, as do the existence of any objects like files or registry keys. To better understand the behaviors of malwares, it is necessary to quantify the level of risk of each behavior.

Malwares have no risk until activation, thus file execution is riskier than file creation. Even the location of the file affects the risk factor, as it is more suspicious to access files in the Windows root directory than the Temporary directory. Then we have the file names: file names with double extensions like "`See_Britney_naked.jpg.scr`", or with white spaces between extensions like "Anna_Kournikova_nude.jpg␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣␣.exe" are commonly used by malwares to trick users into activating them.

We also have the risk of information leakage, where the malware contacts its author to reveal information found within the host. Thus outbound emails or network connections from new processes are risky; as is searching for or enumerating information from the local host.

As all these threats have different levels of risks, we would also need a management system to classify and respond to such threats.

2.3 Justification of Approach

If we look at malwares from a software engineering point of view, we can see that the malware execution process can be decomposed into subgroups of basic processes, each with simpler objections and behaviors. They can be viewed as functions to the main program.

Even though the computer is a deterministic machine and has a limited set of possible behaviors; interaction between programs, other hosts and users results in a very large set of behaviors. This makes quantifying the complete set of malware behavior or function very difficult.

While malwares may have large numbers of attack vectors and exploits, we believe that a lot of the resulting behaviors will be similar. That is, we believe that a lot of the malwares functions will overlap, even though current taxonomy places them into different family groups. Therefore, we believe that functional behaviors of malwares can be used to identify the presence of malwares in a system. If some of these behavioral functions are common to a lot group of malwares, they can even be generalized to detect malwares not seen before.

For example, if we find that most malwares share ten common functions that does not appear in normal applications, the probability of malware infection of any programs displaying these ten behavioral characteristics are very high. As we decrease the number of functions required to signal infection, the odds of catching a novel infection increases at the expense of

an increase in false positives.

Unlike anomaly-based systems, we do not claim to be able to detect all novel attacks.

2.4 Advantages of Approach

2.4.1 Value of Malwares

Hackers are motivated to write malwares for some kind of reward, either for fun or profit. Therefore, a malware without any purpose has no value. Malwares, like all other software programs, have very specific purposes.

Viruses and worms are meant to replicate and spread, so the originator can control more hosts. Hosts that are taken over can be used as launch pads to attack other machines; or to form part of a botnet, used to launch distributed denial-of-service attacks from.

Spywares are meant to collect user information, so that the malware author can profit from these information. This type of information leakage could contribute to credit card fraud or identity theft.

These general behaviors give us a starting point for our behavior-based approach to detect some specific types of malwares.

2.4.2 Limited Malware Actions

We believe that malwares are inherently simple programs, with a limited set of behaviors. If we look at malwares from a software designer point of view, we see that malwares can decomposed into the following packages that provide basic functions as shown below in Table 2.1.

Packages	**Function Examples**
Entry	Buffer Overflow, Weak passwords, Error in network service configuration,
Infection	Install rootkits, Replicate to local files, Enable malware during startup, Hide from system, Sabotage anti-virus defenses,
Propagation	Search hosts in local subnet, Send exploit to other external hosts, Search files, Email malware to addresses found, Copy malware to open network shares,
Payload	Install server allowing remote access, Keystroke Logging, Learn system information, Leak system information, Denial-of-service attacks,

Table 2.1: Malware Packages and Examples of Functions

The bulk of anti-virus research concentrates on preventing the malwares from entering the system; or if the malware succeeds in entering the system, prevents the executable from being executed or loaded. The problem with stopping attack vectors is that there are just too many different kinds. Even if we just look at buffer overflows, there are almost countless possibilities as any network-based applications or services; from the Internet Explorer to the LSASS (Local Security Authority Subsystem Service) could harbor potential vulnerabilities.

In addition, we notice from the initial study of prevalent viruses and worms in Chapter 4 that a large number of attack vectors depend on the carelessness of the user. A number of malwares depend on the users clicking on unknown attachments from emails, internet relay chats (IRC) or instant messengers. In fact, users are so careless that a number of newer malwares expects them to run unknown files from peer-2-peer or network file shares.

Weak password and executable rights on network shares is also another vector. These are all attack vectors that most research cannot guard against.

Our behavioral approach concentrates on dynamically looking for behaviors that indicate malwares had successfully entered our systems. That means we are effectively bypassing the detection of the entry mechanism, which have a large and constantly growing number of attack vectors and innovative exploits. We take advantage of the fact that while malwares can have many attack vectors, they have a limited number of actions that enables them to successfully replicate and perform their nefarious deeds.

2.4.3 Advantage against Obfuscated Threats

Recent malwares have attempted to use obfuscation techniques like polymorphism or metamorphism to hide from signature-based systems. For polymorphic malware, the exploit payload is either encrypted or encoded. For metamorphic malwares, parts of the instruction codes of the exploit are replaced with equivalent but different instruction codes. These obfuscated payloads will not match any previous pattern-based signatures because they will be different every time.

These threats cannot hide from our behavior-based system because exploits must be decrypted or decoded before activation. While binaries of metamorphic exploits can be changed to render previous signatures useless, the actions taken by the exploits are still the same. Unless the malware refrains from any known destructive or suspicious behaviors, we would still be able to detect them.

Thus, evading a behavioral signature requires a change in the fundamental behaviors, not just its binary code. Modifying malwares to escape behav-

ioral detection may be more difficult than just simple code transformation.

2.5 Limitations of Approach

2.5.1 Weakness of Dynamic System

Our behavioral approach, based on dynamic analysis of process behaviors within a system, aims to complement current signature-based techniques. It cannot replace static analysis because not all malware functions can be detected dynamically as certain conditions need to be met for some functions to occur.

For example, a number of malwares we studied attempts to terminate certain anti-virus systems or firewalls. If such software were not installed, we would not be able to study how the malwares kill these processes.

2.5.2 Truly Novel Behaviors

As our approach to detect newer malwares depends on the assumption that most malwares share some behavioral characteristics, it is unlikely that our behavior-based system will be able to detect malwares with truly novel behaviors.

If a new malware has behavioral characteristics so new or novel that no one has seen before, our system will not realize that it is under attack without any description of the new attack vector or characteristics.

It is also possible that some new malware could have functions that when seen individually are benign, but harmful when executed in some particular order. It is extremely difficult to detect this type of malware if we never encountered one before.

2.5.3 False Positive Rates

While the signature-based systems can detect malwares with very high level of confidence, our approach might generate a higher rate of false positives as our detection strategy depends on generalized behaviors that might be shared by normal applications.

Whether our approach can be refined to a satisfactory trade-off between false positive and detection rates is a question that we hope to answer in our future research.

2.6 Motivation

The study of malware behaviors has always been the domain of the anti-virus companies and a handful of malware researchers in various information security firms. Commercial tools like the Norman Sandbox [10] that can extract high-level behaviors from executable files arose from such researches. The problem is that these companies do not reveal any important details or quantitative data to the academic world. Even the information released cannot be readily verified because of the lack of implementation details or because propriety tools were used.

We want to study the behavioral approach to address the malware problems because it provides another angle of looking at these threats. We believe that understanding threats based on their behaviors provides a holistic view, and it is a promising model to start with. Furthermore, we believe that it can complement current technology.

We would like to provide a flexible framework that can be used to study malware behaviors. We hope to use this framework in future research to

provide quantitative data about the behaviors of malwares. This research raises a lot of questions and considerations that are very helpful to malware researchers because there are no current quantitative studies on malware behaviors. We also hope that further research will lead to a better malware classification scheme than the current ad hoc scheme that we will discuss in Section 4.10.1.

At this point, some of the interesting questions we would like to answer with our research are:

- Can behaviors by reliably extract from the operating system?
- Can behaviors be used to detect known malwares?
- Can behaviors be used to detect unknown malwares?
- Are malware behaviors similar to normal application behaviors?

In further research, we would also like to find out if malware behaviors are more similar among malwares within the same family, as opposed to across different families based on the current classification scheme.

2.7 Potential

While this research is only in the initial stage, we believe that further research can provide quantitative data that is useful to many information security researchers and practitioners. For example, the data can be used to help commercial behavior blockers to be more specific when guarding against malware actions. This research also has the potential to allow malware family classification using another paradigm. Finally, the information learned from future research in this area will help virus researchers and reverse engineers understand newer malwares better.

Chapter 3

Related Works

In a nutshell, my research aims to study the high level behaviors of malwares, for the purpose of detection and classification, using the Windows native API system calls. We will discuss the various degrees of overlaps between my work and other research works in this chapter.

3.1 Anomaly-based IDS using System Calls

There are a very large number of intrusion detection researches that looks at using system calls as a proxy for host's behavior, mostly in the Linux and UNIX environment. The number of such research working in the Windows environment is very small (see Section 5.1.3 for details). In many of these researches, the emphasis is on using techniques from various fields like data mining or text categorization to model normal or abnormal behavior based on sequences of system calls.

Using such techniques require a fixed format dataset of "transactions". The API system calls themselves do not have homogeneous format, with different number of parameters, parameters data types and return status codes. And since operating system behaviors like files, memory, network, etc all work differently, it is very hard to use all the system call information.

Many researches only use certain system call information, like the system call name alone, or with return value; but this means a lot of information is lost.

As our approach uses pattern matching to model behaviors, we have the option to use as many parameters as we need because we do not have the restriction of fixed data format.

The most common method to get the sequences of system call is by using sliding windows to extract a certain number of system call events from the entire system, or from just one process. Such solution is not very accurate because it loses context as a system call may rely on information provided by a previous system call event not within the current window. It also suffers from too much noise as system calls from unrelated behaviors like GUI or Windows synchronization will be mixed in.

This is not a big problem for anomaly-based systems as all the errors should be reduced with a large enough training data set, but it will be disastrous for our approach of detecting specific behaviors. We will introduce a new method to extract sequences of related system calls later.

The fixed or variable sliding windows of system call events are then assigned values representing normalcy or abnormality using various techniques. These values are then used to compute numerical results, whereby a value over a predefined threshold represents the probability of a normal behavior or an intrusion.

There are many such related IDS works that should be cited, but as we have limited space in our thesis, we will only cite some of the more relevant

works [14, 20, 35, 43, 44] for brevity.

3.2 Behavior Specific Research

In this section, we will introduce some research that concentrates on one or two behaviors.

3.2.1 Windows Registry Accesses

Stolfo, et al. [46, 1, 17] proposed to monitor Windows registry accesses. They used an anomaly-based approach: by considering the conditional probabilities between registry access datasets, they use this information to score registry records within processes to see if the process is anomalous. The dataset uses five features: name of process, type of query, actual key, return code and value of the key.

3.2.2 File System Accesses

Hershkop, et al. [19, 18] proposed to monitor file system accesses. They use seven features for each file access dataset: UID, user working directory, command line, parent directory of file, file name, PRE-FILE (concatenation of last 3 files) and frequency of file access (discretized: never, few, some, often). They use an anomaly-based detection algorithm similar to the previous work.

3.2.3 Code Injection Attacks

Chung and Mok [11] proposed to target code injection attacks as an improvement to system-call-based anomaly detection systems: trapping intrusion by catching code executing in data space. The claim is that it works like a specification-based intrusion detection system with only one

specified rule. It is also like a behavior-based system detecting only one behavior.

3.2.4 Code Replication

Summerville, et al. [47, 41, 40] proposed to detect the self-replication of codes, both local and network. Their implementation uses native Windows API, and the way they model behaviors from the system calls seems to be very similar to ours. But as the details of their implementation are vague, we cannot tell exactly how similar our implementations are.

3.2.5 Email Propagation Behaviors

Hu and Mok [22] proposed to monitor file searches and emails sent, to detect mass mailer viruses. This approach works because they use honeytoken files and email addresses, which are faked and not supposed to be accessed. Any access will be suspicious. Honeytokens are also used in our work. The behaviors are captured using API calls, and anomaly-based detection techniques are used to determine legal or illegal behaviors.

3.2.6 Network Traffic Monitoring

Williamson, et al. from Hewlett-Packard Labs proposed [57, 58, 50] a virus throttling strategy to slow down propagation of certain classes of worms and viruses based on normal network behavior. It is observed that a computer normally make fairly little attempts to connect to new machines, which is the opposite behavior of a rapidly spreading worm.

If a computer starts to make many connections to new machines, the suspicious traffic will be rate-limited, and can be stopped. They only look out for one behavior: the outgoing traffic rate. This system can be classified as network-anomaly based.

3.3 Behavior-based Research

In this section, other behavior-based research will be explored.

3.3.1 Deductive Reasoning

Hollebeek and Waltzman [21] from Teknowledge Corp proposed using computer forensics techniques to manually create general rules describing suspicious events, and using directed acyclic graph for deductive reasoning of intrusion. The sensor used is the SafeFamily wrapper [3, 2], which intercepts shared library calls.

The basic idea behind both our approaches is very similar, but we create behavioral signatures from previously seen malware behaviors instead.

3.3.2 Static Analysis for Vicious Executable

Xu, et al. from New Mexico Tech [59] proposed an anti-virus system SAVE (Static Analyzer for Vicious Executable) that analyzes the API calling sequence of the binary, instead of the binary code itself. The signatures used are API calling sequence of known malware. Detection is based on the similarity between their database of signatures and the target's calling sequence.

3.3.3 Malware Behavior Detection Systems

Norman Anti-virus has a product Norman SandBox [10] that can study the actions taken by an executable file. The Sandbox captures behaviors like file, registry, memory and network accesses. Because it is a commercial product, we have no knowledge of its implementation.

Willems attempts to replicate and improve upon the Norman SandBox,

and implemented the CWSandbox [54, 55, 56]. But rather than to monitor the operating system, CWSandbox works by injecting API hooking code into the malware application. Thus any API call by the malware is directed to CWSandbox, instead of to Windows. The behaviors provided by CWSandbox are only as descriptive as the system call allows.

Bayer's TTAnalyze [6] is another such system. The implementation is by means of emulating the Windows environment. Like CWSandbox, system calls can only provide low-level behavioral information.

3.3.4 Gatekeeper

Wagner's [52] work uses Florida Institute of Technology's Gatekeeper system to identify malwares.

The initial portion of both our research have very much in common, both our research surveys malware descriptions from anti-virus companies to find out what kind of behaviors to look for. Gatekeeper monitors the Win32 API system call, which is at a higher level than our native level API. While Win32 API system calls are more descriptive than the native level, malwares may utilize other high level APIs thus bypassing Gatekeeper.

As the aim of Gatekeeper is to detect malwares to undo their damages, whereas our aim is to detect and classify, the focus of our analysis are very different.

3.3.5 Behavioral Classification

Lee and Mody's work [26] attempt to classify malwares based on the behaviors. Like our work, they use sequences of native API system calls. But from the examples given, it appears that they capture native APIs

system calls at the kernel mode. This is significant because our work, like many other security products, can only captures the system call at the user mode. Our hypothesis is that because the authors belong to Microsoft's anti-malware team, they have special access to the Windows kernel.

They extract sequences of system calls to form Event Objects. As the article is vague on details, we do not know the algorithm for this extraction. Similarities between objects are then calculated based on string edit distance. The results are then clustered using what the authors call a k-medoid partitioning algorithm, which is a modified K-means algorithm using medoids rather than centroids. Classification of malwares is based on their edit distance from the nearest medoid.

Chapter 4

Malware Behaviors

In this chapter, we will make use of publicly available information from the anti-virus companies. We will first identify some of the malware behaviors worth looking into, and do a preliminary study on the level of shared behaviors within the same family and across different families.

4.1 Malware Propagation Share and Trends

In any behavioral studies, it is important to have a large sample population. But as the number of available malwares is too large for this study, we decided to limit the actual test samples based on their prevalence and importance.

Proof-of-concept malwares are written specifically to test some new vulnerabilities or attack vectors, and do not cause much harm. While this class of malwares is interesting, they do not provide much behavioral information. Therefore we do not bother about this class of malwares.

On the other hand, *in-the-wild* malwares are actually spreading throughout the Internet. A number of anti-virus companies provide lists of the top most prevalent malwares captured, and Kaspersky Lab has a comprehen-

sive archive of their past "Top Twenty viruses" of the month. Kaspersky's Top Twenty [24] virus list begins from 2001, and we compiled 48 months worth of viruses that appeared on the lists, from November 2001 to January 2006. (Except November 2002, December 2002 and July 2003)

Malware	**Share (%)**
Email-Worm.Win32.Klez.a	16.3452
Email-Worm.Win32.NetSky.b	5.6447
Email-Worm.Win32.NetSky.q	5.2725
Email-Worm.Win32.BadtransII	4.8027
Email-Worm.Win32.Zafi.b	4.7541
Net-Worm.Win32.Mytob.c	4.1483
Email-Worm.Win32.Lentin.a	3.7791
Email-Worm.Win32.Zafi.d	3.6954
Email-Worm.Win32.Swen	3.5662
Email-Worm.Win32.NetSky.aa	3.5481
Email-Worm.Win32.Sobig.a	3.4893
Email-Worm.Win32.Mydoom.a	3.4562
Email-Worm.Win32.LovGate.w	1.7710
Email-Worm.Win32.Tanatos.a	1.5202
Email-Worm.Win32.Mimail.c	1.3779
Email-Worm.Win32.NetSky.d	0.9293
Email-Worm.Win32.NetSky.t	0.8166
Email-Worm.Win32.Bagle.z	0.7585
Email-Worm.Win32.Mydoom.m	0.7143
Email-Worm.Win32.Bagle.at	0.6743
	71.0639

Table 4.1: Captured Traffic Share of Top 20 Malwares

Family	**Share (%)**
NetSky	17.4816
Klez	16.5031
Zafi	8.4495
Mytob	8.1370
Mydoom	5.4995
BadtransII	4.8027
Lentin	3.9622
Sobig	3.5816
Swen	3.5662
Bagle	2.4568
Mimail	2.4558
LovGate	1.9050
Tanatos	1.6125
	80.4135

Table 4.2: Captured Traffic Share of Top 13 Malware Families

A total of 274 unique malwares from 168 families were identified. We can see from Table 4.1 that the top twenty malwares represents 71.0639% of the total captured malware traffic population. The 20 most prevalent malwares belong in 13 families and the top 13 families represents 80.4135% of the total population as seen from Table 4.2. Details about the variants within the malware families can be seem from Appendix A.

The malware share information from both Table 4.1 and 4.2 was simply computed from the percentage of the malware traffic shares over 48 months. We know that older results should be less important, and a factor should be included to give shares from recent months more importance. But we believe that this simple result is sufficient for the initial study, and a reward factor biased towards more recent malwares will be included in our future work.

This information provides confidence that a small set of prevalent malwares is a good enough starting point for our research.

4.2 Malware Sample Choices

Anti-virus companies spend enormous effort to study malwares using static and dynamic analysis. As a service to their customers and for public relations purposes, technical characteristics of malwares detected by their products are openly available, albeit lacking in details.

As a starting point in our research and to boost confidence that malwares do exhibit similar behaviors, we decided to first study the descriptions of a small sample of malwares.

From the initial study of the malware descriptions from anti-virus companies, one observation made was that a significant number of malwares from the same family have almost identical technical descriptions, differing only in the keywords or file names used. Even if the newer malware have more complicated actions than its predecessors, the basic infection functions are the same.

Email-Worm.Win32.Bagle.a
Email-Worm.Win32.Ganda
Email-Worm.Win32.Gibe.a
Email-Worm.Win32.Klez.a
Email-Worm.Win32.Lentin.a
Email-Worm.Win32.LovGate.a
Email-Worm.Win32.Lovelorn.a
Worm.Win32.Lovesan.a
Email-Worm.Win32.Mimail.a
Email-Worm.Win32.Mydoom.a
Email-Worm.Win32.Sober.a
Email-Worm.Win32.Sobig.a
P2P-Worm.Win32.SpyBot.a
Net-Worm.Win32.Welchia.a
Email-Worm.Win32.Zafi.a

Table 4.3: First Malware From Each Sample Family

Email-Worm.Win32.Bagle.z
Email-Worm.Win32.Bagle.ai
Email-Worm.Win32.Bagle.at
Email-Worm.Win32.LovGate.b
Email-Worm.Win32.LovGate.ad
Email-Worm.Win32.Klez.e
Email-Worm.Win32.Klez.h
Email-Worm.Win32.Sober.f
Email-Worm.Win32.Sober.g

Table 4.4: Newer Malware Variants From Some Sample Families

We decided to concentrate on the earliest discovered virus of each family. The bulk of the initial behavioral study from the descriptions was from the first malware from each of the more prevalent families, as shown in Table 4.3. We included several newer malware variants from some of the sample families in the study, shown in Table 4.4, to compare the difference between ancestor and descendant behavioral functions.

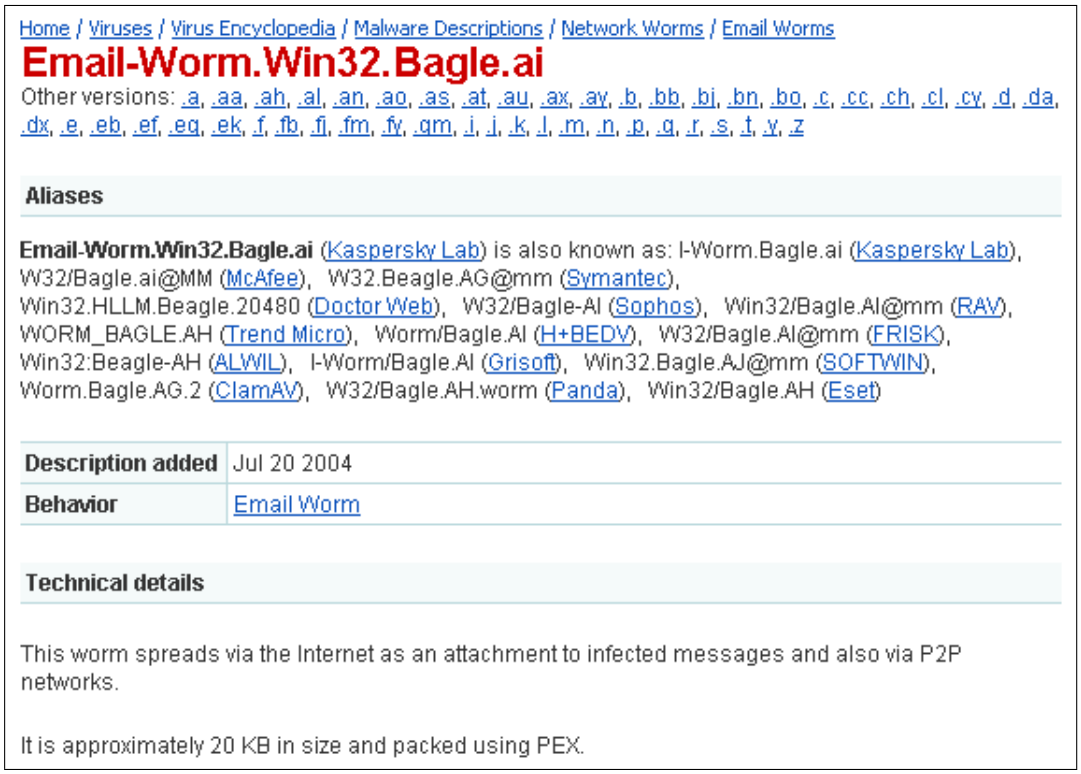
The reader might notice that the sample malwares were not all drawn from the most prevalent families shown in Table 4.2. The reason is because as this research is based on the dynamic behavior of the malware, we are constraint to include only malwares that we can find the executables for. Therefore, the total number of malwares we will be using in this initial study is **twenty-four**.

4.3 Malware Behavior Survey

4.3.1 Choice of Information Source

The samples of malwares that we chose are identified by Kaspersky Lab's naming conventions because we used Kaspersky Lab's Top Twenty Virus ranking information. The problem of using just Kaspersky Lab's malware descriptions is that it does not provide very detailed technical descriptions for all the malwares, and there are some ambiguities because English is a not a precise language. After exploring the databases of different anti-virus companies, I've decided to add information from Computer Associates and Trend Micro because the union of the information from these different anti-virus companies' description database provides a good level of accuracy.

An extract of the technical description of Email-Worm.Win32.Bagle.ai from Kaspersky Lab is as follows. The full technical description is available in Appendix E.



Home / Viruses / Virus Encyclopedia / Malware Descriptions / Network Worms / Email Worms

Email-Worm.Win32.Bagle.ai

Other versions: .a, .aa, .ah, .al, .an, .ao, .as, .at, .au, .ax, .ay, .b, .bb, .bj, .bn, .bo, .c, .cc, .ch, .cl, .cy, .d, .da, .dx, .e, .eb, .ef, .eq, .ek, .f, .fb, .fj, .fm, .fy, .gm, .i, .j, .k, .l, .m, .n, .p, .q, .r, .s, .t, .v, .z

Aliases

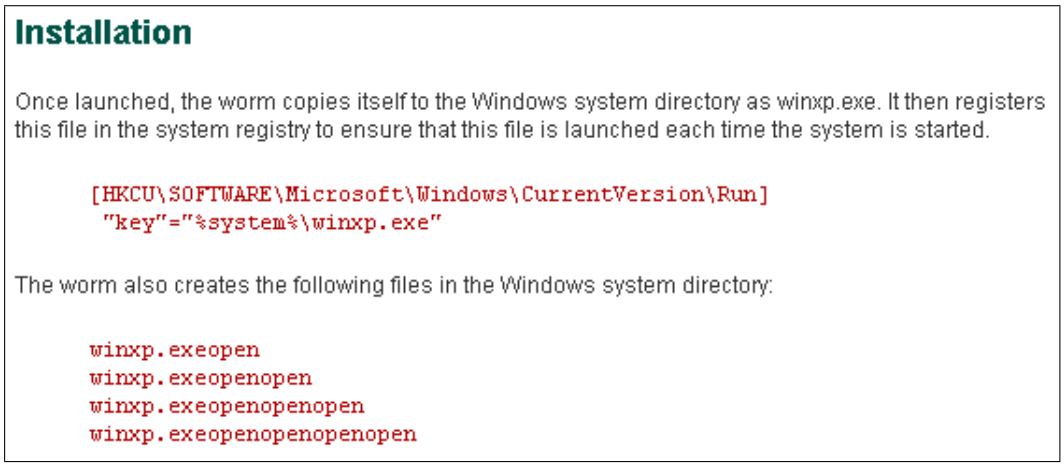
Email-Worm.Win32.Bagle.ai (Kaspersky Lab) is also known as: I-Worm.Bagle.ai (Kaspersky Lab), W32/Bagle.ai@MM (McAfee), W32.Beagle.AG@mm (Symantec), Win32.HLLM.Beagle.20480 (Doctor Web), W32/Bagle-AI (Sophos), Win32/Bagle.AI@mm (RAV), WORM_BAGLE.AH (Trend Micro), Worm/Bagle.AI (H+BEDV), W32/Bagle.AI@mm (FRISK), Win32:Beagle-AH (ALWIL), I-Worm/Bagle.AI (Grisoft), Win32.Bagle.AJ@mm (SOFTWIN), Worm.Bagle.AG.2 (ClamAV), W32/Bagle.AH.worm (Panda), Win32/Bagle.AH (Eset)

Description added	Jul 20 2004
Behavior	Email Worm

Technical details

This worm spreads via the Internet as an attachment to infected messages and also via P2P networks.

It is approximately 20 KB in size and packed using PEX.



Installation

Once launched, the worm copies itself to the Windows system directory as winxp.exe. It then registers this file in the system registry to ensure that this file is launched each time the system is started.

```
[HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
 "key"="%system%\winxp.exe"
```

The worm also creates the following files in the Windows system directory:

```
winxp.exeopen
winxp.exeopenopen
winxp.exeopenopenopen
winxp.exeopenopenopenopen
```

Figure 4.1: Extract of Kaspersky Lab Email-Worm.Win32.Bagle.at Description

4.3.2 Text Description Conversion to Behavioral Functions

To improve the confidence of our assumption that malwares share similar behaviors, we have to quantify the similarities of malware behaviors. The problem is that descriptions written in normal English cannot be used to generate quantifying statistics. We would need to create a grammar to describe behaviors based on logic.

In an ideal situation, we should decide on what behaviors to detect, and then decide on the sensors needed to collect the necessary information. But in reality, we are doing both at same time to find out our limitations and constraints. As we chose to monitor the native API system calls, we understand that there are certain behaviors that cannot be detected: for example, program logic like “if else” decisions; or manipulation of data based on regular expressions (used for ignoring certain type of email addresses).

As there are many unknown variables and the descriptions are highly complex, the conversion matrix is incomplete at this time and are constantly redesigned to fit new scenarios. The basic criteria we imposed on the conversion process are that the descriptive functions must be:

- Expressive enough to replace the language descriptions.
- Simple enough to be parsed by scripting languages using regular expressions.
- Possible to be detected using native API system calls.

We decided to represent the behavior functions using a pseudo language based on the Perl language and UNIX shell commands. Two converted examples are shown in Appendix F.

4.4 Behavior Functions

We will introduce the functions seen in the malware descriptions and their parameters in this section. As we are introducing sixty-nine behaviors, we will only demonstrate a couple of examples of the conversion process.

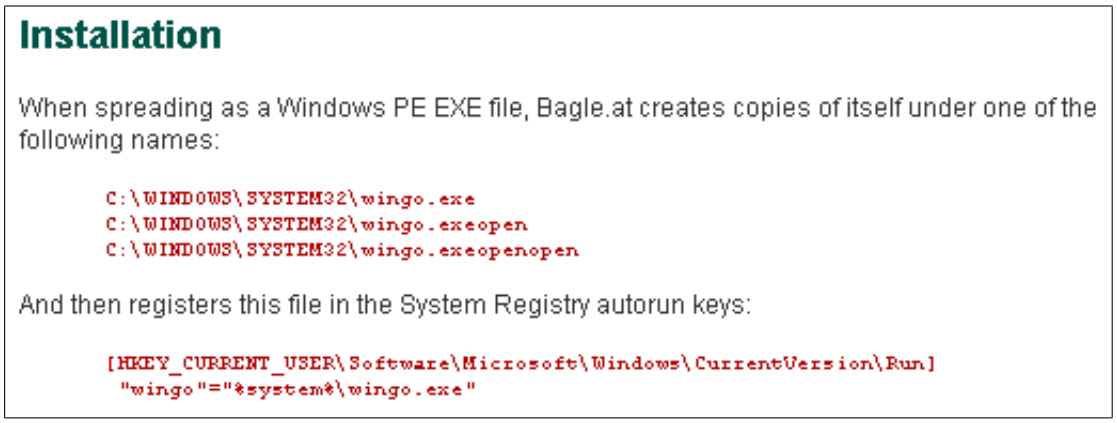


Figure 4.2: Description of Email-Worm.Win32.Bagle.at File Copy and Registry Creation Behaviors

From Figure 4.2, we know that the malware copy itself to three files:

```
file_copy $SELF C:\Windows\System32\wingo.exe ;
file_copy $SELF C:\Windows\System32\wingo.exeopen ;
file_copy $SELF C:\Windows\System32\wingo.exeopenopen ;
```

where `$SELF` is the original malware binary file that was started.

Then we have an addition to the registry:

```
registry_add
   "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
   "wingo = %System%\wingo.exe" ;
```

where %System% is the variable name for `C:\Windows\System32` under certain versions of Windows. We will elaborate on this later. Thus, we captured two behavioral functions.

4.4.1 File and Directory

In most of the write-based file functions below, the parameter that is most important for providing information to differentiate malwares is the path. The paths that we are most concern about are the Windows directory (%Windows%) and the Windows System directory (%System%). The %System% folder is usually `C:\Windows\System` on Windows 95, 98 and ME, `C:\WINNT\System32` on Windows NT and 2000, and `C:\Windows\System32` on Windows XP. The %Windows% folder is usually `C:\Windows` or `C:\WINNT`. These paths are noteworthy because most legitimate programs do not create or write to files within these folders.

FUNCTION:	**file_copy**
SYNOPSIS:	file_copy `$SOURCE` `$TARGET`
DESCRIPTION:	-
SPECIAL:	Many older malwares copy themselves into the Windows or System directories. It is a calculated move because Microsoft discourages most users from changing or viewing anything in the Windows root directory.

FUNCTION:	**file_create**
SYNOPSIS:	file_create `$PATH\$FILE`
DESCRIPTION:	create a new file.
SPECIAL:	Many newer malwares do not just copy themselves into the host. The new versions of themselves are modified slightly to thwart anti-virus systems.
AMBIGUITY:	Due to ambiguities in the descriptions, file_create could also include the file_copy function.
FUNCTION:	**file_append**
SYNOPSIS:	file_append `$PATH\$FILE`
DESCRIPTION:	write data to file in streams.
FUNCTION:	**file_attrib**
SYNOPSIS:	file_attrib [+-]`$ATTRIBUTES $PATH\$FILE`
DESCRIPTION:	change the attribute or permission of the file. The attributes arguments are hidden, system, read-only or archive; and they can be set (+) or unset (-)
FUNCTION:	**file_modify**
SYNOPSIS:	file_modify `$PATH\$FILE`
DESCRIPTION:	write data to file.
FUNCTION:	**file_property**
SYNOPSIS:	file_property `$PROPERTY $PATH\$FILE`
DESCRIPTION:	change the property of the file. There are many possible arguments for property. Time information alone includes CreationTime, LastAccessTime, LastWriteTime and ChangeTime.
FUNCTION:	**file_rename**
SYNOPSIS:	file_rename `$SOURCE $TARGET`
DESCRIPTION:	-

FUNCTION:	**file_delete**
SYNOPSIS:	file_delete `$PATH\$FILE`
DESCRIPTION:	-
FUNCTION:	**file_execute**
SYNOPSIS:	file_execute `$PATH\$FILE` [`$PARAMETERS`]
DESCRIPTION:	execute file, with optional command line parameters
FUNCTION:	**file_read**
SYNOPSIS:	file_read `$PATH\$FILE`
DESCRIPTION:	read data directly from file.
FUNCTION:	**file_load**
SYNOPSIS:	file_load `$PATH\$FILE`
DESCRIPTION:	load file data into memory.
SPECIAL:	There are a number of ways to accomplish this function; the most common using shared Library APIs. The manual way to do this is by executing the "rundll32.exe" system file: `C:> rundll32.exe $DLL_FILE`
FUNCTION:	**file_access**
SYNOPSIS:	file_access `$PATH\$FILE`
DESCRIPTION:	a non-write operation to the file.
AMBIGUITY:	Used when the description is unclear, could represent the file_read, file_load or file_execute function.
FUNCTION:	**ini_modify**
SYNOPSIS:	ini_modify `$INI_FILE`
DESCRIPTION:	modify system initialization files like win.ini or system.ini.
FUNCTION:	**create_autorun**
SYNOPSIS:	create_autorun `$PATH\Autorun.inf`
DESCRIPTION:	create new Autorun.inf files that define the application to run when disk is inserted or mounted.

FUNCTION:	**dir_create**
SYNOPSIS:	dir_create `$PATH`
DESCRIPTION:	create new directory.
FUNCTION:	**find_dir**
SYNOPSIS:	find_dir `$EXPRESSION`
DESCRIPTION:	search for directories with names matching the expression within the current directory.
FUNCTION:	**find_data_files**
SYNOPSIS:	find_data_files
DESCRIPTION:	search for certain types of data files within the current directory. Examples of these files include files with the following extensions: adb, asp, dbx, htm, php, pl, sht, tbb, wab.
FUNCTION:	**find_bin_files**
SYNOPSIS:	find_bin_files
DESCRIPTION:	search for certain types of executable files within the current directory. Examples of these files include files with the following extensions: com, exe, pif, scr.
FUNCTION:	**search_all_dir_recursive**
SYNOPSIS:	search_all_dir_recursive
DESCRIPTION:	enter all the directories and sub-directories, recursively, starting from the root directory of a system (usually "C:").
FUNCTION:	**search_specific_dir_recursive**
SYNOPSIS:	search_specific_dir_recursive `$PATH`
DESCRIPTION:	enter all the directories and sub-directories, recursively, starting from the path in the argument.

4.4.2 Service

A Windows service is a background application that starts when Windows is booted, conceptually similar to a Unix daemon. Microsoft uses the term "service" loosely because a number of other different concepts are named service as well. Windows provides a Service Control Manager (SCM) interface that manages creating, deleting, starting and stopping of services.

FUNCTION:	**service_create**
SYNOPSIS:	service_create `$SERVICENAME $FILE`
DESCRIPTION:	create a new service, either though the SCM, or by adding new key to the registry.

service_create

FUNCTION:	**service_disable**
SYNOPSIS:	service_disable `$SERVICENAME`
DESCRIPTION:	remove a service, either though the SCM, or by modifying registry key.

FUNCTION:	**service_start**
SYNOPSIS:	service_create `$SERVICENAME`
DESCRIPTION:	start a service, either though the SCM, or by executing the "net.exe" system file: `C:> net.exe start $SERVICENAME`

FUNCTION:	**service_stop**
SYNOPSIS:	service_stop `$SERVICENAME`
DESCRIPTION:	stop a service, either though the SCM, or by executing the "net.exe" system file: `C:> net.exe stop $SERVICENAME`

4.4.3 Process

FUNCTION:	**process_monitor**
SYNOPSIS:	process_monitor
DESCRIPTION:	enumerate all running processes.
FUNCTION:	**process_status**
SYNOPSIS:	process_status `$PROCESS`
DESCRIPTION:	Report process status.
FUNCTION:	**kill_process**
SYNOPSIS:	kill_process `$EXPRESSION`
DESCRIPTION:	terminate any running process started by any file or process, with any identifier matching the given expression.
FUNCTION:	**mutex_create**
SYNOPSIS:	mutex_create `$MUTEXNAME`
DESCRIPTION:	create a new mutex (mutual exclusion) object for synchronization purposes.
FUNCTION:	**mutex_check**
SYNOPSIS:	mutex_check `$MUTEXNAME`
DESCRIPTION:	check for the existence of a mutex object.
FUNCTION:	**event_create**
SYNOPSIS:	event_create `$EVENTNAME`
DESCRIPTION:	create a named event object for synchronization purposes.

4.4.4 Graphical User Interface

The GUI (Graphical User Interface) objects that we are interested in are the dialog boxes, which are special windows used to display information to the user, or to get a response if needed.

FUNCTION:	**hidden_msgbox**
SYNOPSIS:	hidden_msgbox
DESCRIPTION:	create a Windows dialog box in the background, unseen by the user.
SPECIAL:	This is a technique to prevent the user from killing its original application process.

FUNCTION:	**window_box_monitor**
SYNOPSIS:	window_box_monitor `$EXPRESSION`
DESCRIPTION:	enumerate and monitor all the dialog boxes for any information matching the expression.

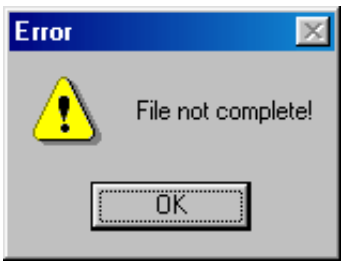


Figure 4.3: Fake Dialog Box displayed by Sober.a

FUNCTION:	**msgbox**
SYNOPSIS:	msgbox `$BUTTONTYPE $TITLE $MESSAGE`
DESCRIPTION:	create a Windows dialog box. As an example, Figure 4.3 is represented by `msgbox OKOnly, "Error", "File not complete!" ;` The common button types of the dialog box are 'OKOnly', 'OKCancel', 'AbortRetryIgnore' and 'YesNoCancel'.

4.4.5 Email

FUNCTION:	**harvest_emails**
SYNOPSIS:	harvest_emails `$FILE`
DESCRIPTION:	search for email addresses within file.
FUNCTION:	**sendmail_with_attachment**
SYNOPSIS:	sendmail_with_attachment `$EMAIL`
DESCRIPTION:	send email with an attachment.
FUNCTION:	**sendmail**
SYNOPSIS:	sendmail `$EMAIL`
DESCRIPTION:	send email without any attachments.
FUNCTION:	**reply_inbox/Outlook_MAPI**
SYNOPSIS:	reply_inbox
DESCRIPTION:	reply to emails inside the INBOX of the user's Outlook program. This is usually accomplished using Outlook's Messaging Application Programming Interface (MAPI) API.

4.4.6 System Information

FUNCTION:	**check_system_date**
SYNOPSIS:	check_system_date
DESCRIPTION:	-
FUNCTION:	**check_system_information**
SYNOPSIS:	check_system_information
DESCRIPTION:	check system information such as the regional locale settings, or which version and service pack of Windows is running.

4.4.7 Network

FUNCTION:	**network_connect**
SYNOPSIS:	network_connect `$HOST $PROTOCOL $PORT`
DESCRIPTION:	any outbound TCP or UDP traffic to remote host.
AMBIGUITY:	Due to ambiguities in the descriptions, network_connect could also include any of the function below with outbound traffic.
FUNCTION:	**scan_network**
SYNOPSIS:	scan_network
DESCRIPTION:	high rate of traffic to existing or non-existent hosts within a subnet.
FUNCTION:	**dns_resolve**
SYNOPSIS:	dns_resolve `$DNSSERVER $DOMAIN`
DESCRIPTION:	perform network domain name resolution on a domain name to get its IP address. The DNS server used is usually predefined in the Windows network configuration.
FUNCTION:	**http_connect**
SYNOPSIS:	http_connect `$URL`
DESCRIPTION:	outbound HTTP traffic, usually to port 80 of remote host.
FUNCTION:	**ntpdate**
SYNOPSIS:	ntpdate `$NTPSERVER`
DESCRIPTION:	outbound NTP traffic, usually to port 123 of remote host. Used to determine the current time and date by synchronizing with the NTP server.
FUNCTION:	**irc_connect**
SYNOPSIS:	irc_connect `$HOST`
DESCRIPTION:	outbound IRC traffic to remote host.

FUNCTION:	**netbios_connect**
SYNOPSIS:	netbios_connect `$HOST`
DESCRIPTION:	outbound NetBIOS traffic, usually to port 135 of remote host.

FUNCTION:	**ping**
SYNOPSIS:	ping `$HOST`
DESCRIPTION:	outbound ICMP ECHO request to remote host.

FUNCTION:	**download_inet**
SYNOPSIS:	download_inet `$URL`
DESCRIPTION:	download file using HTTP or FTP protocol.

FUNCTION:	**listen_port**
SYNOPSIS:	listen_port `$PROTOCOL $PORT`
DESCRIPTION:	open a network port listening for either TCP or UDP protocol traffic.

4.4.8 Windows Network File Sharing

FUNCTION:	**share_enum**
SYNOPSIS:	share_enum
DESCRIPTION:	enumerate or find all Windows network shares within the host's subnet.

FUNCTION:	**remote_share_mount**
SYNOPSIS:	remote_share_mount `$SHARENAME`
DESCRIPTION:	mount Windows network share with either no password, or using predefined usernames and weak passwords.

FUNCTION:	**remote_share_activity**
SYNOPSIS:	remote_share_activity
DESCRIPTION:	any actions performed on, or to a mounted Windows network share.

4.4.9 Registry

The Windows registry is a database that stores the operating system settings and options for Microsoft Windows 95 and later. It contains information and settings for all the hardware, software, users, preferences of the PC and so on. The Registry was introduced to replace most of the text-based .ini files used in Windows 3.x and MS-DOS configuration files, such as the Autoexec.bat and Config.sys.

In most of the write-based registry functions below, the parameter that is most important for providing information to differentiate malwares is the key. Examples of the keys that we are most concern about are those that allow programs to run during or after boot time.

- `$RESTART`
  - `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`
  - `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Runonce`
  - `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\Run`
  - `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`
  - `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Runonce`
  - `HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\Run`
- `$SERVICE`
  - `HKLM\System\CurrentControlSet\Services`
  - `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices`
- `$SHELL`
  - `HKCR\txtfile\shell\open\command`
  - `HKLM\SOFTWARE\CLASSES\txtfile\shell\open\command`
  - `HKLM\SOFTWARE\Classes\exefile\shell\open\command`
- `$DLL_COM`
  - `HKCR\CLSID\{16-byte ID}\InProcServer32`
    this key holds the full path to a DLL file if the "COM" object is implemented as a library. In a nutshell, the DLL file will be launched as a procedure linked to "Explorer.exe" if the 16-byte ID is "`E6FB5E20-DE35-11CF-9C87-00AA005127ED`".

These keys are noteworthy because most legitimate programs do not create or write to them during normal operations.

FUNCTION:	**registry_modify**
SYNOPSIS:	registry_modify `$KEY $VALUE`
DESCRIPTION:	modify the value of an existing registry key.
FUNCTION:	**registry_add**
SYNOPSIS:	registry_add `$KEY $SUBKEY $VALUE`
DESCRIPTION:	add new registry subkey with value data to an existing registry key.
FUNCTION:	**registry_delete**
SYNOPSIS:	registry_delete `$KEY $SUBKEY`
DESCRIPTION:	delete registry subkey.
FUNCTION:	**registry_enum**
SYNOPSIS:	registry_enum `$KEY`
DESCRIPTION:	enumerate all the subkeys of the registry key.
FUNCTION:	**registry_query**
SYNOPSIS:	registry_query `$KEY`
DESCRIPTION:	query the value contained within the registry key.

In the registry_query function, the parameter that is most important for providing information to differentiate malwares is the value data within the key. Examples of these keys are:

$NAMESERVER: IP address of the default DNS Server or Resolver
`HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces\{16-byte ID}\NameServer`

$SMTPSERVER: IP address of the default SMTP Mail Server
`HKLM\SOFTWARE\Microsoft\Internet Account Manager\Accounts\00000001\SMTP Server`

$WAB: Location of user's INBOX file
`HKCU\SOFTWARE\Microsoft\WAB\WAB4\Wab File Name`

$SHELL_FOLDER: Location of user's personal folder
`HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders`

4.4.10 Suspicious Activity or Condition

FUNCTION:	**zombie**
SYNOPSIS:	zombie
DESCRIPTION:	any actions requiring remote activation.

FUNCTION:	**code_injection**
SYNOPSIS:	code_injection `$PROCESS` `$FILE`
DESCRIPTION:	injection of instruction code from file into process not started by the malware.

FUNCTION:	**keylogger**
SYNOPSIS:	keylogger
DESCRIPTION:	captures the user's keystrokes either by hooking to the API I/O library, or kernel's keyboard driver.

FUNCTION:	**date_activated**
SYNOPSIS:	date_activated `$DATE`
DESCRIPTION:	start or terminate malware actions based on time or date.

FUNCTION:	**date_activated_payload**
SYNOPSIS:	date_activated_payload `$DATE`
DESCRIPTION:	start external payload actions based on time or date.

FUNCTION:	**suspicious_file**
SYNOPSIS:	suspicious_file `$FILE`
DESCRIPTION:	any actions involving files with suspicious file names.

Examples of these names are:

- names with double extensions like "`See_Britney_naked.jpg.scr`"
- names with white spaces between extensions like "Anna_Kournikova_nude.jpg␣␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣␣.exe"

FUNCTION:	**suspicious_email_attachment**
SYNOPSIS:	suspicious_email_attachment `$ATTACHMENT`
DESCRIPTION:	any actions involving email attachments with suspicious file names.

4.4.11 Attack Vector

These are all the other ways that a malware can start within the host without relying on explicit user intervention.

FUNCTION:	**start_from_internet_explorer**
SYNOPSIS:	start_from_internet_explorer
DESCRIPTION:	malware started because of Internet Explorer vulnerability.
FUNCTION:	**start_from_outlook**
SYNOPSIS:	start_from_outlook
DESCRIPTION:	malware started because of Outlook or Outlook Express vulnerability.
FUNCTION:	**start_from_windows_exploits**
SYNOPSIS:	start_from_windows_exploits
DESCRIPTION:	malware started because of Windows network service vulnerability.
FUNCTION:	**start_from_network_share**
SYNOPSIS:	start_from_network_share
DESCRIPTION:	malware started remotely through network share.

4.5 Risk Differentiation

In our study of the behavior functions based on the technical description, we learned that just looking at the behavior alone will result in the lost of important information.

We need to include differences in the risk factor based on some of the parameters. For example, there are different levels of risk for `file_copy`, just based on the target directory of the copied file. The risk for a file to be copied into systems directories like the Windows root "`C:\WINNT`" or system "`C:\WINNT\System32`" directory is much higher than any other directories.

Another example is for `file_execute`. As malwares that are not activate carries little risk, the risk for activating a newly created file is higher than an existing system file. In addition, malwares sometimes start applications like the Windows notepad or internet explorer as a form of misdirection, so these behaviors can be used to identify the malwares.

In our current analysis, while we do differentiate certain behaviors based on risk, we do not impose any risk weightage as we do not have enough information to derive the risk modifier for the behaviors and we do not want to do it in an ad hoc way. For example, while `irc_connect` and `ping` are subset of `network_connect`, we treat them as different behaviors. This will affect any analysis of the similar between malwares.

In further work, we would like to study the appropriate modifier or weightage between behaviors, so that two malwares that have the `irc_connect` and `ping` behaviors respectively have a certain similarity factor, instead of none currently.

4.6 Compilation of All Behavior Functions

We compiled a matrix of malwares versus behavior functions based on all the behaviors discussed in the sample malware descriptions in Section 4.4. We will analyze the information from this matrix to support the feasibility of the behavior-based systems and our assumptions. The matrix can be found in Appendix B.

The entries in the matrix are not only categorized by behavior functions, the parameters that introduce different level of risks as discussed in Section 4.5 are also used. Each of the behavior function entries has three possible states: FALSE (0), TRUE (1), MAYBE (2). From the anti-virus descriptions, we notice that some behaviors are certain, while some are optional based on certain conditions. For example, while the behavior of activation of destructive payload by the hacker is very interesting, we are unable to reproduce this. As we aim to take care of all behaviors, we added a MAYBE state to optional behaviors. But compulsory behaviors take precedence in all our analysis.

4.7 Prevalent Behaviors

We believe that malwares from across different families share common behavioral functions. To show this, we compiled the frequency of behavior appearance based on the first malware variant from each of the 15 sample families.

From Figure 4.4, we can see that the most common behaviors are `registry_add`, `file_copy`, `find_data_files`, `file_create` and `harvest_emails`. These functions are related to two complex behaviors: surviving system reboot and finding email addresses. This is not much of a surprise as most of the

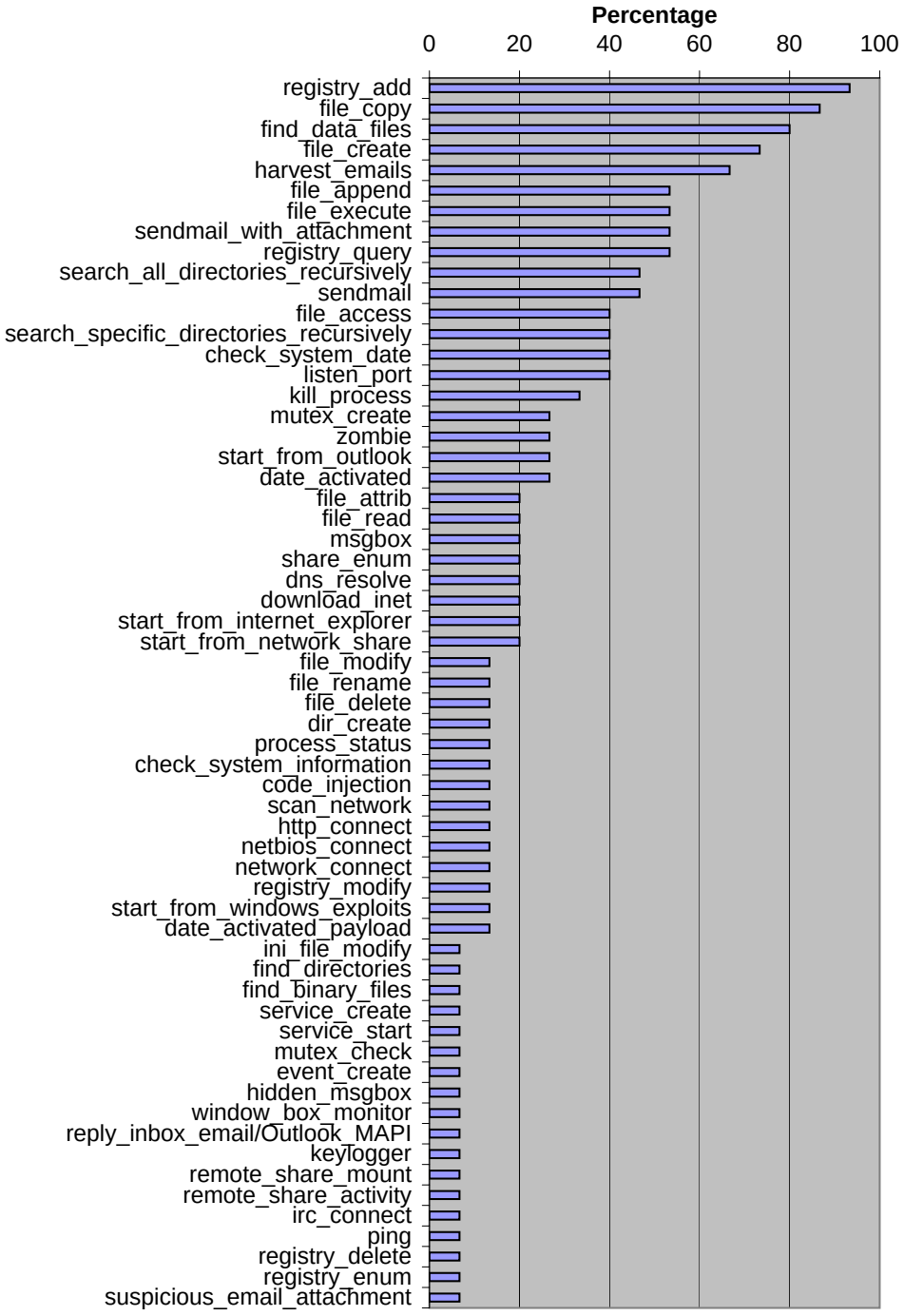


Figure 4.4: Most Prevalent Malware Behaviors

sample malwares are mass mailer viruses. We will use this information in our analysis later.

4.8 Combinations of Independent Behaviors

We can see from Figure 4.4 that no one behavior can identify all the malwares. Thus we will first look at the different combinations of independent or uncorrelated behavioral functions.

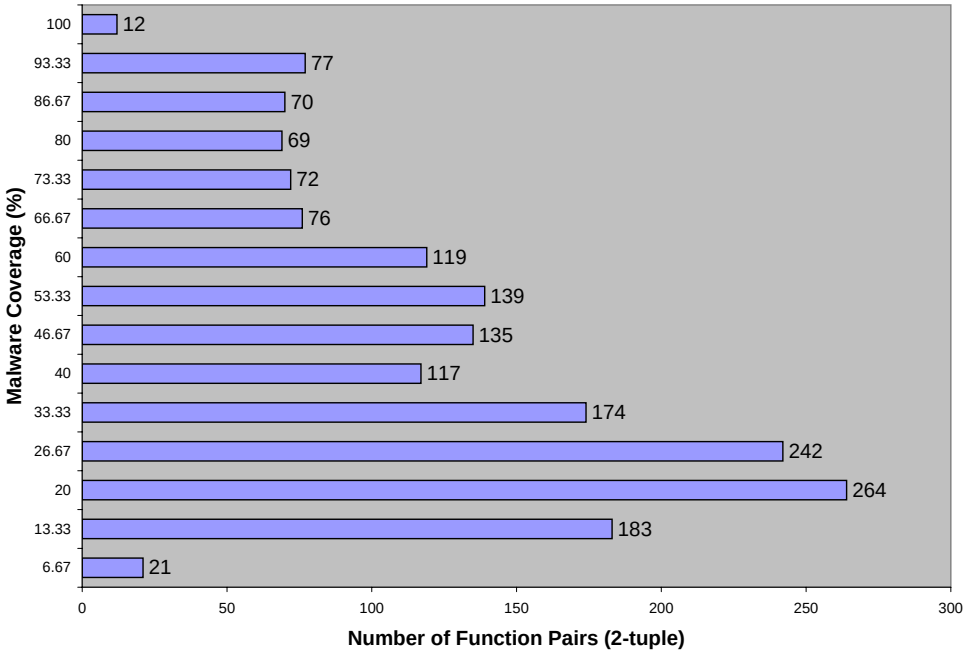


Figure 4.5: Coverage of Malware Behavior Pairs

When we use two behaviors, we can detect all the sample malwares. From Figure 4.5, **12** out of 1770 behavior pairs covers 100% of the detection of malwares. If we use three behaviors, we can see from Figure 4.6 that **849** out of 34,220 behavior triplets offer 100% coverage.

For the behavior pairs that offer 100% coverage, the prevalent behavior is the `registry_add` function. From Table 4.5, we see that `registry_add` accounts for 83.33%, or 10 out of 12 of the behavior pairs. For combinations of three behaviors, `registry_add` accounts for 63.02% or 535 out of 849 of the behavior triplets that offers 100% coverage.

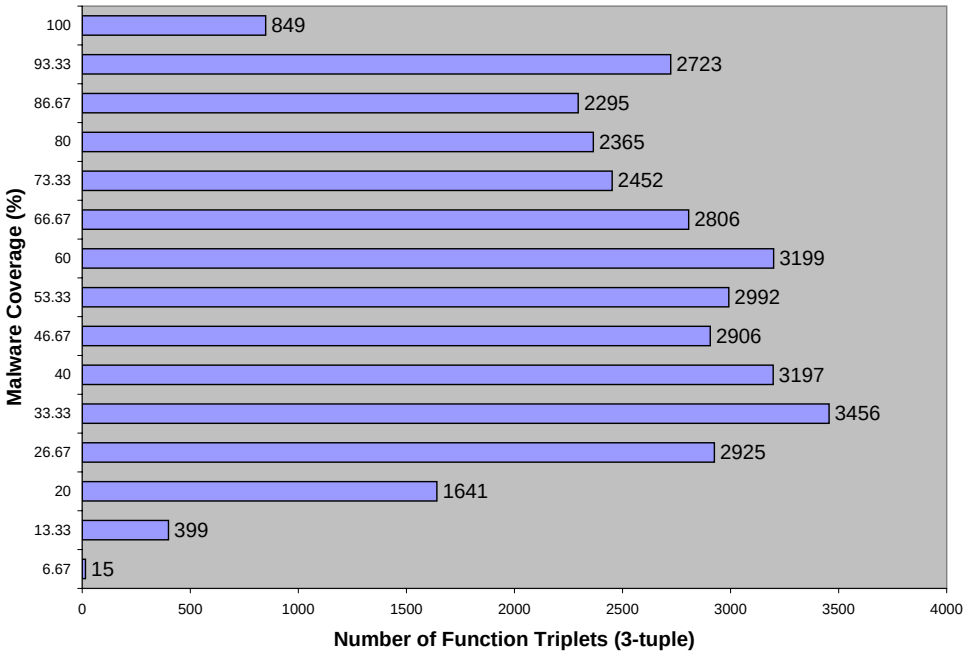


Figure 4.6: Coverage of Malware Behavior Triplets

registry_add,	file_copy
registry_add,	find_data_files
registry_add,	file_create
registry_add,	file_append
registry_add,	registry_query
registry_add,	file_access
registry_add,	search_specific_directories_recursively
registry_add,	file_attrib
registry_add,	msgbox
registry_add,	registry_modify
file_copy,	file_execute
find_data_files,	listen_port

Table 4.5: Behavior Pairs That Cover 100% of Malwares

The first conclusion that we can draw from this analysis is that we do not need to monitor for all of the behaviors to detect malwares. Even a small subset of behavioral functions can do a good job.

The second conclusion is that in both single and combinations of behaviors, some behaviors are more important. We can see from Table 4.5 that `registry_add` is the most important function in behavior pairs. Even when we use three behaviors, this function still carries the most weight.

The third conclusion is that we have choices in the choosing of behaviors to monitor if we just want to detect malwares. For example, while `registry_add` is the dominating function in the behavior triplets as it accounts for 63.02% of the behavior triplets that offers 100% malware coverage, we can also use `file_copy`. The function accounts for 24.15%, or 205 out of 849 of the behavior triplets.

This is important for two reasons: the first is that some behaviors might be very difficult to obtain. The second reason is that it is entirely possible for a malware author to forgo a dominating behavior in order to thwart our behavior-based system. This conclusion tells us that we can use other combinations of behaviors as a competent replacement for any dominating behaviors.

4.9 Complex or Correlated Behaviors

Complex behaviors are formed by correlation of simple behaviors based on certain information. We will provide a few examples in this section.

4.9.1 Survive System Reboot

The most common behavior among all the malwares is the ability to start itself after a system reboot. The most common way to do this is by the combination of two functions: copy itself to the host or create a new file, and add a registry key to run the said file at startup. We can see a sample of this in the example provided in Section 4.4. In Figure 4.7, we see that twenty-two out of twenty-four malwares exhibit this complex behavior.

Adding the file to a startup program key is not the only way. The malware

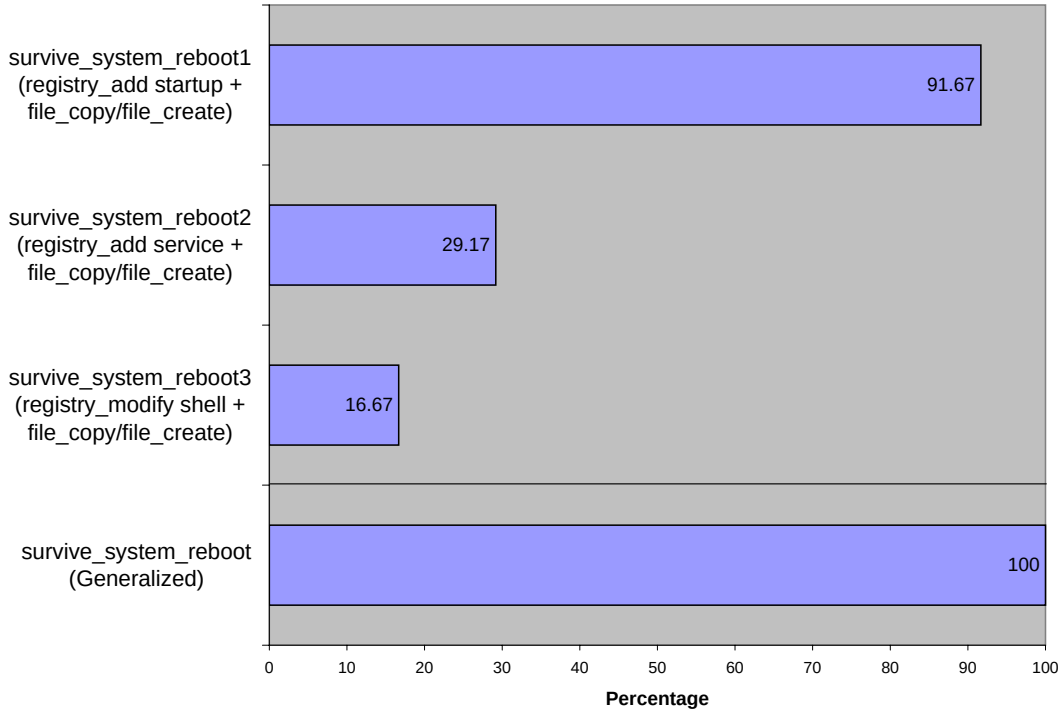


Figure 4.7: Correlated survive_system_reboot Behavior

can also add a new service that runs the file at boot time (`registry_add` service), or modify the registry to run the file whenever files of certain extensions are started (`registry_modify` shell).

If any of the three correlated behaviors in Figure 4.7 is true, we take it that the `survive_system_reboot` behavior is true as well. We can see from the figure that **100%** of the sample malwares exhibits this behavior.

Details about the distribution between malwares and behaviors that formed Figure 4.7 can be seem in Appendix C.1.

4.9.2 Find Email Addresses

The next most common behavior is for the malware to find email addresses in the local host for propagation purposes. The way to do this is by the combination of three functions: search directories recursively, look for data

files like html or text within those directories, and parse the found files for email addresses.

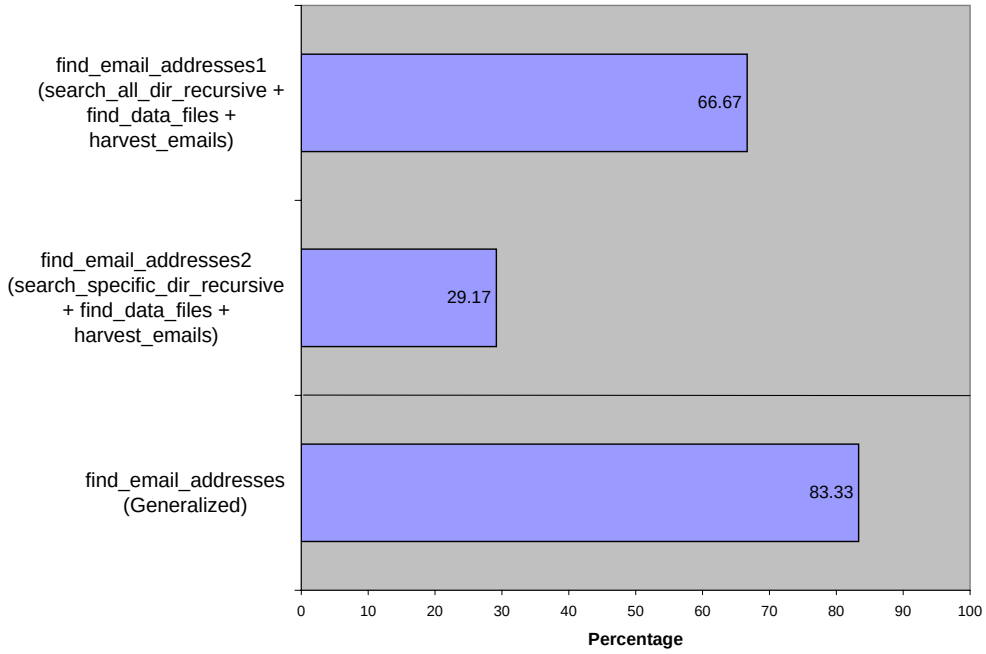


Figure 4.8: Correlated find_email_addresses Behavior

From the descriptions, we know that the malware either search certain directories or all directories starting from the root "`C:\`". If any of the two correlated behaviors in Figure 4.8 is true, we take it that the `find_email_addresses` behavior is true as well. We can see from the figure that only twenty or **83.33%** of the malwares exhibits this behavior.

Details about the distribution between malwares and behaviors that formed Figure 4.8 can be seem in Appendix C.2.

Of the malwares that do not exhibit this behavior, Lovesan.a and Welchia.a are network worms and SpyBot.a is a P-2-P worm. Klez.a only harvest email addresses from the default Windows Address Book and do not search for other files.

4.9.3 Malware Local Replication

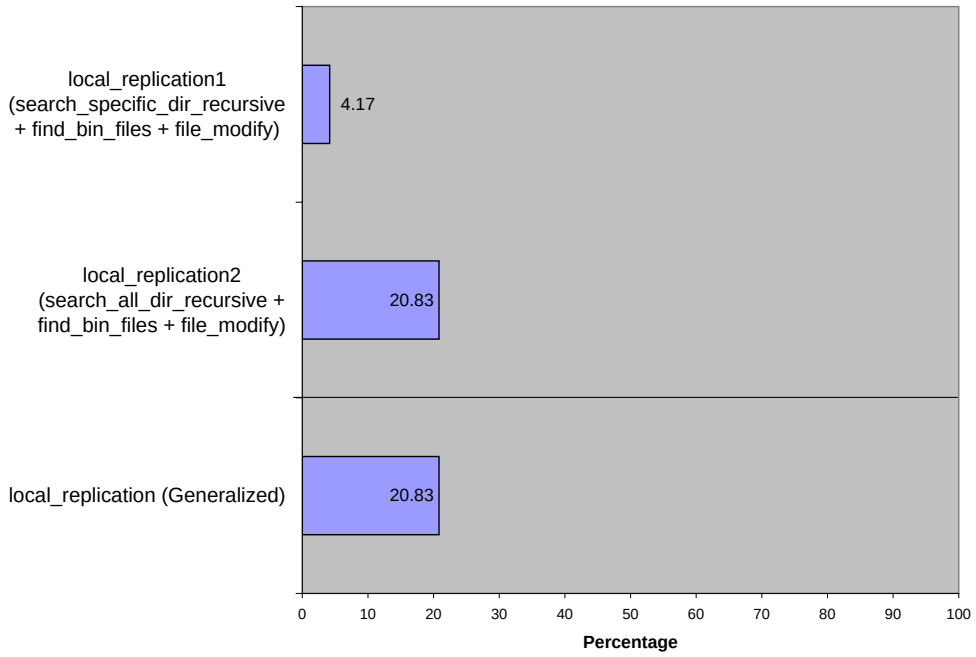


Figure 4.9: Correlated local_replication Behavior

One very interesting observation is that the behavior of `local_replication` does not occur very frequently. From Figure 4.9, only **20.83%** of the malwares exhibits this behavior. This behavior is achieved by the combination of three functions: search directories recursively, look for binary files with extensions like exe or com within those directories, and modify the located files.

Details about the distribution between malwares and behaviors that formed Figure 4.9 can be seem in Appendix C.3.

This is strange because local replication is the hallmark of most viruses. One possible reason why local replication to executable files is not popular could be due to the system restore feature in Windows 2000 and above. We noticed in our analysis of several malwares that Windows performed cryptographic checksum verifications on the system files that were changed. If

the checksum of the file was incorrect, the changed file would be overwritten by the original version of the file.

4.10 Study of Cross Family Behaviors

4.10.1 Malware Naming and Classification Convention

As we want to study the behavioral similarity between malwares within a family, and across different families, we will provide a short background on the current naming and classification convention used by the anti-virus companies.

After decades of virus research, there is still no standard way to name a malware [53]. While there are attempts to standardize the naming convention like the Common Malware Enumeration (CME) Project [13], most researchers still continue the decade old tradition of ad hoc naming due to the commercial pressure to be the first to detect more new malwares.

At best, we have guidelines [36, 23] on how not to name a malware, and the CARO (Computer Anti-Virus Research Organization) Malware Naming Scheme [7, 9] to categorize the malware into different types.

The general format of a full CARO malware name is

`[⟨type⟩://][⟨platform⟩/]⟨family⟩[.⟨group⟩][.⟨length⟩].⟨variant⟩`
`[⟨modifiers⟩][!⟨comment⟩]`

where the items in square brackets are optional. Most anti-virus companies use a variation of this format, and we will take Kaspersky Lab's naming of "Email-Worm.Win32.Bagle.at" as an example:

modifiers -	type .	platform .	family .	variant
Email -	Worm .	Win32 .	Bagle .	at

Malware Family

The malware family name is basically the initial name given to a malware that is significantly different from the anti-virus companies' specification of all the other known malwares. We will provide a few example of how malwares were named to give the reader an idea how ad hoc the process actually is.

We have the totally random ones like: the Code Red worm [27] named after a cola, and the Melissa worm [30] named after a lap dancer in Florida. Then, we have those named from keywords within the malware source code: the Klez virus [16], and MyDoom [5], whose source code included "`mydom`" (short for "my domain").

The Nyxem [15] virus was named because it was the first virus to launch a DDoS attack against the "New York Mercantile Exchange" website (`www.nymex.com`), and the Sasser virus was named because it targets the Local Security Authority Subsystem Service (LSASS) [32] of the Windows operating system.

Malware Variant

The naming tradition affects how the anti-virus companies classify malwares into the different existing families as variants. There is no fixed classification scheme, and could be based on attributes like the malwares source code, keywords found within the malware, exploits used or actions taken by the malware. As the actual classification process varies between the

different anti-virus companies according to the malware researcher's bias, a malware could be classified into different families by different researchers. For example, the same worm was named `W32/Mydoom@MM`, `Novarg` and `Mimail.r` respectively by Network Associates, Symantec Corp and Trend Micro [29].

In spite of this problem, we believe it is likely that malware variants within a family are similar because of their shared attributes. It would be interesting to see if the similarity extends to our behavior-based approach.

4.10.2 Malware Similarity Matrix

Our assumption is that malwares within the same family have more behavioral functions in common, as opposed to malwares from other families. If this is true, then it should be possible to detect previously unseen malwares based on their similar set of functions. To confirm this, we formed a similarity matrix (Table 4.7) from behaviors of all twenty-four malwares introduced earlier.

$$SimilarityIndex_{a,b} = \frac{Cardinality(BehaviorSet_a \bigcap BehaviorSet_b)}{Cardinality(BehaviorSet_a \bigcup BehaviorSet_b)} \times 100\ (\%)$$

where
$SimilarityIndex_{a,b}$ is the similarity factor between $Malware_a$ and $Malware_b$.
$BehaviorSet_{all}$ is the set of all behavior functions studied.
$BehaviorSet_a = \{$m: m is function in $BehaviorSet_{all}$ that exist in $Malware_a\}$
$BehaviorSet_b = \{$n: n is function in $BehaviorSet_{all}$ that exist in $Malware_b\}$

The Similarity Index between malwares is based on existence of behaviors alone. The more behavioral functions the two malwares have in common, the higher the score. Currently, only functions that are compulsory were used. Functions that only activate in conditions that we cannot replicate are not used. In further work, we would like to add in these optional

functions with a modifier so that they are less important than compulsory function. We would also like to add in weightage for the different levels of risks between certain behaviors as discussed in Section 4.5 in future works.

	Klez.a	Klez.e	Klez.h	Zafi.a	Bagle.a	Bagle.z	Bagle.ai	Bagle.at	Ganda	Gibe.a	Lentin.a	LovGate.a	LovGate.ad	LovGate.b	Lovelorn.a	Lovesan.a	Mimail.a	Mydoom.a	Sober.a	Sober.f	Sober.g	Sobig.a	SpyBot.a	Welchia.a
Klez.a		52	33	29	17	25	21	19	24	11	13	20	22	21	30	10	19	18	17	14	18	19	20	6
Klez.e	52		43	43	25	33	32	26	31	14	9	17	30	21	26	10	23	35	21	23	22	14	16	6
Klez.h	33	43		33	24	29	28	22	39	12	15	19	27	24	35	3	26	20	25	27	21	12	14	6
Zafi.a	29	43	33		41	38	37	34	30	19	15	19	24	24	30	7	21	40	25	33	32	12	23	10
Bagle.a	17	25	24	41		48	45	36	19	28	4	18	21	28	20	9	26	28	25	28	26	15	17	12
Bagle.z	25	33	29	38	48		68	56	23	23	7	19	22	24	30	11	21	35	30	27	32	12	23	10
Bagle.ai	21	32	28	37	45	68		74	26	27	3	22	27	26	29	19	17	34	24	26	30	16	22	13
Bagle.at	19	26	22	34	36	56	74		24	25	6	21	26	25	27	18	19	32	22	24	28	19	21	12
Ganda	24	31	39	30	19	23	26	24		19	19	24	21	29	27	6	41	23	16	17	16	17	8	5
Gibe.a	11	14	12	19	28	23	27	25	19		28	25	21	26	16	11	35	23	15	12	20	32	18	6
Lentin.a	13	9	15	15	4	7	3	6	19	28		15	13	9	11	0	21	12	9	5	9	15	4	4
LovGate.a	20	17	19	19	18	19	22	21	24	25	15		36	72	19	9	24	22	19	16	19	20	14	5
LovGate.ad	22	30	27	24	21	22	27	26	21	21	13	36		35	20	14	15	22	16	14	17	14	16	18
LovGate.b	21	21	24	24	28	24	26	25	29	26	9	72	35		24	6	29	30	24	26	21	21	19	5
Lovelorn.a	30	26	35	30	20	30	29	27	27	16	11	19	20	24		4	32	17	32	42	33	17	19	3
Lovesan.a	10	10	3	7	9	11	19	18	6	11	0	9	14	6	4		8	16	9	5	21	10	8	42
Mimail.a	19	23	26	21	26	21	17	19	41	35	21	24	15	29	32	8		18	22	24	29	30	7	3
Mydoom.a	18	35	20	40	28	35	34	32	23	23	12	22	22	30	17	16	18		20	26	17	10	23	15
Sober.a	17	21	25	25	25	30	24	22	16	15	9	19	16	24	32	9	22	20		47	53	16	13	0
Sober.f	14	23	27	33	28	27	26	24	17	12	5	16	14	26	42	5	24	26	47		50	18	14	0
Sober.g	18	22	21	32	26	32	30	28	16	20	9	19	17	21	33	21	29	17	53	50		24	14	4
Sobig.a	19	14	12	12	15	12	16	19	17	32	15	20	14	21	17	10	30	10	16	18	24		4	0
SpyBot.a	20	16	14	23	17	23	22	21	8	18	4	14	16	19	19	8	7	23	13	14	14	4		7
Welchia.a	6	6	6	10	12	10	13	12	5	6	4	5	18	5	3	42	3	15	0	0	4	0	7	

Table 4.7: Malware Similarity Matrix

4.10.3 Analyzing the Similarity Matrix

It is very hard to analyze the large similarity matrix, so we extracted some of the more interesting information here.

In most cases, malwares are more similar to later variants of the same family than to the earlier variants. This gives us more confidence that we can use behaviors from an earlier malware variant to detect a newer one.

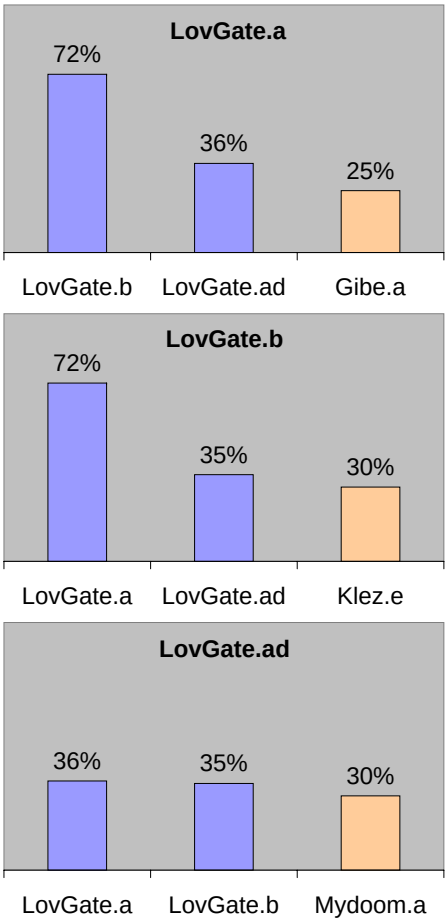


Figure 4.10: Top Three Most Similar Malwares To LovGate Family Variants

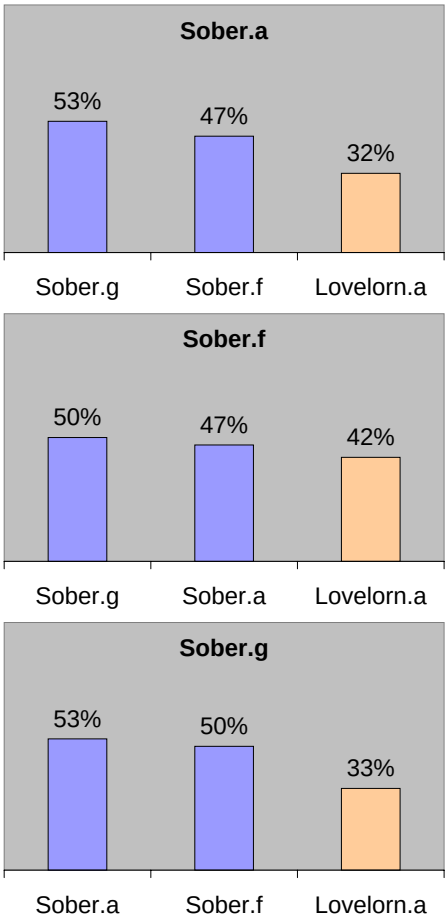


Figure 4.11: Top Three Most Similar Malwares To Sober Family Variants

We can see from Figure 4.10 and 4.11 that in the LovGate and Sober families, the variants within the same family have higher similarity index than from other families. This fits our assumption that malwares have a higher

inter-family similarity.

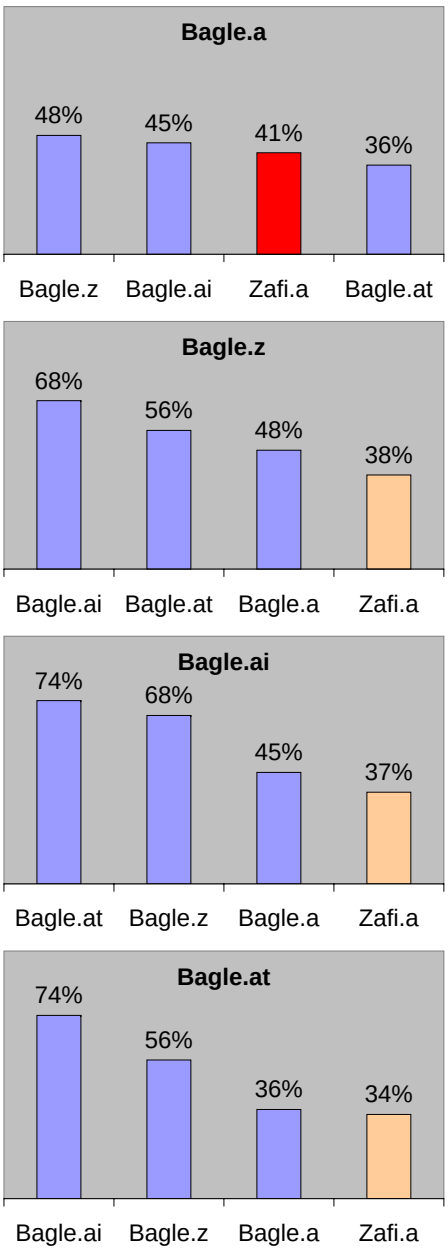


Figure 4.12: Top Three Most Similar Malwares To Bagle Family Variants

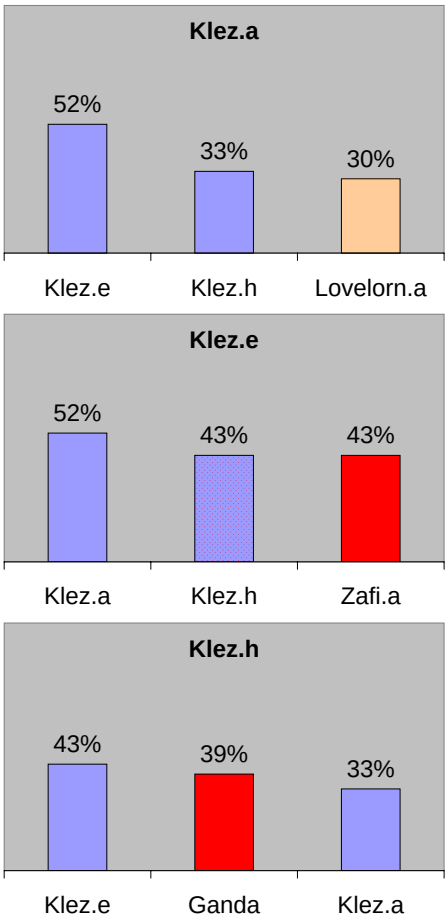


Figure 4.13: Top Three Most Similar Malwares To Klez Family Variants

But in the Bagle family, Bagle.a is more similar to Zafi.a than Bagle.at. In the Klez family, Klez.h is more similar to Ganda than Klez.a, and Klez.e has the same similarity index for both Klez.h and Zafi.a. In these anomalies, the similarity indexes for these intra-family malwares are higher than

the average.

We believe that there are two main possibilities for this anomaly. The first and most likely possibility is that the Similarity Index we used was too simple to study the intricate behavioral relationships between malwares. Modifying the Similarity Index equation based on the previously proposed suggestions in Section 4.10.2 will result in a more accurate score, and may correct this problem.

The second possibility is that the current classification scheme is unsuitable for our behavior-based approach, and a new paradigm is required. This can be the focus of our future work.

Chapter 5

Experimental Methodology

5.1 Choice of Sensor

5.1.1 Experimental Objectives

The aim of our experiment is to get the list of behaviors described in Section 4.4 that we are interested in. The choice of the sensor is very important as it directly affects how we analyze the malware behaviors. We imposed the following criteria for our choice:

- Must be able to capture information on most of the behaviors
- Data output must be semantically rich enough to reveal higher level behaviors
- Data output must be in format flexible enough to allow statistical analysis
- Must not impact system performance too much
- Must not adversely affect "normal" malware operations

5.1.2 Static Analysis versus Dynamic Monitoring

The first decision that we have to make is to choose to either perform static analysis on the malware binary without executing it, or actually execute the binary and observe its interaction with the operating system environment.

Static Analysis

Static analysis of a malware binary let us find out exactly how a malware work, the resources that it uses, and the objects it carries within its payload (files, scripts, HTML, GUI, passwords, commands, control channels, and so on). The API system calls used by the malware can also be reverse engineered from the binary, for example using SAVE [59] (Static Analyzer for Vicious Executable). Most anti-virus solutions use this approach.

The problem with static analysis is that it is not very effective against polymorphic or metamorphic malwares. While it is possible to recover the code portions that polymorphic malwares attempts to hide via encryption or encoding by studying the API system calls used by the binary, there is no quantitative study to show its effectiveness. Also, the general consensus on the effectiveness of this approach against metamorphic malwares is dismay. Metamorphic malwares constantly mutates its payload by using different registers, inserting junk code like no operations (NOP's), and jumping over (JMP) or rearranging code segments.

Dynamic Monitoring

Dynamic monitoring does not have the same problem with polymorphic or metamorphic malwares. No matter how much the binary code changes, the actions of the malware do not change. Since we look at behaviors, the few ways for the malware to escape detection are by not performing any known malicious actions, performing only novel actions, or taking out the sensor system before its own detection. (Protection of the detection system is not covered within the scope of this thesis)

The weakness of dynamic monitoring is that it might not capture all the behaviors of the malware. Some behaviors might require certain conditions

to be met before activation. For example, a malware might only perform destructive actions on the hard drive or engage in a DOS attack only on certain dates. A multi-vector malware might need certain software to be installed in order to propagate in a different way; for example, a mass mailer virus that can also be spread via the Kazaa distributed peer-to-peer file sharing service.

We choose to use dynamic monitoring because it relates well to our behavior-based approach and we hope to implement a real-time behavior-based detection system in the future. Despite its weakness, we believe that it is a good guide to malware behaviors and it can complement static analysis well. Finally, it is our assumption that we do not need to catch all the malware behaviors for detection or classification.

5.1.3 Sensor Level

The next question that we have to ask is where do we monitor, and how much sensor details do we want. Let us look at the following three levels:

- Instruction set level
- System call level
- Application level

Instruction Set Level

The trade-off is that at the lower instruction set level, we have higher coverage but lower semantic information. That means, it will be harder for malwares to hide from the sensor, but it will also mean that it is harder to get high-level behavioral information from the large stream of instruction codes that will be generated from the monitoring.

Application Level

At the higher application level, we have lower coverage but higher semantic information. Windows itself provides Windows Event, Security and Application logs, and performance counters that provides a good source of information. Unfortunately, the lack of details and flexibility of these tools makes them a bad sensor choice.

There are also a number of tools that we can use to look for specific behaviors. For example, Sysinternals offers a great range of tools that can study a lot of different Windows behaviors in real-time; like Filemon [48] that monitors all file system activities, or Regmon [49] that monitors all registry activities. These tools can offer very specific and detailed behavioral information with just a small amount of generated data.

The downside of using these tools is that every new behavior that we want to cover requires additional tools. We lose flexibility, as we must know exactly what behaviors we want before our experiments. Furthermore, it is very hard to correlate information from different tools accurately.

System Call Level

The middle ground between the instruction set and application level is the system call level. At this level, we look at information that passes from the process to the kernel: system call names, arguments, and result values. In many cases, system calls happen at a relatively low frequency compared to machine instructions.

Another reason to choose the system call level is because of our assumption that malware writers want portability, like most Win32 developers. Most of the malwares discussed in Section 4.1 are written in C or Visual Basic,

which are highly dependent on shared libraries or APIs that are common over different versions of Windows. It is likely that similar system calls will be used by these common APIs.

Many intrusion detection research on Unix based systems uses API system calls for their sensor. That is because Unix has a small set of API system calls that is open and well documented. These system calls combine to form complex actions. On the other hand, Windows provides a large set of APIs and system calls where the same function can be accomplished via several different ways using different system calls. To ensure back-compatibility, the number of Windows APIs and the system calls within these APIs are increasing at every upgrade.

While we acknowledge that it is difficult to extract behaviors from Windows system calls, this level provides the best trade-off in terms of capabilities and coverage.

5.2 Windows Internal Architecture

The details of the internal workings of Windows, especially NT's architecture, are beyond the scope of this thesis. We will discuss some relevant details under the assumption that the reader has some familiarity with Windows. We refer the interested reader to Russinovich's books [39, 42] for more details.

The Windows NT's architecture consists of two main layers: user and kernel. The mode of a process depends on which layer it is working in. Processes in the user mode have limited rights to system resources, while the kernel mode has unrestricted access to the system memory and external

devices. While NT's kernel is structured like a microkernel, it is in essence a monolithic kernel.

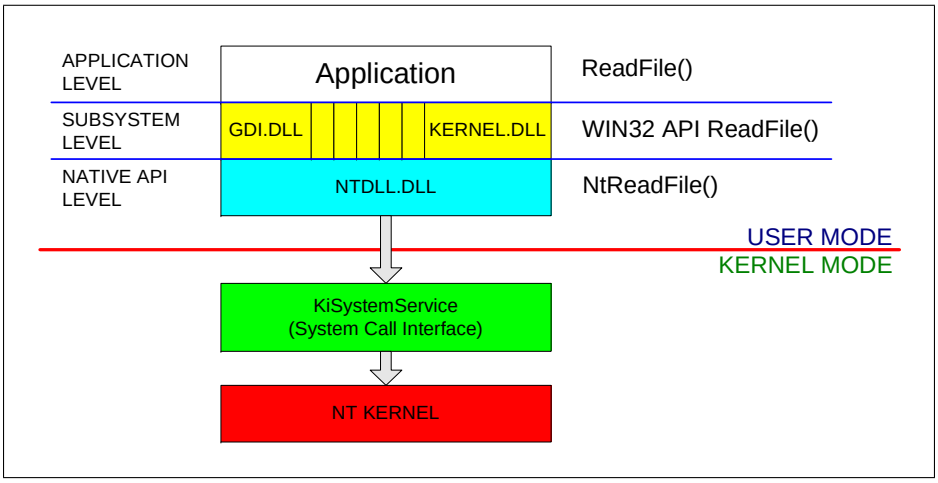


Figure 5.1: Windows API Call

To provide access from the user to kernel mode, Microsoft provides several user subsystems. The most common one is the Win32 API, while others include the OS/2 and POSIX API. But there is a hidden API that NT uses internally, the Native API. This API is obfuscated from most programmers, with hardly any documentation provided by Microsoft. The Windows NT Native API is used to call operating system services located in kernel mode from the user mode by higher level APIs such as the Win32, OS/2, POSIX, Winsock or .NET APIs. An example of an application level API call is shown in Figure 5.1. The technical details of the Native API are beyond the scope of this paper, and we refer the reader to [38] for more detailed information.

5.3 Choice of API Level Monitoring

The next problem we face is choosing the API to monitor. From the discussion in Section 5.2, we can roughly separate the APIs into two groups: the higher level shared library APIs, and the lower level native API.

- subsystem (Win32, OS/2, POSIX) and application (Winsock, .NET)
- native

Our choice is to either monitor only the native API, or all the APIs.

5.3.1 Advantages of Native API

Higher level shared library APIs (Win32, OS/2, POSIX) must interface to kernel via the lower level native API, thus the coverage is very wide for the native API. In fact, hooks in the native API can provide global control over the system.

As even assembly-based programs trigger native API system calls when performing functions such as accessing files, it is very hard for most malwares to hide from such a low level sensor. While it is possible to write malwares that do not use standard API calls [4], doing so is very difficult and will break any compatibility between Windows versions.

Finally, the performance hit to the system for monitoring all the APIs is very high. When we first experimented with Rohitab's API Monitor [37] to capture system calls from all APIs, the system slowed down until it crashes when we ran the Welchia worm. Since we hope to implement a real-time detection system in the future, we cannot afford to monitor all the APIs.

5.3.2 Limitations of Native API

The main disadvantage of monitoring native API is that we can only see what the malware ask the operating system to do. The decision-making process, or logic, of the malware cannot be inferred from the system calls.

In addition, system calls from other higher-level APIs such as Winsock are not monitored, so we cannot get detailed information about the network

traffic. But the native API does provide clues that indicate network activities. For example, in a number of traces, any successful NtCreateFile call to the "\Device\RasAcd" device is indicative of SMTP activity.

Finally, the native API is not as descriptive as the higher level APIs. For example, the single `CopyFile()` Win32 API need to be represented by a sequence of native API system calls.

5.4 Chosen Implementation

We would like to extract host behaviors from Native API system calls alone. The tool we have chosen to implement is based on BindView's *strace for NT* [12], as the source code of strace is available under Open Source license. While our ultimate aim is to modify strace to serve as a real-time defense module, we will capture all the data to disk so that we can work off-line for now.

To show the reader the output format of strace, we quote the following from strace's readme file:

```
1 133 139 NtOpenKey (0x80000000, {24, 0, 0x40, 0, 0, "\Registry\Machine [...]
2 133 139 NtCreateEvent (0x100003, 0x0, 1, 0, ... 8, ) == 0x0
3 133 139 NtAllocateVirtualMemory (-1, 1243984, 0, 1244028, 8192, 4, ... ) == 0x0
4 133 139 NtAllocateVirtualMemory (-1, 1243980, 0, 1244032, 4096, 4, ... ) == 0x0
5 133 139 NtAllocateVirtualMemory (-1, 1243584, 0, 1243644, 4096, 4, ... ) == 0x0
6 133 139 NtOpenDirectoryObject (0x3, {24, 0, 0x40, 0, 0, "\KnownDlls"}, ... 12, )
== 0x0
7 133 139 NtOpenSymbolicLinkObject (0x1, {24, 12, 0x40, 0, 0, "KnownDllPath"}, ...
16, ) == 0x0
8 133 139 NtQuerySymbolicLinkObject (16, ... "C:\WINNT\system32", 0x0, ) == 0x0
9 133 139 NtClose (16, ... ) == 0x0
⋮
```

```
The first column is an identity, which lets you match up calls
that don't complete immediately (and are broken onto two lines).
The second and third columns are the process and thread ids of
```

```
the thread making the call.  Next is the name of the system call,
the input parameters, three dots (...), then output parameters,
and the return code.
```

5.5 Experimental Environment

5.5.1 Virtualization versus Emulation

There are advantages and disadvantages to both virtualization and emulation of the platform.

Emulation, or sandboxing, allows us better control because the malware binary is not executed in a real environment, but within a "jail" operating system emulated by software. Thus, it is possible to monitor both native and application system calls. It is also more convenient when we want to capture the system call traces from a large batch of malwares. Unfortunately, emulated environments are extremely restrictive: adding in new programs or implementing complex network services that interact with the sandbox can be very troublesome.

Virtualization, on the other hand, emulates a PC and the operating system is installed within a virtual machine. From the operating system's point of view, everything from the hardware to the network environment is real. This means we can install whatever software we want, and the network configuration is decoupled from the virtual machine.

We acknowledge that some malwares might change their behaviors or do not activate, if they detect the virtual environment. This is to prevent attempts at reverse engineering the malwares. The most common way to detect virtualization is by checking the hardware configurations as most

virtualization software uses fixed names for the virtual hardware. There are no solutions at this time, but vendors such as VMWare are working to rectify this flaw in their products.

5.5.2 Platform Operating System

As we decided to study Windows malwares, we have to decide which version to run the malwares on. The possible versions of Windows that we could use are shown in Table 5.1.

The research platform we chose for the victim is the Windows 2000. The first reason is because next to Windows XP, Windows 2000 is the second most deployed version in both the home user and corporate market. The second reason is because this operating system was targeted by a large number of malwares in the past, and is still targeted in newer attacks in addition to Windows XP. A number of older malwares may not exhibit all their behaviors in XP. The large amount of available malwares is important in the study of behaviors.

Year	1993	1994	1995	1996	1997	1998	1999	2000	2003	2005
Home User Market	-	-	-	95 OSR2	95 OSR2.1, OSR2.5	98	98 SE	ME, XP	XP (MC)	-
Enterprise Market	NT 3.1	NT 3.5	NT 3.51	NT 4.0	-	-	-	2000	2003	2003 R2

Table 5.1: Versions of Microsoft Windows

5.5.3 Network Configuration

To prevent any accidental release of malwares during experiment, virtual machines are used within a single host PC to simulate an isolated network environment. VMWare Workstation [51] is the virtualization software used to provide a victim guest running Windows 2000 Professional and a Gate-

way providing faked network services running Fedora Core 4 Linux.

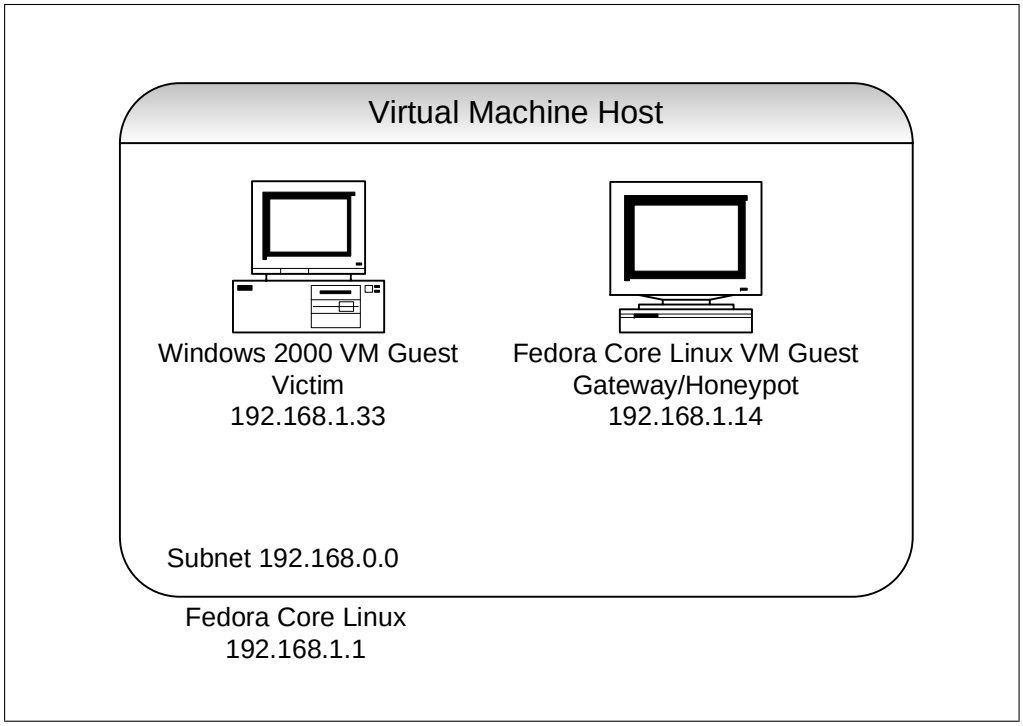


Figure 5.2: Experiment Virtual Network Diagram

The services provided by the Gateway are:

- DNS Server resolving all URL to a single address
- Sendmail mail server
- dovecot IMAP server
- samba file server providing Windows Network Share

In addition, the Gateway also runs *honeyd* [34], a low interaction honeypot that responds to any IP addresses simulating Windows 2000 with the following services:

- NetBIOS service
- MS Exchange POP3 service
- MS Exchange NNTP service
- MS Exchange IMAP service
- MS Exchange SMTP service
- LDAP service
- IIS Web server

- MS ftp service
- VNC service

In future work, we plan to add in a dummy Windows 2000 VMWare guest that had a previous file share relation with the victim because none of the malwares attacked the samba file server. We also plan to include fake IRC and P2P servers.

5.5.4 Honeytokens: Email Addresses and Files

To coax the full set of email behaviors from malwares, we can use the concept of honeytokens. Honeytoken [45] is a system resource whose value lies in unauthorized or illicit use of that resource. In this case, we can create fake email addresses that the malwares can find and propagate to.

To make sure the generated fake addresses are used by as many malwares as possible, we can make use of the descriptions from the anti-virus companies that include patterns that malwares avoid. See Table 5.2 for some examples.

Bagle.a	@hotmail.com, @msn.com, @microsoft, @avp, .r1
Bagle.z	abuse, admin, anyone@, @avp., bsd, bugs@, ...
LovGate.ad	.gov, .mil, accoun, acketst, admin, arin., ...
Mydoom.a	@*.gov, @*.mil, @*acketst*, ... , *accoun*@, *anyone*@, ...
Sober.f	@arin, @foo., @iana, @ikarus., @kaspers, @messagelab, ...

Table 5.2: Examples of Email Patterns Avoided by Malwares

In addition to putting these fake addresses in the Outlook or Windows Address Books, we can also embed them in files that the malwares can find. Rather than to randomly place and name these files, we can refine the location and file names using the anti-virus company descriptions as well. Table 5.3 shows some examples of file extensions searched by the malwares.

As these files are honeytokens and should not be accessed by any legal programs, any file read or load is highly suspicious and can provide another behavioral function that we can use.

Bagle.a	wab, txt, htm, html
Bagle.z	adb, asp, cfg, cgi, dbx, dhtm, eml, htm, jsp, mbx, mdx, ...
Ganda	eml, *htm*, dbx
Gibe.a	bmp, cpp, doc, htm, html, jpg, mpeg, mpg, txt, xls
Klez.a	asp, htm, html, php
Klez.h	asp, bak, c, cpp, doc, htm, html, jpg, mp3, mpeg, mpg, ...
Lentin.a	htm*
LovGate.a	ht*
LovGate.ad	adb, asp, dbx, htm, php, pl, sht, tbb, TXT, wab
Lovelorn.a	EML, dbx, htm, *ITEM, *BOX
Mydoom.a	adb, asp, dbx, htm, php, pl, sht, tbb, txt, wab
Sober.a	abc, ade, adp, asp, cfg, doc, dsp, dsw, eml, fdb, htm, ...
Sober.f	abc, abd, abx, adb, ade, adp, adr, asp, bas, cfg, cgi, ...
Sobig.a	dbx, eml, htm, html, txt, wab

Table 5.3: Examples of File Extensions Searched by Malwares

5.6 Experimental Progress

To test the feasibility of using the behaviors to detect malwares, we look at some of the most common malwares and normal applications to date. For each malware, we traced its live activity for fifteen minutes, more than enough time for any recurrent behaviors to surface. While we do not claim any of the traces to be representative of normal activities, these traces have been gathered in environments as deterministic as possible. We believe that they can help us detect malware behaviors by providing a baseline of activities.

5.6.1 Traces of Common or Commercial Applications

To have a baseline to compare malware against, traces of the following applications common to most home users and office workers were captured

and analyzed. This baseline comparison is very important in the study of false positives rates for our approach.

For the normal applications, we used the host as any normal user would have, and traced its execution using various lengths of time depending on the action we are trying to capture, from five to fifteen minutes. For some actions like the browsing the Internet, the target system is connected to a live network. Other actions that do not require Internet access are restricted to our isolated network. The applications used and actions taken are shown in Table 5.4.

Programs	Actions
Adobe Acrobat Reader 6	Open file in Explorer.
Ghostgum GhostScript Viewer 4.7	Open file in Explorer.
Internet Explorer 5.0.2920.0	Normal browsing.
ICQ Messenger 2001b Build 3659	Connect to server.
MSN Messenger 5.0	Connect to server.
Windows Media Player 6.4.9.1109	Open file in Explorer and from URL.
Nullsoft Winamp Audio Player 5.094	Open file in Explorer.
Microsoft Access 2000 9.0.0.2719	Open file in Explorer, Save file.
Microsoft Excel 2000 9.0.0.2719	Open file in Explorer, Save file.
Microsoft Outlook 2000 9.0.0.2416	Receive/send mail with IMAP.
Microsoft Powerpoint 2000 9.0.0.2716	Open file in Explorer, Save file.
Microsoft Word 2000 9.0.0.2717	Open file in Explorer, Save file.
Microsoft FrontPage 2000 4.0.2.2717	Open file in Explorer, Save file.
WinZip 7.0	Open file in Explorer, Uncompress file.
WinRAR 3.3	Open file in Explorer, Uncompress file.

Table 5.4: Normal Applications Studied

5.6.2 Traces of Malwares

We attempted to capture the traces of all twenty-four malwares discussed in Section 4.2. We encountered some problems in trying to capture the traces of certain malwares. These problems include: malware samples unable to execute on our virtual machine (possibly because of damaged binaries), incompatibility with the strace monitoring program (malware process dies if

being monitored), and failure of the malware to launch its payload.

Because of these problems, we only managed to successfully obtain traces of eleven malwares as shown in Table 5.5.

Malwares	Status	Comments
Email-Worm.Win32.Bagle.a	FAIL	fail to replicate.
Email-Worm.Win32.Bagle.z	FAIL	fail to activate.
Email-Worm.Win32.Bagle.ai	SUCCESS	
Email-Worm.Win32.Bagle.at	SUCCESS	
Email-Worm.Win32.Ganda	SUCCESS	
Email-Worm.Win32.Gibe.a	FAIL	fail to replicate.
Email-Worm.Win32.Klez.a	FAIL	clash with sensor.
Email-Worm.Win32.Klez.e	FAIL	fail to activate.
Email-Worm.Win32.Klez.h	FAIL	fail to activate.
Email-Worm.Win32.Lentin.a	FAIL	fail to activate.
Email-Worm.Win32.Lovelorn.a	SUCCESS	
Email-Worm.Win32.LovGate.a	SUCCESS	
Email-Worm.Win32.LovGate.b	SUCCESS	
Email-Worm.Win32.LovGate.ad	FAIL	fail to replicate.
Email-Worm.Win32.Mimail.a	SUCCESS	
Email-Worm.Win32.Mydoom.a	FAIL	fail to replicate.
Email-Worm.Win32.Sober.a	SUCCESS	
Email-Worm.Win32.Sober.f	SUCCESS	
Email-Worm.Win32.Sober.g	SUCCESS	
Email-Worm.Win32.Sobig.a	FAIL	fail to replicate.
Email-Worm.Win32.Zafi.a	FAIL	fail to activate.
Net-Worm.Win32.Welchia.a	SUCCESS	
P2P-Worm.Win32.SpyBot.a	FAIL	fail to activate.
Worm.Win32.Lovesan.a	FAIL	fail to replicate.

Table 5.5: Trace Capture Status of Malwares Studied

Chapter 6

Behavior Modeling

One difficulty faced by our behavioral approach is how to extract meaningful behaviors from system call traces. A single software behavior can be called using different combinations of API system calls, so one behavioral function could be represented by many different API sequences.

The problem is that if we only look at a limited set of sequences to represent a behavioral function, we would not be able to recognize all the different sequences of equivalent behavior. And as the collection of sequences to represent one behavior may be very large, we require a better method of behavior detection than looking for frequent patterns within the system call traces.

In this chapter, we will attempt to show how to model behaviors from sequences of system calls.

6.1 Recap of Anomaly-based Systems using System Calls

From what we had seen in Section 3.1, a large number of intrusion detection system research use system calls as proxy for the host's behavior. Typically, fixed or variable sliding windows of system call event sequences are used as the basic unit, and are assigned values representing normalcy or abnormality, using various data mining techniques.

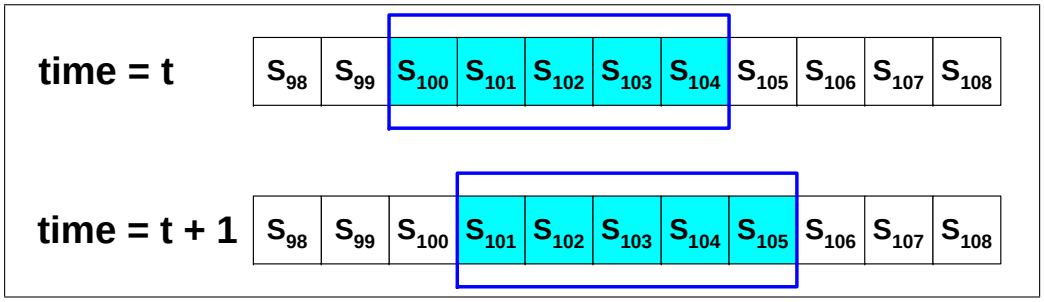


Figure 6.1: API System Call Event Sequence with Sliding Window of 5

These values are then used to compute numerical results, whereby a value over a predefined threshold represents the probability of normal behavior or intrusion.

6.2 Behavioral Blocks

From the analysis of the raw system call traces, we observed that some individual API system calls could be used to infer certain behaviors, while other behaviors require an entire sequence. For example, the checking of date or time can be accomplished with a single NtQuerySystemTime call. In this thesis, we will refer to the sequence of system calls that infer behaviors as a block.

Because of the multi-tasking and multi-threading nature of the Windows operating system, the system call events generated by different processes are interleaved together. Application or system programs spawn processes and different tasks of the process are assigned to different threads within the process. As threads do the actual work, the sequence of system call events in each thread are in sequential order. By looking at the system call traces at the thread level, it allows us a more manageable way to look for behaviors.

$$
\begin{aligned}
S_{all} &= \{A_{1,1}[1], A_{1,2}[2], A_{1,3}[3], A_{2,1}[4], A_{3,1}[5], A_{2,2}[6], ..., A_{P,T}[D]\} \\
S_{1,1} &= \{A_{1,1}[1], A_{1,1}[2], A_{1,1}[3], A_{1,1}[4], A_{1,1}[5], A_{1,1}[6], ...\} \\
&\vdots \\
S_{3,1} &= \{A_{3,1}[1], A_{3,1}[2], A_{3,1}[3], A_{3,1}[4], A_{3,1}[5], A_{3,1}[6], ...\}
\end{aligned}
$$

where

S - sequence of API system call events,

$A_{P,T}[D]$ - system call event of process P and thread T, at delta time D.

6.2.1 Delimiters

We will use delimiters provided by the format structure of Windows' Native API to construct a block, instead of using a sliding window.

The delimiter to end a block is always the `NtClose` system call, while the delimiter to begin the block depends on the object being manipulated. For example, File blocks begin with the system call `NtCreateFile` or `NtOpenFile`; Registry blocks with `NtCreateKey` or `NtOpenKey`; and so on. Please refer to Table 6.1 for more examples. The sequence of system calls within the block are linked by their Object Handles.

Object	Begin Delimiter	Object Handles
File	NtCreateFile, NtOpenFile	FileHandle
Registry	NtCreateKey, NtOpenKey	KeyHandle
Memory	NtCreateSection, NtOpenSection	SectionHandle
Mutex	NtCreateMutant, NtOpenMutant	MutantHandle
Process	NtCreateProcess, NtOpenProcess	ProcessHandle
Event	NtCreateEvent, NtOpenEvent	EventHandle
Thread	NtCreateThread, NtOpenThread	ThreadHandle

Table 6.1: Examples of Begin Delimiter System Calls

As Windows treats resources like file, memory or network points as objects, the block can be used to model behaviors manipulating different types of resources in a similar fashion.

6.2.2 Block Property

A system call event consists of: an unique identifier or counter, process ID (PID), thread ID (TID), input and/or output arguments and return status. Within a block, a system call is able to access all the input and output information of its preceding system call events, and manipulate the objects initialized by those system calls according to the access rights granted. We will show this property with an example.

In Figure 6.2, we see that five system calls are linked by the same Object Handle 752. Therefore, these five system calls form a block.

$$Block_{unknown} = \{NtCreateFile, NtSetInformationFile, NtWriteFile, NtSetInformationFile, NtClose\}$$

From Figure 6.4, the `NtCreateFile` event creates a new file "`C:\Program Files\Common Files\Microsoft Shared\Ahead Nero 7.exe`". This information can be seen from the input arguments CreateDisposition (`FILE_CREATE`) and ObjectName. This file is initialized as a file object with write permission, and is referenced by the File Handle 752.

140117 772 720	NtCreateFile (0x40110080, {24, 0, 0x40, 0, 1242692, "\??\C:\ProgramFiles\CommonFiles\MicrosoftShared\AheadNero7.exe"}, 0x0, 128, 0, 2, 100, 0, 0, ... 752, {status=0x0, info=2},) == 0x0
140118 772 720	NtSetInformationFile (752 , 1242728, 8, EndOfFile, ... {status=0x0, info=0},) == 0x0
140119 772 720	NtCreateSection (0xf001f, 0x0, 0x0, 2, 134217728, 768, ... 756,) == 0x0
140120 772 720	NtMapViewOfSection (756, -1, (0x0), 0, 0, {0, 0}, 0, 1, 0, 2, ... (0x2040000), 0, 0, 24576,) == 0x0
140121 772 720	NtClose (756, ...) == 0x0
140122 772 720	NtWriteFile (752 , 0, 0, 0, "MZ\0\0\1⟨DELETED⟩", 21358, 0x0, 0, ... {status=0x0, info=21358},) == 0x0
140123 772 720	NtUnmapViewOfSection (-1, 0x2040000, ...) == 0x0
140124 772 720	NtSetInformationFile (752 , 1243632, 40, Basic, ... {status=0x0, info=0},) == 0x0
140125 772 720	NtClose (768, ...) == 0x0
140126 772 720	NtClose (752 , ...) == 0x0

Figure 6.2: Extract of Bagle.ai Sample Trace

System Call:			NtWriteFile
Counter:			140122
Process ID:			772
Thread ID:			720
Input Arguments:	FileHandle:		752
	Event:		0
	ApcRoutine:		0
	ApcContext:		0
	Buffer:		"MZ\0\0\1⟨DELETED⟩"
	Length:		21358
	ByteOffset		0x0
	Key:		0
Output Arguments:	IoStatus	status:	0x0
	Block:	info:	21358
ReturnStatus:			0x0 SUCCESS

Figure 6.3: NtWriteFile System Call Event from Bagle.ai Sample Trace

System Call:			NtCreateFile
Counter:			140117
Process ID:			772
Thread ID:			720
Input Arguments:	DesiredAccess:		0x40110080 FILE_READ_ATTRIBUTES, DELETE,SYNCHRONIZE, GENERIC_WRITE
	Object Attributes:	Length:	24
		RootDirectory:	0
		Attributes:	0x40 OBJ_CASE_INSENSITIVE
		Security Descriptor:	0
		Security Quality Of Service:	1242692
		ObjectName:	"\??\C:\ProgramFiles\ CommonFiles\ MicrosoftShared\ AheadNero7.exe"
	AllocationSize:		0x0
	FileAttributes:		128 FILE_ATTRIBUTE_NORMAL
	ShareAccess:		0 NONE
	CreateDisposition:		2 FILE_CREATE
	CreateOptions:		100 FILE_SEQUENTIAL_ONLY,FILE_ SYNCHRONOUS_IO_NONALERT, FILE_NON_DIRECTORY_FILE
	EaBuffer:		0
	EaLength:		0
Output Arguments:	FileHandle:		752
	IoStatus Block:	status:	0x0
		info:	2 FILE_CREATE
ReturnStatus:			0x0 SUCCESS

Figure 6.4: NtCreateFile System Call Event from Bagle.ai Sample Trace

When we look into the details of the `NtWriteFile` system call event from Figure 6.3, we see that it successfully wrote data into the file referenced by File Handle 752. It was able to access the all the information of the file object created by the preceding `NtCreateFile` event.

6.3 Identification of Block Behavior

Without any advanced knowledge, it might seem that we would need a sophisticated learning algorithm to learn the behavior of a block; but by understanding the workings of Windows [42] and the native system call API, it makes behavior identification easier. In most cases, the blocks themselves provide enough information to identify the functions they serve. By studying the system calls used [31, 33], and the source code from the Windows Driver Development Kit [28], we were able to identify a large number of behavioral blocks. We will demonstrate the process with an example of how we identify a `file_write` behavior from the trace in Figure 6.5.

2098 816 764	NtCreateFile (0xc0100080, {24, 0, 0x42, 0, 1240460, "\??\C:\WINNT\System32\netdll.dll"}, 0x0, 32, 3, 1, 96, 0, 0, ... 92, {status=0x0, info=1},) == 0x0
2099 816 764	NtQueryVolumeInformationFile (92, 1240556, 8, Device, ... {status=0x0, info=8},) == 0x0
2100 816 764	NtWriteFile (92, 0, 0, 0, "MZP\0\2\0\0\0\4\0\17\0\377\377\0\0\270\0\4\0\0\0\0@\0\32\0", 28, 0x0, 0, ... {status=0x0, info=28},) == 0x0
2101 816 764	NtClose (92, ...
2101 816 764	NtClose ...) == 0x0

Figure 6.5: Extract of Lovelorn.a Sample Trace

System Call:			NtWriteFile
Counter:			2100
Process ID:			816
Thread ID:			764
Input Arguments:	FileHandle:		92
	Event:		0
	ApcRoutine:		0
	ApcContext:		0
	Buffer:		"MZP\0\2\0⟨DELETED⟩"
	Length:		28
	ByteOffset		0x0
	Key:		0
Output Arguments:	IoStatus	status:	0x0
	Block:	info:	28
ReturnStatus:			0x0 SUCCESS

Figure 6.6: NtWriteFile System Call Event from Lovelorn.a Sample Trace

System Call:			NtQueryVolumeInformationFile
Counter:			2099
Process ID:			816
Thread ID:			764
Input Arguments:	FileHandle:		92
	FileSystemInformation:		1240556
	Length:		8
	FileSystemInformationClass:		Device
Output Arguments:	IoStatus	status:	0x0
	Block:	info:	8
ReturnStatus:			0x0 SUCCESS

Figure 6.7: NtQueryVolumeInformationFile System Call Event from Lovelorn.a Sample Trace

System Call:			NtCreateFile
Counter:			2098
Process ID:			816
Thread ID:			764
Input Arguments:	DesiredAccess:		0xc0100080 FILE_READ_ATTRIBUTES, SYNCHRONIZE,GENERIC_ WRITE,GENERIC_READ
	Object Attributes:	Length:	24
		RootDirectory:	0
		Attributes:	0x42 OBJ_INHERIT,OBJ_CASE_ INSENSITIVE
		Security Descriptor:	0
		Security Quality Of Service:	1240460
		ObjectName:	"\??\C:\WINNT\System32\ netdll.dll"
	AllocationSize:		0x0
	FileAttributes:		32 FILE_ATTRIBUTE_ARCHIVE
	ShareAccess:		3 FILE_SHARE_READ, FILE_SHARE_WRITE
	CreateDisposition:		1 FILE_OPEN
	CreateOptions:		96 FILE_SYNCHRONOUS_IO_ NONALERT,FILE_NON_ DIRECTORY_FILE
	EaBuffer:		0
	EaLength:		0
Output Arguments:	FileHandle:		92
	IoStatus Block:	status:	0x0
		info:	1 FILE_OPEN
ReturnStatus:			0x0 SUCCESS

Figure 6.8: NtCreateFile System Call Event from Lovelorn.a Sample Trace

6.3.1 Detection

For certain behaviors, we noticed that we do not need to study every system call that appears within a block. Depending on the behavior required, some system call types can be ignored.

The trace and system call information presented previously in Section 6.3 shows the behavior of writing to a file.

$$Block_{file\_write} = \{NtCreateFile, NtQueryVolumeInformationFile, NtWriteFile, NtClose\}$$

We know that if we see an `NtWriteFile` event in the block, a write operation was performed on the file ObjectName. Since the `NtCreateFile` event input argument CreateDisposition is `FILE_OPEN`, we know that an existing file was written to, instead of creating a new file. The `NtQueryVolumeInformationFile` event provides no useful information as we are convinced that the block performs a `file_write`.

Other examples in File operations include an `NtSetInformationFile` event with input argument FileInformationClass of `Disposition` indicating a `file_delete` behavior.

We are trying to show that there is no need to study all the system call events in a block. If we are looking for specific behaviors, certain system call types can give us enough information.

6.3.2 Identification

Compared to simple detection, the criteria for identification are more stringent. This is because the identified block behavior is used for classifying

malwares. While the behaviors detected may be the same for some malwares, how these behaviors were accomplished might be entirely different.

Identification is more complex because we have to look at all the system calls and their parameters. Not only are the parameters of different system calls are used differently, some parameters are more important than others. We will demonstrate how we chose the parameters.

Let us take a look at the `NtQueryVolumeInformationFile` event from Figure 6.7, it has three input arguments excluding the FileHandle: FileSystemInformation, Length and FileSystemInformationClass. The two output arguments, status and info, do not matter because they reflect the values from Length and ReturnStatus respectively. The input argument FileSystemInformationClass is `Device`, which tells us that it tried to get information about the volume device containing "netdll.dll". FileSystemInformation is a buffer, and Length is the size of the buffer, so these two arguments do not provide us with much information and can be discounted.

From Figure 6.6, the `NtWriteFile` event has seven input arguments excluding the FileHandle: Event, ApcRoutine, ApcContext, Buffer, Length, ByteOffset and Key. The two output arguments, status and info, do not matter because they also reflect the values from Length and ReturnStatus respectively. Of all the input arguments, only Buffer and Length are not optional arguments that can be discounted. The Buffer argument shows part of the binary data to be written, and Length is the size of Buffer. These two arguments do not provide us with more information and can be discounted too.

Finally, let us take a look at Figure 6.8 showing the `NtCreateFile` event. It

has nine input arguments: DesiredAccess, ObjectAttributes, AllocationSize FileAttributes, ShareAccess, CreateDisposition, CreateOptions, EaBuffer, EaLength. The two output arguments, status and info, do not matter because they reflect the values from CreateDisposition and the ReturnStatus respectively. AllocationSize and EaBuffer are optional arguments and can be discounted. As EaLength is the size of EaBuffer, we can disregard this argument too. ObjectAttributes is a buffer, containing the following arguments: Length, RootDirectory, Attributes, SecurityDescriptor, SecurityQualityOfService and ObjectName. We only need RootDirectory and ObjectName to know what Object we are accessing, but these are not required to determine behaviors.

Of the remaining five input arguments, we can disregard DesiredAccess and ShareAccess, which provides the access rights to the file because `NtWriteFile` must be successful for this block to be true. We also disregard FileAttributes because knowing the changed file attribute information does not contribute more data in this case. We are interested in CreateDisposition, which tells us the file we are accessing is an existing file, and CreateOptions that tells us the file object is not a directory.

As this is the ninth `file_write` block we had encountered, we will name it `file_write9`. The system call events and arguments needed to represent `file_write9` are shown in Figure 6.9.

As the sequence of system call events gets longer and more varied, the more complex the behaviors become. As this is a new area of research, the choosing of parameters to identify a block is done by a human expert. At this time, we have not found any algorithmic method to accomplish this as we are still in the infancy of our work.

NtCreateFile	
FileAttributes:	FILE_ATTRIBUTE_ARCHIVE
CreateDisposition:	FILE_OPEN
CreateOptions:	FILE_SYNCHRONOUS_IO_NONALERT FILE_NON_DIRECTORY_FILE
ReturnStatus:	SUCCESS

NtQueryVolumeInformationFile	
FileSystem InformationClass:	Device
ReturnStatus:	SUCCESS

NtWriteFile	
ReturnStatus:	SUCCESS

Figure 6.9: System Call Events and Arguments Representing file_write9

We would like to reiterate the importance of choosing the right combinations of parameters. If we use too few parameters, we would likely lose some context about the behavior. For example in this section, we chose not to use the FileAttributes argument in the `file_write9` block. This means we lose the information that the file access rights was changed, but as we are identifying write behavior, this information is not important.

On the other hand, if we use too many parameters, we lose the ability to generalize. If we encounter another similar block with minor difference in parameters like the desired access rights, we would have to create another identification signature with minor differences.

6.4 Matching Blocks with Finite State Automata

6.4.1 Block FSA

We use finite state automata to model block information because they enable efficient behavior function identification. Using FSAs to perform pattern matching on blocks allows us flexibility in the software implementation as FSA can be implemented using regular expressions, which is supported in many programming languages.

In our implementation, the three different transitions are the System Call name (SYSCALL), the arguments (ARG), and the return status (RET) of the event. We will use the `file_write9` block from the previous section for illustration. The information from the block of system call events in the previous Figure 6.9 is represented in Figure 6.10.

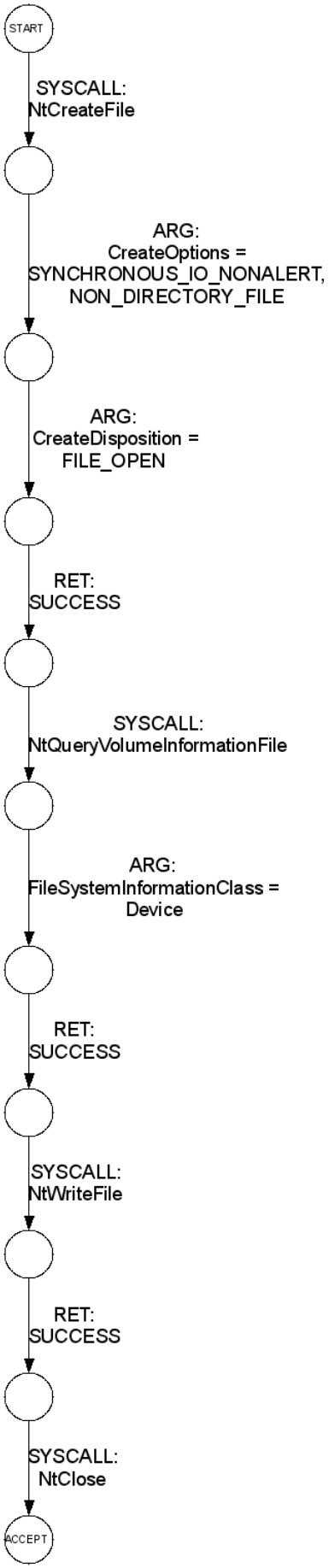


Figure 6.10: file_write9 Block FSA

6.4.2 Generalized Block FSA

In longer block patterns, we notice that there are a lot of repetitions within the blocks. By finding frequent patterns within the blocks, we can generalize one FSA to model similar blocks.

We will look at the simplified representation of $Block_{file\_write9}$ below. This block can be modeled with the FSA shown in Figure 6.10. The difference between $Block_{file\_write9}$ and $Block_{file\_write89}$ is the repetition of the `NtWriteFile` event.

$$\begin{aligned} Block_{file\_write9} &= \{NtCreateFile, NtQueryVolumeInformationFile, \\ &\quad NtWriteFile, NtClose\} \end{aligned}$$

$$\begin{aligned} Block_{file\_write89} &= \{NtCreateFile, NtQueryVolumeInformationFile, \\ &\quad NtWriteFile, NtWriteFile, NtWriteFile, NtWriteFile, \\ &\quad NtWriteFile, NtWriteFile, NtWriteFile, NtWriteFile, \\ &\quad NtWriteFile, NtWriteFile, NtWriteFile, NtWriteFile, \\ &\quad NtWriteFile, NtClose\} \end{aligned}$$

By slightly modifying the FSA to include an additional transition from the return status state of `NtWriteFile` back to itself, as shown in Figure 6.11, this FSA can be used to model both $Block_{file\_write9}$ and $Block_{file\_write89}$.

A more complicated FSA is shown in Figure 6.12. At this time, the process of sequence reduction using frequent patterns is done manually. We hope to study various algorithms for this process in future works.

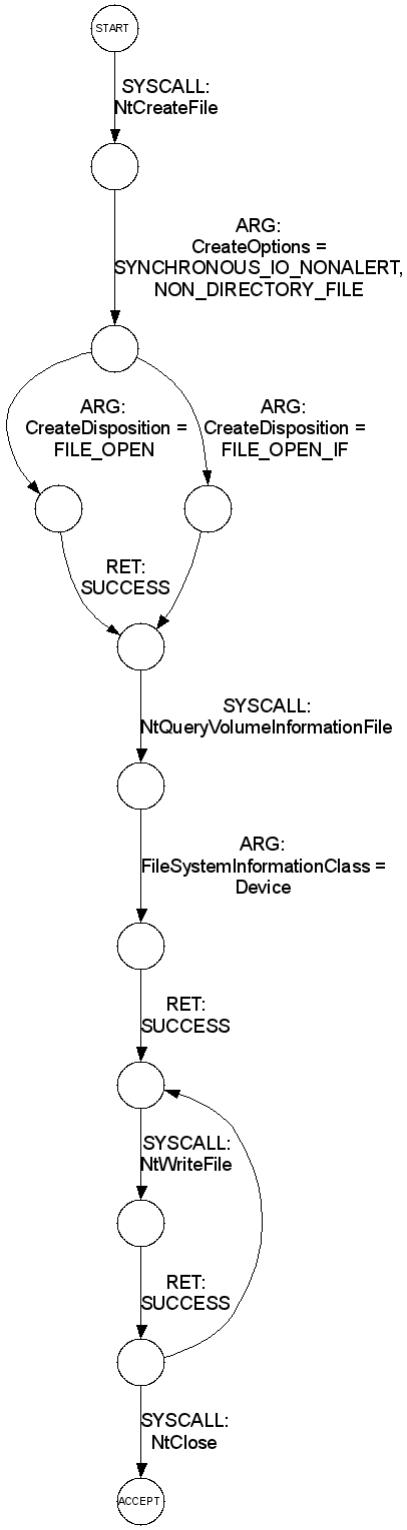


Figure 6.11: Generalized file_write9 Block FSA

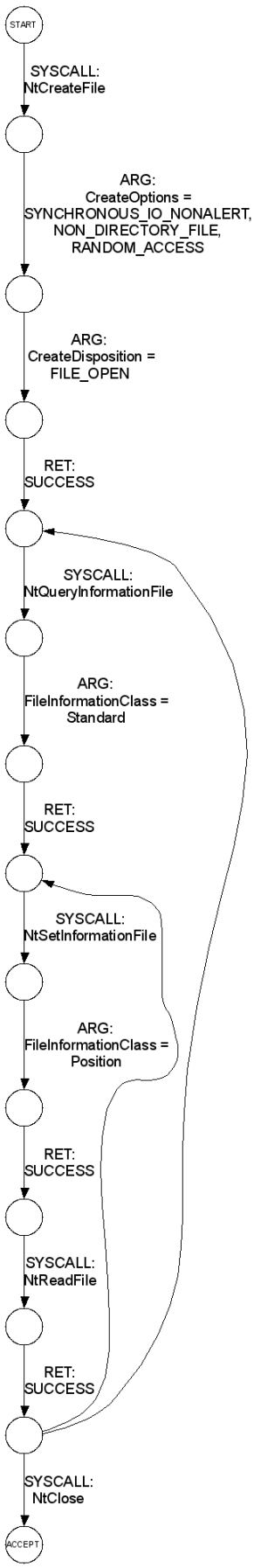


Figure 6.12: Generalized file_read5 Block FSA

6.5 Behavioral Macros

The basic unit to represent behaviors is the individual system call events. A structured sequence of system call events is a block. When blocks are combined to form more complex behaviors, we call them macros.

We will introduce three main types of block relationship within a macro: interleave, intersect and super.

6.5.1 Interleaving Blocks

Interleave is the simplest relationship. The two blocks or macros share no direct relationship or information, other than the fact that the sequences of both are sequentially interleaving.

We refer to Figure 6.13, which is a representation of the traces from Figure 6.14. The macro `file_copy` is formed from the `file_write` block and `file_load` macro. We can see from Figure 6.14 that they share no common information.

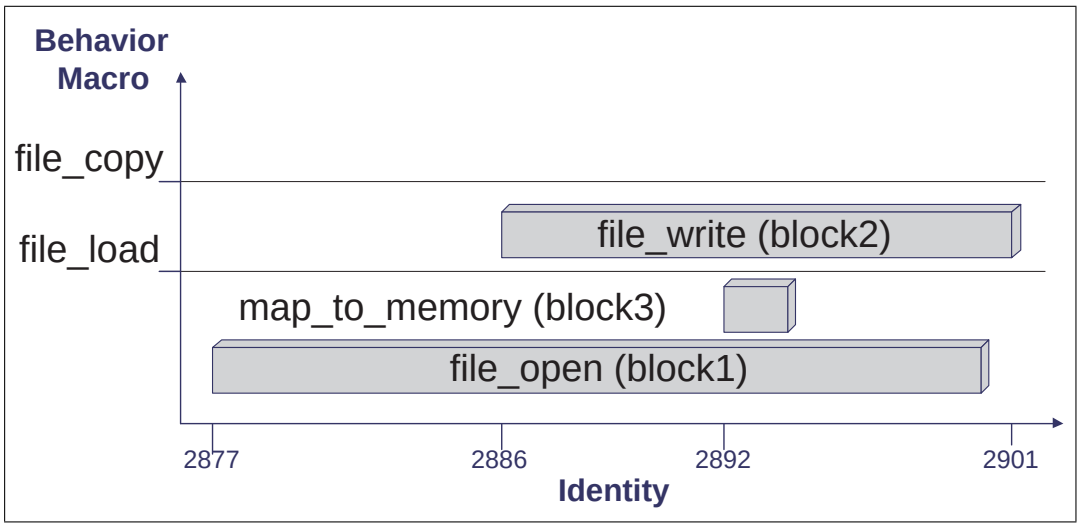


Figure 6.13: Bagle.at File Copy Macro Behavior

Block	Trace	
1	2877 764 740	NtCreateFile (0x80100080, {24, 0, 0x40, 0, 1242960, "\??\C:\Email-Worm.Win32.Bagle.at.exe"}, 0x0, 0, 1, 1, 2097252, 0, 0, ... 448, status=0x0, info=1,) == 0x0
	⋮	
1	2885 764 740	NtQueryInformationFile (448, 1242864, 4, Ea, ... {status=0x0, info=4},) == 0x0
2	2886 764 740	NtCreateFile (0x40110080, {24, 0, 0x40, 0, 1242852, "\??\C:\WINNT\System32\wingo.exe"}, 0x0, 33, 0, 5, 100, 0, 0, ... 444, {status=0x0, info=2},) == 0x0
2	2889 764 740	NtSetInformationFile (444, 1242888, 8, EndOfFile, ... {status=0x0, info=0},) == 0x0
1,3	2892 764 740	NtCreateSection (0xf001f, 0x0, 0x0, 2, 134217728, 448, ... 452,) == 0x0
3	2893 764 740	NtMapViewOfSection (452, -1, (0x0), 0, 0, {0, 0}, 0, 1, 0, 2, ... (0xcf0000), {0, 0}, 20480,) == 0x0
3	2894 764 740	NtClose (452, ...) == 0x0
2	2895 764 740	NtWriteFile (444, 0, 0, 0, "BINARY_DATA", 19069, 0x0, 0, ... {status=0x0, info=19069},) == 0x0
	2896 764 740	NtUnmapViewOfSection (-1, 0xcf0000, ...) == 0x0
2	2897 764 740	NtSetInformationFile (444, 1243792, 40, Basic, ... {status=0x0, info=0},) == 0x0
1	2900 764 740	NtClose (448, ...) == 0x0
2	2901 764 740	NtClose (444, ...) == 0x0

Figure 6.14: Extract of Email-Worm.Win32.Bagle.at Sample Trace

6.5.2 Intersecting Blocks

Intersecting blocks has the closest relationship. The two blocks or macros share common system call events, in addition to the sequences of both sequentially interleaving.

From Figure 6.13, the macro `file_load` is formed from the `file_open` and `map_to_memory` blocks. We can see from Figure 6.14 that they share the `NtCreateSection` system call event.

6.5.3 Super Blocks

Super relationship is when one block or macro inherits objects or parameter information from another, in addition to the sequences of both sequentially

interleaving.

The traces from Figure 6.15 shows two blocks, where $Block_1$ is the super block of $Block_2$. In $Block_1$, the `NtOpenKey` event initialized a registry object "\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000", which is a registry path, referenced by the KeyHandle 76. In $Block_2$, the `NtCreateKey` event inherits the object via the handle 76. The path of the previous object is combined with current ObjectName "SOFTWARE\Microsoft\Windows\CurrentVersion\Run" to initialize the current object "\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000SOFTWARE\Microsoft\Windows\CurrentVersion\Run".

Block	Trace	
1	4603 772 720	NtOpenKey (0x2000000, 24, 0, 0x40, 0, 0, "\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000", ... 76 ,) == 0x0
	⋮	
2	6052 772 720	NtCreateKey (0x2000000, 24, 76 , 0x40, 0, 0, "SOFTWARE\Microsoft\Windows\CurrentVersion\Run", 0, 0x0, 0, ... 448, 2,) == 0x0
2	6053 772 720	NtSetValueKey (448, "key", 0, 1, "C\0:\0\\0W\0I\0N\0N\0T\0\\0S\0y\0s\0t\0e\0m\03\02\0\\0w\0i\0n\0x\0p\0.\0e\0x\0e\0\0\0", 56, ...) == 0x0
2	6054 772 720	NtClose (448, ...) == 0x0
	⋮	
1	328060 772 720	NtClose (76, ...) == 0x0

Figure 6.15: Extract of Sample Trace from Bagle.ai

6.6 Mapping of Behaviors to Blocks

We have seen in Section 6.3.2 how we can identify simple blocks based on the system call information. But identifying complex behaviors will require more effort.

While it is very hard to find behaviors from a system trace, it is much easier to extract sequences from system trace if the behavior is known (for example, if we know in advance a malware creates an identifier in the memory to indicate it's presence, we can analyze the system calls related to that identifier to form a behavioral function).

Therefore, our approach is to find obscure blocks of system calls that indicate the behaviors of the malwares based on the technical descriptions from anti-virus companies we discussed in Section 4.3 and footprints left behind from the actions taken by the malwares.

This approach is inspired by criminal profiling and medical differential diagnosis. An medical analogy we can use is that if a doctor suspects that his patient have a very tiny tumor that cannot be seen by a MRI scan, he can check the patient's blood for certain antibodies that might indicate the presence of a tumor.

Any actions taken by a process in a host leaves traces. In some cases, preparations taken to undertake an action leave traces. For example, using native API alone cannot accurately identify behaviors like network connections, we would need to monitor the Winsock API to get details. But by looking for blocks involving network device objects or network DLLs, we can get hints about network activities.

We will demonstrate this with the process of how we identified a code injection macro. Code injection is a technique whereby we inject executable code into an existing process. While it might seem like a bad idea for non-malicious programs, it is used quite frequently.

Block	Trace	
1	9114 220 824	NtOpenFile (0x100020, {24, 0, 0x40, 0, 0, "\??\C:\WINNT\system32\1.dll"}, 5, 96, ... 336, {status=0x0, info=1},) == 0x0
1,2	9115 220 824	NtCreateSection (0xf, 0x0, 0x0, 16, 16777216, 336, ...
2	9115 220 824	NtCreateSection ... 560,) == 0x0
1	9121 220 824	NtClose (336, ...) == 0x0
2	9122 220 824	NtMapViewOfSection (560, -1, (0x0), 0, 0, 0x0, 0, 1, 0, 4, ... (0x1110000), 0x0, 86016,) == STATUS_IMAGE_NOT_AT_BASE
I1	9123 220 824	NtProtectVirtualMemory (-1, (0x1111000), 45056, 4, ... (0x1111000), 45056, 32,) == 0x0
I2	9124 220 824	NtProtectVirtualMemory (-1, (0x111c000), 12288, 4, ... (0x111c000), 12288, 2,) == 0x0
I3	9125 220 824	NtProtectVirtualMemory (-1, (0x1123000), 8192, 4, ... (0x1123000), 8192, 2,) == 0x0
2	9126 220 824	NtMapViewOfSection (560, -1, (0x1110000), 0, 0, 0x0, 86016, 1, 0, 4, ...) == STATUS_CONFLICTING_ADDRESSES
I4	9127 220 824	NtProtectVirtualMemory (-1, (0x1111000), 45056, 16, ... (0x1111000), 45056, 4,) == 0x0
I5	9128 220 824	NtProtectVirtualMemory (-1, (0x111c000), 12288, 2, ... (0x111c000), 12288, 4,) == 0x0
I6	9129 220 824	NtProtectVirtualMemory (-1, (0x1123000), 8192, 2, ... (0x1123000), 8192, 8,) == 0x0
I7	9130 220 824	NtFlushInstructionCache (-1, 0, 0, ...) == 0x0
2	9131 220 824	NtClose (560, ...) == 0x0

Figure 6.16: code_injection Extract of Sample LovGate.a Trace

From the technical description of Email-Worm.Win32.LovGate.a, we learn that the malware does a code injection using the dynamic link library file "1.dll". By searching for the file in the trace, we found the macro as shown in Figure 6.16. Block 2, interleaved with seven individual system call events, shows sign of a code injection. The return status from the two `NtMapViewOfSection` events, `STATUS_IMAGE_NOT_AT_BASE` and `STATUS_CONFLICTING_ADDRESSES`, are the main indicators. We double-checked this with Email-Worm.Win32.LovGate.b, and security applications like Spybot - Search & Destroy and ClamWin Anti Virus, which are also known to use code injections.

6.7 Correlation of Behavior Blocks or Macros

We can gather more behavioral information when we look at the correlations of blocks or macros. We will demonstrate with one example whereby we get the behavior for searching all the directories on the local drive: `search_all_dir_recursive`.

The block behavior,

`dir_search2 "*" "\??\C:\"`

represents the searching of all files in the `"C:"` root directory. From Table 6.5, we can see that the search starts from `"C:"`, and continues depth-first from `"C:\WINNT"` all the way to `"C:\Recycled"`, the last directory on the drive. Thus, these behavioral function blocks are used to model `search_all_dir_recursive`.

Block Start	Block End	Behavior Functions
5636	5651	`dir_search2 "*" "\??\C:\"`
5680	5712	`dir_search2 "*" "\??\C:\WINNT\"`
5723	6233	`dir_search2 "*" "\??\C:\WINNT\system32\"`
6242	6252	`dir_search2 "*" "\??\C:\WINNT\system32\config\"`
6452	6489	`dir_search2 "*" "\??\C:\WINNT\system32\drivers\"`
6498	6502	`dir_search2 "*" "\??\C:\WINNT\system32\drivers\etc\"`
		⋮
376585	376593	`dir_search2 "*" "\??\C:\ProgramFiles\WinZip\"`
378849	378855	`dir_search2 "*" "\??\C:\Recycled\"`

Table 6.5: dir_search2 Blocks from Sober.f Sample Trace

Chapter 7

Malware Behavioral Analysis

In this chapter, we first demonstrate the detection capability of our system using the most prevalent behaviors as discussed in Section 4.7. We also introduce the generalization of behavioral functions by showing reuse of basic behavioral blocks among malwares. In the rest of the chapter, we will explore various issues important to our behavior-based approach.

7.1 Accuracy of Technical Descriptions from Anti-virus Companies

One interesting piece of information that may interest many researchers is the accuracy of the malware descriptions provided by anti-virus companies. We will study the accuracy of the descriptions by looking for a small set of described behaviors in the captured behavioral traces of a sample number of malwares.

The more prevalent behaviors are the sample choices, and they are also the behaviors that form the `survive_system_reboot` and `find_email_addresses` complex behaviors. The sample malware choices are limited to those we managed to capture successfully, as discussed in Section 5.6.2.

They are Bagle.ai, Bagle.at, Ganda, LovGate.a, LovGate.b, Lovelorn.a, Mimail.a, Sober.a, Sober.f, Sober.g and Welchia.a .

7.1.1 Recap of Behavioral Functions Used

The behavioral functions that we will use in our analysis were discussed in the previous chapters, but we will provide a brief summary to help the reader recall the details.

The complex functions of `survive_system_reboot` and `find_email_addresses` are formed by combinations of the following behavioral functions with parameters offering risk differentiation:

- `file_copy` others
- `file_copy` System
- `file_copy` Windows
- `file_create` others
- `file_create` System
- `file_create` Windows
- `registry_modify` shell
- `registry_add` startup
- `registry_add` service
- `find_data_files`
- `search_all_dir_recursive`
- `search_specific_dir_recursive`
- `harvest_emails`

We will show the possible complex behaviors combinations using Boolean notations.

For `survive_system_reboot` function:

- `registry_add` startup AND (`file_copy` OR `file_create`)
- `registry_add` service AND (`file_copy` OR `file_create`)
- `registry_modify` shell AND (`file_copy` OR `file_create`)

For `find_email_addresses` function:

- `search_all_dir_recursive` AND `find_data_files` AND `harvest_emails`
- `search_specific_dir_recursive` AND `find_data_files` AND `harvest_emails`

The function `file_create` is a basic behavior that creates a new file, while `file_copy` duplicates a file, to a certain directory. The parameters shown are the target directories, where *Windows* is the "`C:\Windows`" folder, and *System* is the "`C:\Windows\System32`" folder, of a Windows 2000 system. Any other target directories are represented by *others*.

As a behavior-based system, the filenames of the created files are not directly used, but the filenames are used to show correlation to the registry-based functions. Thus a `survive_system_reboot` function can only be confirmed if the file-based and registry-based functions operate on the same files.

The function `registry_add` adds data like file names and locations to the registry, where the parameters are the locations of the target registry keys. The parameter *startup* is the location of the key that stores information about the programs that should run during a Windows start up cycle, and *service* stores information about the programs that should run as services during boot time. Adding file information to these keys will result in new programs activating when Windows starts up. The function `registry_modify` modifies existing registry data. The parameter *shell* stores information about which programs to run when files of certain extensions are activated. For example, the default "*txt*" shell of a Windows 2000 system points to the *notepad* application. Changing the *txtshell* key data will result in another program being called when text files are activated.

The function `search_all_dir_recursive` traverse through the entire filesystem starting from the root directory "`C:\`", whereas `search_specific_dir_recursive` starts from a specific directory.

The `find_data_files` function is used to represent the search of text, html or other application data files. In the Windows environment, the different file types are usually indicated by their extensions. This function must occur within the lifetime of a `search_all_dir_recursive` or `search_specific_dir_recursive` function to be used to form the complex `find_email_addresses` function.

The function `harvest_emails` indicates the parsing of files for patterns matching email addresses. This behavior can be achieve by many different methods. For example, the Email-Worm.Win32.Sobig.a worm searches for patterns within files using the regular expression:
`[A-Za-z0-9]+[A-Za-z0-9_.-]+@(([A-Za-z0-9\-])+[.])+[A-Za-z]+`

As there are many possible ways for the `harvest_emails` function to occur, we will exclude this function from the analysis presented in this section.

7.1.2 Discussion of Description Accuracy

By looking for the behaviors that form the `survive_system_reboot` and `find_email_addresses` complex behaviors, we formed a behavioral functions versus malwares matrix in Appendix D.1. From the matrix, we derived Figure 7.1 which shows the percentage of correct versus wrong identification of behavioral functions among the malwares.

Based on the description from the anti-virus companies, we expected to detect the twelve basic behaviors that form `survive_system_reboot` and

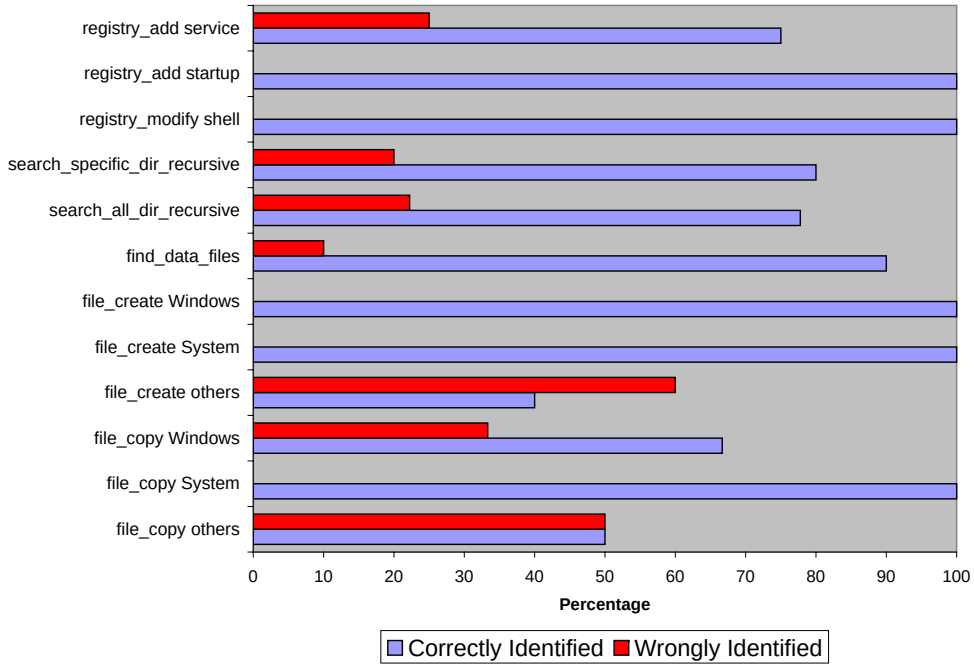


Figure 7.1: Percentage of Correctly Detected Malware Behaviors

`find_email_addresses` 63 times from the traces of the eleven sample malwares. The number of correctly detected behaviors is 53, and 10 expected behaviors were not detected at all. We did not detect any additional behaviors. At this point, the accuracy is only about **84%**.

As the numbers of malwares and behaviors used are statistically insignificant, we will not draw any conclusions.

7.2 Detection Capability

We would like to see how effective is the system in detecting the presence of malwares. First, we look for malware behaviors in normal applications. Using the functions in Section 7.1, we formed a behavioral functions versus normal applications matrix in Appendix D.2. A summary of this matrix can be seem in Figure 7.2, which shows that most normal applications do not display behaviors similar to malwares. For example, only the ICQ application attempts to restart itself at the next reboot by modifying the

registry (`registry_add startup`), out of all these sample applications.

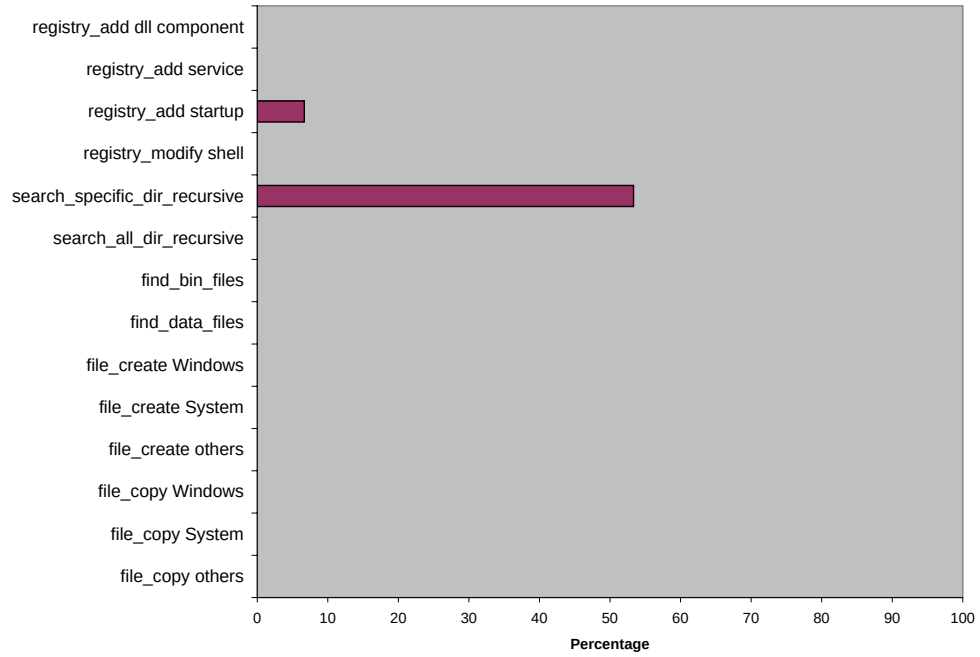


Figure 7.2: Percentage of Detected Malware Behaviors in Normal Application

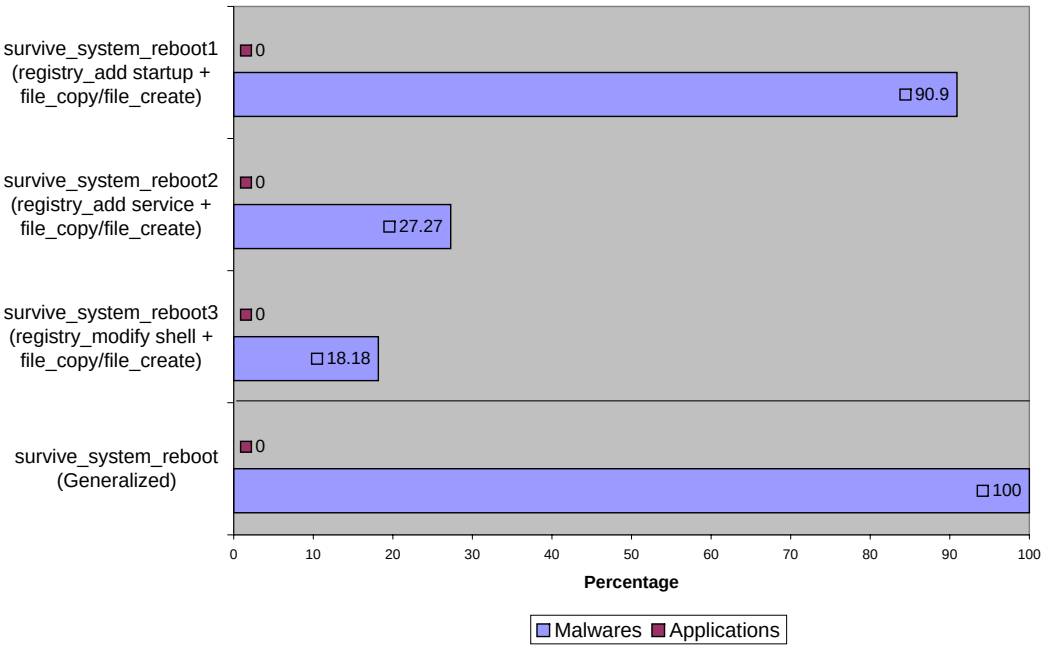


Figure 7.3: Percentage of Detected Correlated survive_system_reboot Behaviors

Looking at Figure 7.3, we see that **100%** of the malwares perform behaviors that allow themselves to be started at the next reboot while none of the

applications do that. This shows that the `survive_system_reboot` function is indicative of malware presence. The details of distribution between malwares and behaviors can be found in Appendix D.3.

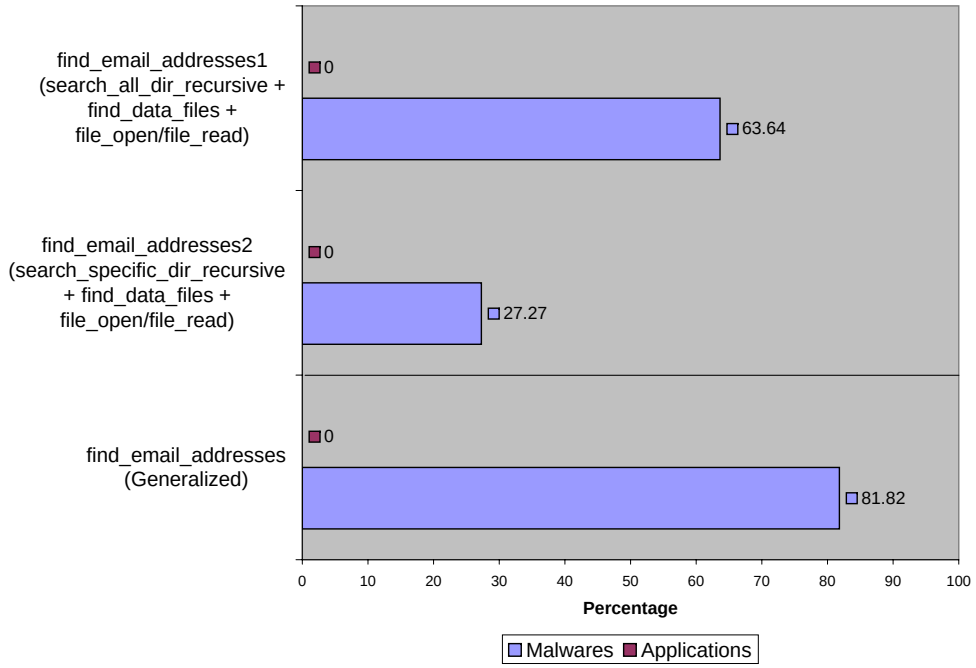


Figure 7.4: Percentage of Detected Correlated find_email_addresses Behaviors

In Figure 7.4, we weakened the requirement for detecting the behavior of harvesting email addresses from data files found on the host. Instead of requiring the detection of the parsing process, we will accept the behavior as detected if data files are opened or read while the process is searching for files. This is represented by the `file_open` and `file_read` functions. **0%** of the applications display this behavior, while 9 out of 11 malwares do. The details can be found in Appendix D.4. The two malwares that do not display this behavior are Mimail.a and Welchia.a. As Welchia.a is a network worm, we did not expect it to exhibit any email propagation activities anyway, but we did expect Mimail.a to display this behavior. From further investigation of the traces, we find that Mimail.a only harvest the email addresses from the Windows Address Book of the current user. This

behavior will be added to refine our detection of `find_email_addresses` behavior.

Therefore, we can use `survive_system_reboot` and `find_email_addresses` as the main behavioral functions indicating the presence of malwares.

7.3 Generalization of Behaviors

To be able to stop newer malwares based on the behaviors of older malwares, we must be able to generalize the detected behaviors. The generalization depends in the reuse of basic blocks. If our assumption is wrong and malware behaviors cannot be generalized, the basic behavior blocks used should be unique to each malware. But if the basic blocks are shared among malwares, then it provides confidence in our approach.

We will take a look at the blocks that form the `file_create` behavior. From Table 7.1, the `file_create` behavior is formed by the `file_write9` block in all the three Sober variants. This can be generalized to detect the `file_create` behavior in Lovelorn.a, which also uses `file_write9`. In another example, the two LovGate variant uses the same `file_write6` block for `file_create`.

Malware	Block
Sober.a	`file_write9 "C:\WINNT\System32\similare.exe"`
Sober.f	`file_write9 "C:\WINNT\System32\winrundiag.exe"`
Sober.g	`file_write9 "C:\WINNT\System32\crypt32sys.exe"`
Lovelorn.a	`file_write9 "C:\NQHLL.exe"`
LovGate.a	`file_write6 "C:\WINNT\System32\WinRpcsrv.exe"`
LovGate.b	`file_write6 "C:\WINNT\System32\WinRpcsrv.exe"`

Table 7.1: Blocks That Form the file_create Behavior

In addition, we can also see from Figure 7.5 that the different `file_write`

blocks are shared among the eleven malwares.

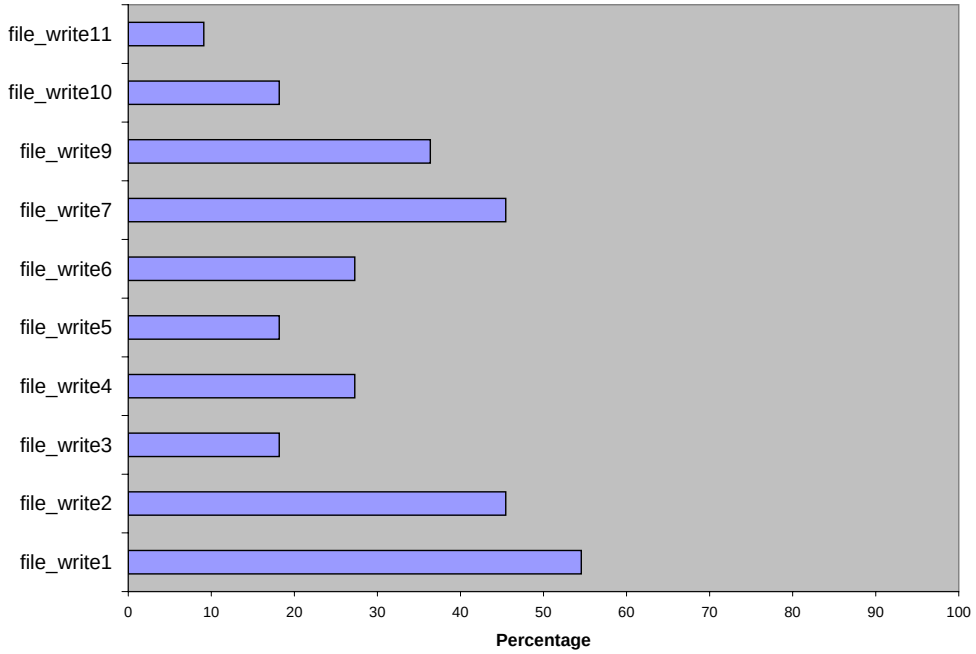


Figure 7.5: Percentage of Malwares Sharing file_write Blocks

7.4 Discussions About Behaviors

7.4.1 Importance of Behavior Functions

We noticed that some functions do not provide much information in our behavior-based system, or poses some problems.

One example is the `mutex_create` function. While unique mutex (mutual exclusion) names like `"MuXxXxTENYKSDesignedAsTheFollowerOfSkynet-D"` is a good identifier in a misuse-signature based system, the action of creating a mutex is not a good behavioral signature because most processes create mutexes with various names.

We face another problem with detecting the behavior of malwares killing anti-virus processes. From the sample traces we have, the malwares all

enumerate the processes first, instead of wildly killing non-existent programs. The decision to kill the processes is in the program logic, which is not visible from the API system call. That means we will not be able to detect the `kill_process` behavior unless the correct anti-virus or firewall programs are running in the system.

7.4.2 New Behavior: Repeated Functions

From a simple system call sequence trace, we cannot see repeating API sequences easily. But when formed into blocks, we can see a lot of repeating behavior blocks like the `registry_add` functions in the two Bagle malwares shown in Table 7.2 and 7.3.

Freq	PID TID	Function
1	744 268	registry_add key="\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\key" data="C:\WINNT\System32\winxp.exe"
8233	772 720	registry_add key="\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\key" data="C:\WINNT\System32\winxp.exe"

Table 7.2: Frequency of registry_add Functions in Bagle.ai

Freq	PID TID	Function
1	764 740	registry_add key="\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\wingo" data="C:\WINNT\System32\wingo.exe"
9287	672 760	registry_add key="\REGISTRY\USER\S-1-5-21-515967899-299502267-839522115-1000\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\wingo" data="C:\WINNT\System32\wingo.exe"

Table 7.3: Frequency of registry_add Functions in Bagle.at

In other systems studying malware behaviors, they only care about the first behavior function that succeeds. But this provides us with the opportunity to study a new behavior.

Why does Bagle.ai need to repeat that one particular behavior 8,234 times within fifteen minutes? The first time is at identity 6,053, and the rest is from 142,746 to 264,331, at almost random intervals. Other than Bagle.ai, this behavior is only seen in Bagle.at among all the malware sample traces. Our preliminary hypothesis is that it is a crude attempt to ensure that the registry key to allow the malware itself to be started at the next boot time is not changed by other malwares or anti-virus systems.

We can use this pattern of repeating functions as a new behavioral function. It will be interesting to see if we can see even more repeating behaviors after we combine correlated blocks into macros.

7.4.3 Consideration About Processes

In most older research systems using system calls as the sensor, they do not care about the inter-process and inter-thread relationships. This is because most older virus actions are contained within a single process spawned from the original infection. But it is crucial to understand the inter-process and inter-thread relationships of the newer malwares as they are no longer restricted to just one process. The malwares that we had analyzed create other processes to do different work. We think that it is to prevent detection, as a single process performing multiple virus-like behavioral functions is more likely to arouse suspicion.

To monitor inter-process communications, we first have to look at the two "correct" ways. In most cases, the original process will create a sub-process

to do its work by means of process creation (NtCreateProcess and NtCreateThread). The other way is by means of LPC (Local/Lightweight Procedure Call), which is an inter-process communication mechanism provided by Windows. These two ways of monitoring are relatively straightforward.

The problem is code injection. Detecting code injection is hard, and it is difficult to find out which process was affected. In the past, it is only necessary to monitor related processes for malware behaviors. But with code injection, we would need to correlate information from seemingly unrelated processes too.

We now face the additional problem of deciding the weightage for the relationship of behavioral functions in different processes.

7.4.4 New Local Infection Trend

One of the interesting trends that we learned from studying the malware behaviors is that out of the twelve sample malwares, only one search for and infects executable files.

One possibly explanation could be because Windows 2000 and later versions will restore system files that were changed, but do not pass the cryptographic checksum. We see this system restore behavior in Ganda, when it targets specific Windows system files like welcome.exe (Getting Started Screen) or osk.exe (On-Screen Keyboard).

We believe that the more current trend is to created files with names similar to the real system files at strategic locations, rather than infecting executable files at random.

7.5 Early Detection versus Identification Accuracy

When we extract behaviors from the malware traces, we face the problem of early detection versus accuracy. When matching behavior functions from blocks or macros, early detection is important for online detection while accuracy is important for offline identification. It is important for us to consider the reasonable trade-off. We will explore some issues that affect the consideration.

7.5.1 Blocks

For blocks, we can only accurately identify the block signature when we encounter the NtClose system call, as seen in Figure 7.6. But for detection of a generalized `file_write` function, we only need to match until state 4 because the behavior of the whole block is decided by the NtWriteFile event, the most important system call for this function.

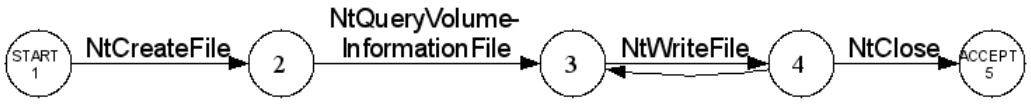


Figure 7.6: Simplified file_write9 Block FSA

7.5.2 Macros

For macros, some of the blocks may not terminate until a lot later. We will use the `search_all_dir_recursive` macro function in Bagle.at for illustration.

In Figure 7.7, the first block starts at identity 6807 and ends at 33,783: that means that we have to wait for 26,976 system calls to pass before we

can be absolutely certain that it is a recursive directory search. (Identity is the number of system calls captured since monitoring started)

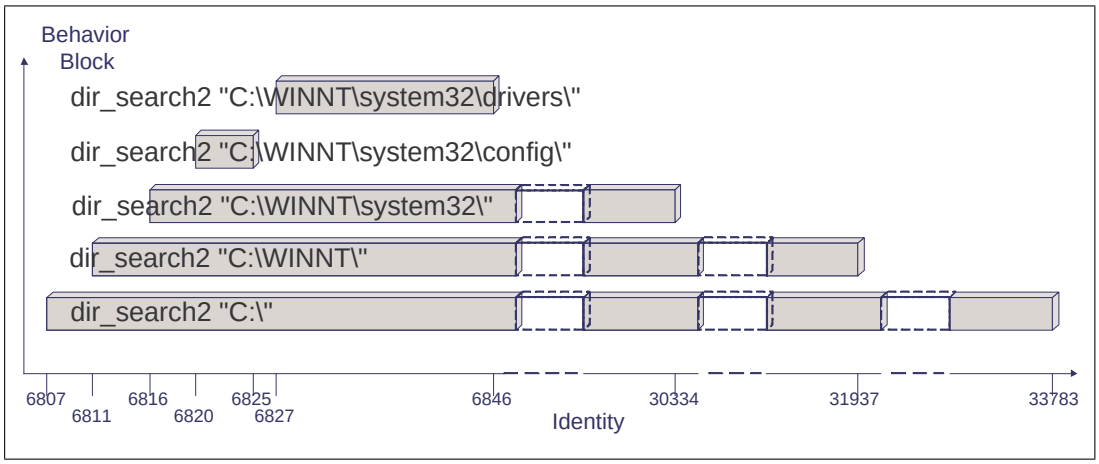


Figure 7.7: Bagle.at search_all_dir_recursive Macro Behavior

But for detection, we want to know as soon as possible. We can set threshold for a certain depth of directory recursion as a reduced criterion. By choosing an ad hoc depth of 3, we can reasonable say that it is a recursive directory search by the time the `dir_search` block reaches the path "`C:\WINNT\system32\config`" at identity 6825.

For some behaviors, a certain amount of expert knowledge is required to determine the reasonable trade-offs for identifying the macro behaviors.

7.6 Speed of Behavior Identification or Detection

One interesting piece of information we would like to find out is how fast can we detect a malware behavior, or what is the time delay. Analysis on this kind of information can provide more insight into malwares.

7.6.1 Unit of Measurement: Delta Time

The first problem that we face is the unit of time measurement. By modifying strace, each system call requires one API call like the NtQuerySystemTime to get the system time which is a 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601 (UTC). We need at least another API call to translate the system time into a human-readable time format. To use real time as our unit of measurement would mean each system call captured will require at least one additional system call, which may impact the performance severely.

As a trade-off, we use the identity counter as our unit of time measurement. The identity is a unique number accompanying each system call event that shows the position of the system call within the strace output sequence.

Rather than to start counting from the monitoring process, we start counting from the moment the malware is activated; we call this the *delta time*.

7.6.2 Example: Identification of survive_system_reboot Behavior

Let us take the `survive_system_reboot` behavior for our example. This complex behavior is modeled by correlating `file_copy` or `file_create`, with `registry_add startup` or `registry_add service`. The time taken to identify the slower basic behavior is used as the identification time of `survive_system_reboot`. The details of the speed of behavior detection among malwares can be found in Appendix D.5.

From Figure 7.8, which summarizes the details, we can see that the LovGate family display the `survive_system_reboot` behavior fastest, at just

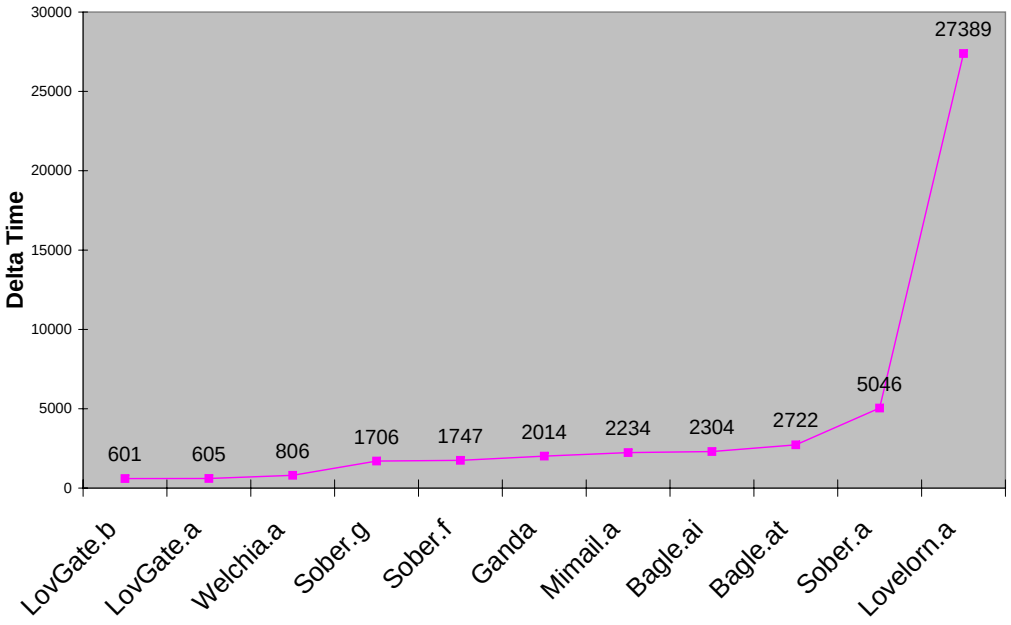


Figure 7.8: survive_system_reboot Detection Speed in Delta Time

above delta time 600, while Lovelorn.a only show that behavior at over delta time 26,000. The most frequent detection times are between delta time 2000 and 3000.

While 2000 delta time might seem like a lot, we must remember that it is the number of system calls, which can occur within seconds in real time. We can also see that the range of detection speed among malwares is quite large, which tells us that although these malwares share common behaviors, the order in which these behaviors occur varies between malwares.

7.6.3 Importance of Detection Speed

It is important to study how fast our system can detect malware behaviors as it directly impacts the effectiveness of our future real-time system implementation. We would like to know how urgent it is to stop certain actions.

Based on the study of malware descriptions in Chapter 4, we find that the

malware trend has changed significantly over the years. In the past, viruses tend to display destructive properties, thus it is imperative to be able to detect early. But currently, the trend is for the malwares to allow the hackers to control the infected hosts as zombies, so we have to change our focus.

We acknowledge that because our behavioral approach needs the malwares to display certain behaviors before they can be accurately detected, the accuracy of the detection is proportional to the time delay between the start and detection of the malware. In other words, the more accurate our detection is, the more time we have to wait. This means that the drawback of our approach is that it is slower than the misuse-signature based approach, and it cannot respond to infections as fast.

Our argument is that because of the changes in the malware behavioral trend, we can afford to be a little slower in detecting the malwares these days. And while our behavioral approach is inferior to the misuse-signature based approach when it comes to detecting known malwares as the signature approach is preventive, our approach is not that bad either.

Chapter 8

Conclusions and Further Works

8.1 Conclusions

This research attempts to study the feasibility of detecting malwares based on behaviors and forms the basis of a new behavior-based detection system. The reason for this approach is that we believe all malwares share some common behaviors, and malwares within the same families display more similar behaviors.

We attempt to infer high-level behaviors from the native API system call traces. But rather than simply using sequences of API calls to build behavior signatures like many others, we built semantically rich behavioral signatures based on context provided the system call and reverse engineering based on descriptions provided by anti-virus companies. By correlating related behavioral signatures, we were able to detect more complex behavioral functions. In our behavioral analysis, we were successful in identifying some behaviors common to all or most of our malware samples, but not to the set of normal applications used as baseline. We were also able to ob-

serve some interesting features of the malwares by studying the behavioral information provided by the framework. The results we got bode well for the feasibility of our behavior-based approach.

8.2 Further Works

While the framework has shown to be capable of detecting high level malware behaviors, there is still a lot of work to be done to improve its capabilities. In fact, this research has made us aware of many other questions that we would like to answer. We will discuss a couple of research items that we would like to work on in the future.

8.2.1 Modifiers

Throughout the thesis, we have emphasized that we use each malware behavior without weightage in both detection and malware similarity analysis. To improve the framework, we need to consider adding modifiers to the following items:

Risk of Behaviors: different behaviors pose different levels of risks. For example, opening and listening a network is riskier than querying a registry key. Modifiers should be added to the behaviors to let riskier behaviors have more weightage.

Risk of Parameters: the different parameters of the behaviors also have risks. For example, adding a registry key to registry paths that allow computer programs to run at boot time is riskier than any other paths. Modifiers should be added to reflect this information.

Similarity between Related Behaviors: when studying the similarity of mal-

wares, related behaviors should contribute to the similarity index. For example, if two malwares have the `irc_connect` and `http_connect` behaviors respectively, these two related network-based behaviors should contribute to the similarity index even though they are not identical.

Weightage of Correlated Behaviors in Different Processes: we need to consider the situation where correlated behaviors occur in two different processes. The processes could be related (sub-processes or LPC), or unrelated (code injection). Should correlated behaviors in related processes have higher weightage than those in unrelated processes, or vice versa?

In other work involving malware behaviors [26, 52], the authors assigned different weights to behaviors, or between behaviors, in a seemingly ad hoc manner. We believe that we need to study the behaviors of a larger number of malwares before we can derive the required weightages or modifiers.

8.2.2 Behavior-based System Implementation

As our current work is proof-of-concept, detection and analysis of malware behaviors are all done off-line. We hope that the knowledge gained from this area of research will be helpful in developing a real-time behavior-based intrusion prevention system in the future. The current work has shown the basic requirements for such systems, and the difficulties that will be encountered.

In addition, because our system monitors all system resources like files and registry, it is possible to adapt the implementation to backtrack [25] to the source of an unknown infection. This means the system has the potential to recover from an intrusion and undo some damages.

Bibliography

[1] Frank Apap, Andrew Honig, Shlomo Hershkop, Eleazar Eskin, and Salvatore J. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, October 2003. `http://www1.cs.columbia.edu/~sh553/papers/drafts/rad-dist02.pdf`.

[2] Robert M. Balzer and Neil M. Goldman. Mediating Connectors. In *Proceedings of the ICDCS Workshop on Electronic Commerce and Web-Based Applications 1999*, page 0073, 1999.

[3] Robert M. Balzer and Neil M. Goldman. Mediating Connectors: A Non-By Passable Process Wrapping Technology. In *Proceedings of the 1st DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 2, page 1361, 2000.

[4] Piotr Bania. Windows Syscall Shellcode. *SecurityFocus*, 4 August 2005. Retrieved on 26 June 2006 from `http://www.securityfocus.com/infocus/1844`.

[5] Jennifer Barrett. Are There More MyDoom Attacks to Come? Newsweek Technology, 3 February 2004. `http://msnbc.msn.com/id/4154289/`.

[6] Ulrich Bayer. TTAnalyze: A Tool for Analyzing Malware. Master's thesis, Technical University of Vienna, Information Systems Institute and Institute of Computer Aided Automation, 12 December 2005. `http://www.seclab.tuwien.ac.at/people/ulli/TTAnalyze_A_Tool_for_Analyzing_Malware.pdf`.

[7] Vesselin Bontchev. Current Status of the CARO Malware Naming Scheme. In *Proceedings of The 15th International Virus Bulletin Conference (VB2005)*, Dublin, Ireland, October 2005. `http://www.people.frisk-software.com/~bontchev/papers/naming.html`.

[8] CAIDA. CAIDA Analysis of Code-Red, 21 June 2005. Retrieved on 12 June 2005 from `http://www.caida.org/analysis/security/code-red/`.

[9] F-Secure Computer Virus Info Center. F-Secure Computer Virus Naming. `http://www.f-secure.com/v-descs/info/name.shtml`.

[10] Norman SandBox Information Center. Norman SandBox Live. Online Software Behavioral Analysis Demo. http://sandbox.norman.no/live_4.html.

[11] Simon P. Chung and Aloysius K. Mok. On Random-Inspection-Based Intrusion Detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, volume 3858 of *LNCS*, pages 165–184, Seattle, WA, USA, 7-9 September 2005.

[12] BindView Corporation. Strace for NT. Windows Process Monitor. http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm.

[13] MITRE Corporation. Common Malware Enumeration (CME) Project. http://cme.mitre.org.

[14] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Computer Security and Privacy 1996*, Oakland, California, 6-8 May 1996. http://www.cs.rpi.edu/~brancj/publications/ieee-sp-96-unix.pdf.

[15] Nick Gibson. Blackworm/Nyxem.E - Damaging but Not Catastrophic. Channel News, 3 February 2006. http://www.thechannelshow.com/ChannelNews/irep199.htm.

[16] Lea Goldman. Attack of the Clones. Forbes, 6 October 2002. http://www.cs.virginia.edu/~evans/press/forbes20020610.html.

[17] Katherine A. Heller, Krysta M. Svore, Angelos D. Keromytis, and Salvatore J. Stolfo. One Class Support Vector Machines for Detecting Anomalous Windows Registry Accesses. In *Proceedings of the IEEE ICDM Workshop on Data Mining for Computer Security (DMSEC'03)*, pages 2–9, Melbourne, Florida, November 2003. http://www.cs.fit.edu/~pkc/dmsec03/dmsec03notes.pdf.

[18] Shlomo Hershkop, Linh H. Bui, Ryan Ferster, and Salvatore J. Stolfo. Host-based Anomaly Detection by Wrapping File Systems. Technical report, Department of Computer Science, Columbia University, April 2004. http://www1.cs.columbia.edu/ids/publications/hershkop_bui_ferster_stolfo_04_1.pdf.

[19] Shlomo Hershkop, Ryan Ferster, Linh H. Bui, Ke Wang, and Salvatore J. Stolfo. Host-based Anomaly Detection by Wrapping File System Accesses. Technical report, Department of Computer Science, Columbia University, April 2003. http://www1.cs.columbia.edu/~kewang/paper/fwraps-final.pdf.

[20] Steven A. Hofmeyr, Anil Somayaji, and Stephanie Forrest. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6:151–180, 1998. http://www.cs.unm.edu/~steveah/jcs-accepted.pdf.

[21] Timothy Hollebeek and Rand Waltzman. The Role of Suspicion in Model-based Intrusion Detection. In *Proceedings of the 2004 Workshop on New Security Paradigms*, pages 87–94, Nova Scotia, Canada, 2004.

[22] Ruiqi Hu and Aloysius K. Mok. Detecting Unknown Massive Mailing Viruses Using Proactive Methods. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*, volume 3224 of *LNCS*, pages 82–101, Sophia Antipolis, France, 15-17 September 2004.

[23] Edward Hurley. The virus name game. SearchSecurity.com, 20 December 2002. `http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci870539,00.html`.

[24] Kaspersky Lab. `http://www.viruslist.com`.

[25] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems (TOCS)*, 23(1):51–76, February 2005.

[26] Tony Lee and Jigar J. Mody. Behavioral Classification. In *Proceedings of the 15th EICAR Annual Conference 2006*, Hamburg, Germany, 29 April 2006. `http://download.microsoft.com/download/9/0/7/907b6f32-dd84-46d9-8458-8dfdea61e6f3/Behavioral_Classification.doc`.

[27] Jay Lyman. Name That Worm - How Computer Viruses Get Their Names. NewsFactor, 8 January 2002. `http://www.newsfactor.com/perl/story/15662.html`.

[28] Microsoft. Windows 98 DDK, Windows 2000 Driver Development Kit - October 2000 Edition, Windows XP Service Pack 1 Driver Development Kit. CDROM 1009.1 from MSDN Academic Alliance, February 2003.

[29] Mike Musgrove. Who names computer viruses? Everybody. Washington Post, 26 February 2004. `http://www.msnbc.msn.com/id/4376005/`.

[30] Ryan Naraine. From Melissa to Zotob: 10 Years of Windows Worms. eWEEK.com, 24 August 2005. `http://www.eweek.com/article2/0,1895,1851792,00.asp`.

[31] Gary Nebbett. *Windows NT/2000 - Native API References*. SAMS, 1 edition, 1999.

[32] CBC News. Sasser virus hits internet users, 3 May 2004. `http://www.cbc.ca/stories/2004/05/03/sci-tech/sass040503`.

[33] Tomasz Nowak. *Undocumented Functions Microsoft Windows NT-2000*. NTinterlnals, 2000. `http://undocumented.ntinternals.net`.

[34] Niels Provos. Honeyd. Virtual Honeypot Daemon. Available from http://www.citi.umich.edu/u/provos/honeyd.

[35] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium 2003*, pages 257–272, Washington, DC, USA, August 2003. http://www.citi.umich.edu/u/provos/papers/systrace.pdf.

[36] Costin Riau. A Virus by Any Other Name: Virus Naming Practices. SecurityFocus, 3 June 2002. http://www.securityfocus.com/infocus/1587.

[37] Rohitab. API Monitor. Registry activity monitoring tool. http://www.rohitab.com/apimonitor/.

[38] Mark Russinovich. Inside the Native API, 23 November 2004. http://www.sysinternals.com/Information/NativeApi.html.

[39] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 8 edition, December 2004.

[40] Victor Skormin, Alexander Volynkin, Douglas Summerville, and James Moronski. In The Search of the Gene of Self-Replication In Malicious Codes. Presentation at 6th IEEE Information Assurance Workshop, 15-17 June 2005. http://www.itoc.usma.edu/Workshop/2005/Papers/Follow%20ups/TheGene.pdf.

[41] Victor A. Skormin, Douglas H. Summerville, and James S. Moronski. Detecting Malicious Codes by the Presence of Their Gene of Self-replication. In *Proceedings of the 2nd International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security (MMM-ACNS 2003)*, volume 2776 of *LNCS*, pages 206–216, St. Petersburg, Russia, 21-23 September 2003.

[42] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 3 edition, August 2000.

[43] Anil Somayaji and Stephanie Forrest. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium 2000*, Denver, CO, August 2000. http://www.cs.unm.edu/%7Eimmsec/publications/uss-2000.pdf.

[44] Anil Buntwal Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, Columbia University, Department of Computer Science, 2002. Retrieved on 13 Feb 2006 from http://www1.cs.columbia.edu/~locasto/projects/candidacy/papers/somayaji2002thesis.pdf.

[45] Lance Spitzner. Honeytokens: The Other Honeypot. *SecurityFocus*, 17 July 2003. Retrieved on 18 June 2004 from http://www.securityfocus.com/infocus/1713.

[46] Salvatore J. Stolfo, Frank Apap, Eleazar Eskin, Katherine Heller, Shlomo Hershkop, Andrew Honig, and Krysta Svore. A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection. *Journal of Computer Security*, 13(4):659–693, 2005. http://www1.cs.columbia.edu/ids/publications/A%20comparative%20Evaluation%20of%20Two%20Algorithms%20for%20Windows%20Registry%20Anomaly%20Detection.pdf.

[47] Douglas Summerville, Victor Skormin, Alexander Volynkin, and James Moronski. Prevention of Information Attacks by Run-Time Detection of Self-replication in Computer Codes. In *Proceedings of the 3rd International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security (MMM-ACNS 2005)*, volume 3685 of *LNCS*, page 54, St. Petersburg, Russia, 24-28 September 2005.

[48] Sysinternals. Filemon. File system activity monitoring tool. http://www.microsoft.com/technet/sysinternals/Utilities/Filemon.mspx.

[49] Sysinternals. Regmon. Registry activity monitoring tool. http://www.microsoft.com/technet/sysinternals/SystemInformation/Regmon.mspx.

[50] Jamie Twycross and Matthew M. Williamson. Implementing and Testing a Virus Throttle. In *Proceedings of the 12th USENIX Security Symposium 2003*, pages 285–294, Washington, DC, USA, August 2003. http://www.hpl.hp.com/techreports/2003/HPL-2003-103.pdf.

[51] VMware. VMware Workstation. Virtualization Software. http://www.vmware.com.

[52] Matthew Evan Wagner. Behavior Oriented Detection of Malicious Code at Run-Time. Master's thesis, College of Engineering at Florida Institute of Technology, 2002. http://www.se.fit.edu/Gatekeeper/papers/bodmalcode.pdf.

[53] Joe Wells. *How Scientific Naming Works*. WildList Organization International. http://www.wildlist.org/naming.htm.

[54] Carsten Willems. CWSandbox Live Demo. Online Software Behavioral Analysis Demo, 2006. http://www.cwsandbox.org/.

[55] Carsten Willems. Description of the CWSandbox. Work in progress at RWTH Aachen, 15 March 2006. http://www.consolo.de/paper/CWSandbox_Description.pdf.

[56] Carsten Willems. Usage of the CWSandbox. Work in progress at RWTH Aachen, 14 March 2006. http://www.consolo.de/paper/CWSandbox_Usage.pdf.

[57] Matthew Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2002. `http://www.acsac.org/2002/papers/97.pdf`.

[58] Matthew M. Williamson, Jamie Twycross, Jonathan Griffin, and Andy Norman. Virus Throttling. *Virus Bulletin*, March 2003. `http://www.hpl.hp.com/techreports/2003/HPL-2003-69.pdf`.

[59] J.-Y. Xu, A. H. Sung, P. Chavez, and S. Mukkamala. Polymorphic Malicious Executable Scanner by API Sequence Analysis. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems (HIS 2004)*, pages 378–383, 05-08 Dec 2004.

Appendix A

Variants Within Malware Families

Family	Variants	Variant Names
Mytob	21	a, ar, au, ba, bd, be, bf, bi, bk, bt, bw, c, h, q, r, t, u, v, w, x, y
Bagle	19	a, ah, ai, as, at, au, ay, b, ba, c, dx, e, g, gen, i, j, s, y, z
NetSky	13	aa, af, b, c, d, m, o, q, r, t, x, y, z
Mydoom	11	a, ab, b, e, g, l, m, q, r, t, u
Mimail	8	a, c, e, f, g, h, j, q
Lentin	6	a, g, j, m, o, v
LovGate	4	a, ad, ae, w
Zafi	2	b, d
Tanatos	2	a, b
Sobig	2	a, f
Klez	2	a, h
Swen	1	
BadtransII	1	

Table A.1: Variants of Top 13 Malware Families

Appendix B

Behavior Functions Compilation

Legend:
0 - behavior not seen,
1 - behavior always seen,
2 - behavior seen only under certain conditions.

Functions	Klez.a	Klez.e	Klez.h	Zafi.a	Bagle.a	Bagle.z	Bagle.ai	Bagle.at	Ganda	Gibe.a	Lentin.a	LovGate.a	LovGate.ad	LovGate.b	Lovelorn.a	Lovesan.a	Vimal.a	Mydoom.a	Sober.a	Sober.f	Sober.g	Sobig.a	SpyBot.a	Welchia.a
file_copy others	1	0	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1
file_copy System	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	0	0	1	0	0	0	0	1	0
file_copy Windows	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0	0
file_copy share	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
file_copy remote	2	2	2	0	0	0	0	0	0	0	0	2	2	2	0	0	0	0	0	0	0	1	0	0
file_create others	0	0	1	0	0	0	0	0	0	0	0	2	0	1	1	2	2	1	1	1	0	0	0	0
file_create System	1	1	0	1	0	1	1	1	0	0	0	1	1	1	1	0	0	1	1	1	1	0	1	0
file_create Windows	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1	0	0
file_append others	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
file_append System	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	1	0
file_append Windows	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0
file_attrib others	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
file_attrib System	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
file_modify	2	2	2	0	0	0	2	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
file_property TIME	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
file_rename	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
file_delete	0	2	2	0	0	1	0	0	0	2	0	0	0	0	0	0	1	0	0	0	0	1	0	0
file_execute others	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	2
file_execute System	1	0	0	0	0	0	2	2	0	0	0	1	1	0	0	1	0	0	1	0	1	0	2	0
file_execute Windows	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
file_execute notepad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	2	0	0	0
file_execute calc	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
file_execute IEXPLORE	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

file_read others	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
file_read System	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2	0	0	0	
file_read Windows	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	
file_access others	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	
file_access System	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
file_access Windows	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	2	0	0	
file_access OAB	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	
file_access WAB	1	1	1	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	
file_load System	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	
ini_modify win.ini	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	
ini_modify system.ini	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	
create_autorun	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	
dir_create local	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
dir_create remote	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	
find_dir	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
find_data_files	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0
find_bin_files	0	1	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	
search_all_dir_recursive	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	1	0	1	1	1	0	0
search_specific_dir_recursive	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0
mutex_check	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
event_create	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	
hidden_msgbox	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	
window_box_monitor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
msgbox	0	0	0	2	0	1	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	0
harvest_emails	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	0
sendmail_with_attachment	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0	1	1	1	1	1	0
sendmail	1	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0	1	0	0	0	0	1	0
reply_inbox_email /Outlook_MAPI	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
check_system_date	1	1	0	1	1	1	1	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0
check_system_information	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0
zombie	0	0	0	0	1	1	1	1	0	1	0	1	1	1	0	0	0	0	0	0	0	1	0
code_injection	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0
keylogger	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	2	0
share_enum	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0
remote_share_mount	1	1	0	0	0	0	0	0	0	0	0	2	2	2	0	0	0	0	0	0	0	0	0
remote_share_activity	1	0	0	0	0	0	0	0	0	0	0	2	2	2	0	0	0	0	0	0	0	0	0
scan_network	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1
dns_resolve	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	0	0	1	0	0	1
http_connect	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
download_inet	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	0	0	1	1	0	2
ntpdate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
irc_connect	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
netbios_connect	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1
network_connect	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
ping	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

listen_port	0	0	0	0	0	0	1	1	0	1	0	1	1	1	0	1	0	1	0	0	0	0	1	1
registry_modify shell	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
registry_add others	0	0	0	1	1	0	0	0	1	1	0	0	0	1	0	0	1	1	0	0	0	0	1	0
registry_add startup	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0
registry_add service	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1
registry_add dll component	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
registry_delete restart	0	2	2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_delete service	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_enum restart	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_enum services	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_query others	0	1	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	1
registry_query restart	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_query NameServer	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_query SHELL	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0
registry_query SMTP	0	0	1	1	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
registry_query WAB	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
start_from_internet_explorer	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
start_from_outlook	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
start_from_network_share	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0
start_from_windows_exploits	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
date_activated	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
date_activated_payload	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
suspicious_file	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
suspicious_email_attachment	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0

Table B.1: Behavior Function Compilation

Appendix C

Complex or Correlated Behaviors

C.1 Survive System Reboot

Functions	Klez.a	Klez.e	Klez.h	Zafi.a	Bagle.a	Bagle.z	Bagle.ai	Bagle.at	Ganda	Gibe.a	Lentin.a	LovGate.a	LovGate.ad	LovGate.b	Lovelorn.a	Lovesan.a	Mimail.a	Mydoom.a	Sober.a	Sober.f	Sober.g	Sobig.a	SpyBot.a	Welchia.a	Total
registry_add startup + file_copy/file_create	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	22
registry_add service /service_create + file_copy/file_create	0	1	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	7
registry_modify shell + file_copy/file_create	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	4
survive_system_reboot	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	24

Table C.1: Correlated Survive System Reboot Behavior

C.2 Find Email Addresses

Functions	Klez.a	Klez.e	Klez.h	Zafi.a	Bagle.a	Bagle.z	Bagle.ai	Bagle.at	Ganda	Gibe.a	Lentin.a	LovGate.a	LovGate.ad	LovGate.b	Lovelorn.a	Lovesan.a	Mimail.a	Mydoom.a	Sober.a	Sober.f	Sober.g	Sobig.a	SpyBot.a	Welchia.a	Total
search_all_dir_recursive + find data files + harvest emails	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	1	0	1	1	1	1	0	0	16
search_specific_dir_recursive + find data files + harvest emails	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	7
find_email_addresses	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	20

Table C.2: Correlated Find Email Addresses Behaviors

C.3 Malware Local Replication

Functions	Klez.a	Klez.e	Klez.h	Zafi.a	Bagle.a	Bagle.z	Bagle.ai	Bagle.at	Ganda	Gibe.a	Lentin.a	LovGate.a	LovGate.ad	LovGate.b	Lovelorn.a	Lovesan.a	Mimail.a	Mydoom.a	Sober.a	Sober.f	Sober.g	Sobig.a	SpyBot.a	Welchia.a	Total
search_specific_dir_recursive + find_bin_files + file_modify	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
search_all_dir_recursive + find_bin_files + file_modify	0	1	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	5
local_replication	0	1	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	5

Table C.3: Correlated Local Replication Behaviors

Appendix D

Behavior Analysis

D.1 Malware Detected Behaviors

Legend:
0 - behavior not seen,
y - behavior seen from both descriptions and trace,
n - behavior seen from in descriptions but not in trace.

Functions	Bagle.ai	Bagle.at	Ganda	LovGate.a	LovGate.b	Lovelorn.a	Mimail.a	Sober.a	Sober.f	Sober.g	Welchia.a
file_copy others	0	0	0	0	0	n	0	0	0	0	y
file_copy System	y	y	0	y	y	0	0	0	0	0	0
file_copy Windows	0	n	y	0	0	0	y	0	0	0	0
file_create others	0	0	0	n	n	n	0	y	y	0	0
file_create System	y	y	0	y	y	y	0	y	y	y	0
file_create Windows	0	0	0	0	0	0	y	0	0	0	0
find_data_files	y	y	y	y	y	y	n	y	y	y	0
search_all_dir_recursive	y	y	y	0	n	y	n	y	y	y	0
search_specific_dir_recursive	0	0	y	y	y	0	n	0	0	0	y
registry_modify shell	0	0	0	y	y	0	0	0	0	0	0
registry_add startup	y	y	y	y	y	y	y	y	y	y	0
registry_add service	0	0	n	y	y	0	0	0	0	0	y

Table D.1: Malware Detected Behaviors

D.2 Malware Detected Behaviors in Normal Application

Legend:
0 - behavior not seen,
1 - detected behavior.

Functions	Acrobat Reader	Ghostview	Internet Explorer	ICQ	MSN	Windows Media Player	WinAmp	Access	Excel	Outlook	PowerPoint	Word	FrontPage	WinZip	WinRAR
file_copy others	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
file_copy System	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
file_copy Windows	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
file_create others	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
file_create System	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
file_create Windows	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
find_data_files	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
find_bin_files	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
search_all_dir_recursive	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
search_specific_dir_recursive	1	0	0	0	1	0	0	1	1	0	1	1	1	1	0
registry_modify shell	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_add startup	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
registry_add service	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
registry_add dll component	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table D.2: Detected Malware Behaviors in Normal Application

D.3 Detected Correlated survive_system_reboot Behaviors

Legend:
0 - behavior not seen,
1 - detected behavior.

Functions	Bagle.ai	Bagle.at	Ganda	LovGate.a	LovGate.b	Lovelorn.a	Mimail.a	Sober.a	Sober.f	Sober.g	Welchia.a	Acrobat Reader	Ghostview	Internet Explorer	ICQ	MSN	Windows Media Player	WinAmp	Access	Excel	Outlook	PowerPoint	Word	FrontPage	WinZip	WinRAR
survive_system_reboot1 (registry_add startup + file_copy/file_create)	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
survive_system_reboot2 (registry_add service + file_copy/file_create)	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
survive_system_reboot3 (registry_modify shell + file_copy/file_create)	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
survive_system_reboot4 (registry_add dll component + file_copy/file_create)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
survive_system_reboot (Generalized)	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table D.3: Detected Correlated survive_system_reboot Behaviors

D.4 Detected Correlated find_email_addresses Behaviors

Legend:
0 - behavior not seen,
1 - detected behavior.

Functions	Bagle.ai	Bagle.at	Ganda	LovGate.a	LovGate.b	Lovelorn.a	Mimail.a	Sober.a	Sober.f	Sober.g	Welchia.a	Acrobat Reader	Ghostview	Internet Explorer	ICQ	MSN	Windows Media Player	WinAmp	Access	Excel	Outlook	PowerPoint	Word	FrontPage	WinZip	WinRAR
find_email_addresses1 (search_all_dir_recursive + find_data_files + file_open/file_read)	1	1	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
find_email_addresses2 (search_specific_dir_recursive + find_data_files + file_open/file_read)	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
find_email_addresses (Generalized)	1	1	1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table D.4: Detected find_email_addresses Behaviors

D.5 Detection Speed of survive_system_reboot Basic Behavior

Basic Blocks	Bagle.ai	Bagle.at	Ganda	LovGate.a	LovGate.b	Lovelorn.a	Mimail.a	Sober.a	Sober.f	Sober.g	Welchia.a
file_copy / file_create	2304	2722	1986	530	526	26810	2232	1866	1747	1706	769
registry_add startup / registry_add service	2272	2690	2014	605	601	27389	2234	5046	1735	1692	806

Table D.5: survive_system_reboot Detection in Delta Time

Appendix E

Kaspersky Lab Email-Worm.Win32.Bagle.ai Description

Home / Viruses / Virus Encyclopedia / Malware Descriptions / Network Worms / Email Worms

Email-Worm.Win32.Bagle.ai

Other versions: .a, .aa, .ah, .al, .an, .ao, .as, .at, .au, .ax, .ay, .b, .bb, .bj, .bn, .bo, .c, .cc, .ch, .cl, .cy, .d, .da, .dx, .e, .eb, .ef, .eq, .ek, .f, .fb, .fi, .fm, .fv, .gm, .i, .j, .k, .l, .m, .n, .p, .q, .r, .s, .t, .y, .z

Aliases

Email-Worm.Win32.Bagle.ai (Kaspersky Lab) is also known as: I-Worm.Bagle.ai (Kaspersky Lab), W32/Bagle.ai@MM (McAfee), W32.Beagle.AG@mm (Symantec), Win32.HLLM.Beagle.20480 (Doctor Web), W32/Bagle-AI (Sophos), Win32/Bagle.AI@mm (RAV), WORM_BAGLE.AH (Trend Micro), Worm/Bagle.AI (H+BEDV), W32/Bagle.AI@mm (FRISK), Win32:Beagle-AH (ALWIL), I-Worm/Bagle.AI (Grisoft), Win32.Bagle.AJ@mm (SOFTWIN), Worm.Bagle.AG.2 (ClamAV), W32/Bagle.AH.worm (Panda), Win32/Bagle.AH (Eset)

Description added	Jul 20 2004
Behavior	Email Worm

Technical details

This worm spreads via the Internet as an attachment to infected messages and also via P2P networks.

It is approximately 20 KB in size and packed using PEX.

Installation

Once launched, the worm copies itself to the Windows system directory as winxp.exe. It then registers this file in the system registry to ensure that this file is launched each time the system is started.

```
[HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
 "key"="%system%\winxp.exe"
```

The worm also creates the following files in the Windows system directory:

```
winxp.exeopen
winxp.exeopenopen
winxp.exeopenopenopen
winxp.exeopenopenopenopen
```

Propagation

The worm searches disks for files with extensions from the following lists. It sends itself to all addresses harvested from these files.

```
adb             mdx             shtm
asp             mht             stm
cfg             mmf             tbb
cgi             msg             txt
dbx             nch             uin
dhtm            ods             wab
eml             oft             wsh
htm             php             xls
jsp             pl              xml
mbx             sht
```

It uses its own SMTP server to send messages.

Infected messages

Message header:

```
Re:
```

Versions of message body:

```
>Animals
>foto3 and MP3
>fotogalary and Music
>fotoinfo
>Lovely animals
>Predators
>Screen and Music
>The snake
```

Attachment name:

```
Cat
Cool_MP3
Dog
Doll
Fish
Garry
MP3
Music_MP3
New_MP3_Player
```

Attachment name:

```
com
cpl
exe
scr
zip
```

The worm can send itself as a password protected ZIP archive. If it does this, the password will be shown in the message body. The password may be in text or graphical format.

The worm will not send itself to addresses containing text strings from the list below:

```
@avp.
@foo
@hotmail
@iana
@messagelab
@microsoft
@msn
abuse
admin
anyone@
bsd
bugs@
cafee
certific
contract@
feste
free-av
```

Propagation via P2P

The worm searches disks for folders containing the text string shar. It then copies itself several times to these folders under the following names:

```
ACDSee 9.exe
Adobe Photoshop 9 full.exe
Ahead Nero 7.exe
Kaspersky Antivirus 5.0
KAV 5.0
Matrix 3 Revolution English Subtitles.exe
Microsoft Office 2003 Crack, Working!.exe
Microsoft Office XP working Crack, Keygen.exe
Microsoft Windows XP, WinXP Crack, working Keygen.exe
Opera 8 New!.exe
Porno pics arhive, xxx.exe
Porno Screensaver.scr
Porno, sex, oral, anal cool, awesome!!.exe
Serials.txt.exe
WinAmp 5 Pro Keygen Crack Update.exe
WinAmp 6 New!.exe
Windown Longhorn Beta Leak.exe
Windows Sourcecode update.doc.exe
XXX hardcore images.exe
```

Remote administration

The worm opens port 1080 and another port chosen at random. It then tracks port activity.

Other

The worm is programmed to cease activity and self-destruct after 5th May 2006.

It tracks the execution of most well-known antivirus products and firewalls and terminates these processes..

The worm's body contains a list of URLs. It attempts to download from these sites. At the moment of writing, none of the sites are functioning.

Appendix F

Examples of Converted Malware Descriptions

The converted malware descriptions are in a pseudo language that was created based on the Perl language and UNIX shell scripting conventions. The argument types are *$scalar variable*, *@list* and *&stream*. Any text behind '#' are comments. Underscored arguments like *$_* or *$_variable* represent unknown arguments streamed from the preceding function. The symbol '|' is the OR operator.

As there are some descriptions that are really unclear or ambiguous, we use a dummy function "unknown_process" as a stub first.

F.1 Email-Worm.Win32.Bagle.at

```
Kaspersky:   Email-Worm.Win32.Bagle.at;
             http://www.viruslist.com/en/viruses/encyclopedia?virusid=64658;
CA:          Win32.Bagle.AR;
             http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?ID=40602;
Trend:       WORM_BAGLE.AU;
             http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=WORM_BAGLE.AU&VSect=T;
DATE:        2004-10-29;
TYPE:        Win32 PE EXE;
COMPRESSION: PeX;
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # creates several mutexes to prevent WORM_NETSKY variants from running
    mutex_create "MuXxXxTENYKSDesignedAsTheFollowerOfSkynet-D" ;
    mutex_create "'D'r'o'p'p'e'd'S'k'y'N'e't'" ;
    mutex_create "_-oOaxX|-+S+-+k+-+y+-+N+-+e+-+t+-|XxKOo-_" ;
    mutex_create "[SkyNet.cz]SystemsMutex" ;
    mutex_create "AdmSkynetJklS003" ;
    mutex_create "____--->>>>U<<<<--____" ;
    mutex_create "_-oO]xX|-S-k-y-N-e-t-|Xx[Oo-_" ;

    # also creates unnamed mutexes for own thread synchronization purposes
    mutex_create $UNKNOWN_MUTEX ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    system_query %System% ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # When executed
    file_copy $SELF, %System%\wingo.exe ;
```

```
    registry_add "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
            wingo = %System%\wingo.exe ;

    file_copy $SELF, %Windows%\cjector.exe ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # scans the local fixed drives for files with the following extensions
    search_loc @LOCAL_DIRS ;

    find\_data\_files
         *.adb | *.asp | *.cfg | *.cgi | *.dbx | *.dhtm | *.eml | *.htm | *.jsp
         | *.mbx | *.mdx | *.mht | *.mmf | *.msg | *.nch | *.ods | *.oft | *.php
         | *.pl | *.sht | *.shtm | *.stm | *.tbb | *.txt | *.uin | *.wab | *.wsh
         | *.xls | *.xml ;


    # searches for email in files with above extensions
    grep $EMAIL_PATTERN @_ ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # skips email that contain
    grepv @avp | @foo | @hotmail | @iana | @messagelab | @microsoft | @msn
          | abuse | admin | anyone@ | bsd | bugs@ | cafee | certific | contract@
          | f-secur | feste | free-av | gold-certs@ | google | help@ | icrosoft
          | info@ | kasp | linux | listserv | local | news | nobody@ | noone@
          | noreply | ntivi | panda | pgp | postmaster@ | rating@ | root@
          | samples | sopho | spam | support | unix | update | winrar | winzip ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # may be four extra files created in process of generating attachments
    file_create %System%\wingo.exeopen ;
    file_create %System%\wingo.exeopenopen ;
    file_create %System%\wingo.exeopenopenopen ;
    file_create %System%\wingo.exeopenopenopenopen ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # If cannot find DNS server used by the local system
    # it tries to use the one at 217.5.97.137
    registry_query "HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\
                    Interfaces\*\NameServer" ; ;
    if $_REG_QUERY_STATUS == FAIL ; then
            unknown_process "217.5.97.137" ;
    fi

    # sends copies of itself to any email addresses it finds
    # sends e-mail using its own SMTP engine

    *sendmail @_COLLECTED_EMAILS, ATTACHMENT;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # enables spread through peer-to-peer file sharing networks, such as Kazaa

    NFILE1 =  "ACDSee 9.exe"
            | "Adobe Photoshop 9 full.exe"
            | "Ahead Nero 7.exe"
            | "Kaspersky Antivirus 5.0"
            | "KAV 5.0"
            | "Matrix 3 Revolution English Subtitles.exe"
            | "Microsoft Office 2003 Crack, Working!.exe"
            | "Microsoft Office XP working Crack, Keygen.exe"
            | "Microsoft Windows XP, WinXP Crack, working Keygen.exe"
            | "Opera 8 New!.exe"
            | "Porno pics arhive, xxx.exe"
            | "Porno Screensaver.scr"
            | "Porno, sex, oral, anal cool, awesome!!.exe"
            | "Serials.txt.exe"
            | "WinAmp 5 Pro Keygen Crack Update.exe"
```

```
            | "WinAmp 6 New!.exe"
            | "Windown Longhorn Beta Leak.exe"
            | "Windows Sourcecode update.doc.exe"
            | "XXX hardcore images.exe" ;

    # While searching for files with e-mail, also
    # searches for folders containing "shar"
    find_dir *shar* ;

    # drops copies of itself in folders
    file_copy "%System%\wingo.exe", @_DIR\$NFILE1 ;


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # opens a backdoor on TCP port 81
    listen_port TCP, 81;


    # installs a proxy server that can be controlled via this port
    # allowing remote access to the machine
    zombie ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # contains a list of 146 URLs
    for URL in @URL_LIST; do
            download_inet http://${URL}/g.jpg ;
            file_rename g.jpg %System%\re_file.exe ;
            file_execute %System%\re_file.exe ;
    done

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    check_system_date ;
    if $DATE == "2006-04-25"; then
            # stop functioning on April 25, 2006
            date_activated ;
            exit ;
    fi

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # terminates the following antivirus and security-related programs
    @AV_PROG1 = "alogserv.exe | APVXDWIN.EXE | ATUPDATER.EXE | AUPDATE.EXE
         | AUTODOWN.EXE | AUTOTRACE.EXE | AUTOUPDATE.EXE   | Avconsol.exe
         | AVENGINE.EXE | AVPUPD.EXE | Avsynmgr.exe | AVWUPD32.EXE | AVXQUAR.EXE
         | bawindo.exe  | blackd.exe | ccApp.exe | ccEvtMgr.exe | ccProxy.exe
         | ccPxySvc.exe | CFIAUDIT.EXE | DefWatch.exe  | DRWEBUPW.EXE
         | ESCANH95.EXE | ESCANHNT.EXE | FIREWALL.EXE | FrameworkService.exe
         | ICSSUPPNT.EXE | ICSUPP95.EXE | LUALL.EXE | LUCOMS~1.EXE | mcagent.exe
         | mcshield.exe | MCUPDATE.EXE | mcvsescn.exe | mcvsrte.exe
         | mcvsshld.exe | navapsvc.exe | navapw32.exe | NISUM.EXE | nopdb.exe
         | NPROTECT.EXE | NUPGRADE.EXE | OUTPOST.EXE | PavFires.exe
         | pavProxy.exe | pavsrv50.exe | Rtvscan.exe | RuLaunch.exe
         | SAVScan.exe | SHSTAT.EXE | SNDSrvc.exe | symlcsvc.exe | UPDATE.EXE
         | UpdaterUI.exe | Vshwin32.exe | VsStat.exe | VsTskMgr.exe" ;

    kill_process @AV_PROG1 ;

    # stop and disable Internet Connection Firewall (ICF)
    service_stop "Internet Connection Firewall" ;
    service_disable "Internet Connection Firewall" ;

    # Internet Connection Sharing (ICS) service  (the "SharedAccess" service)
    service_stop "Internet Connection Sharing" ;
    service_disable "Internet Connection Sharing" ;

    # Security Center service ("wscsvc" - introduced in XP SP2)
    service_stop "Security Center" ;
    service_disable "Security Center" ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
    # deletes several registry entries associated with WORM_NETSKY variants
    @AV_LIST1 = "9XHtProtect | Antivirus | EasyAV | FirewallSvr | HtProtect
                 | ICQ Net | ICQNet | Jammer2nd | KasperskyAVEng | MsInfo
                 | My AV | NetDy | Norton Antivirus AV | PandaAVEngine
                 | service | SkynetsRevenge | Special Firewall Service
                 | SysMonXP | Tiny AV | Zone Labs Client Ex" ;

    registry_delete "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
            @AV_LIST1 ;

    registry_delete "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
            @AV_LIST1 ;

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

F.2 Email-Worm.Win32.Sober.g

```
Kaspersky:   Email-Worm.Win32.Sober.g;
             http://www.viruslist.com/en/viruses/encyclopedia?virusid=50433;
CA:          Win32.Sober.G;
             http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?id=39112;
Trend:       WORM_SOBER.G;
             http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=WORM_SOBER.G&VSect=T;
DATE:        2004-06-03;
SOURCE:      Visual Basic;
COMPRESSION: UPX;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    system_query %System% ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # Once launched
    msgbox YesNo, "File not found",
            "Special-UnZip Data-Module\n\nis missing\n\nOpen with Notepad?" ;

    if msgbox.response == YES ; then
            # diversionary trick.

            process_status %SELF% ;

            FILE2 = "Converted_${_FILENAME}" ;

            file_create %System%\$FILE2 ;

            file_read %System%\$FILE2 ;
            # contains nonsense text

            file_execute "notepad.exe %System%\$FILE2" ;
    fi

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# creates a copy of itself under a name chosen at random from list below
# For example, winrun.exe, sysrunsmss32.exe, cryptsys.exe, discwinlog, dirspool

    STRING = "32 | crypt | data | diag | dir | disc | expolrer | host | log | run
         | service | smss32 | spool | sys | win" ;

    unknown_process $STRING ;

    VKEY1 = $_ ;
    VFILE1 = $_ ;

    file_create %System%\$VFILE1 ;

    # registered in the system registry auto-run key
    registry_add "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
            $VKEY1 = %System%\$VFILE1 ;

    registry_add "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
            $VKEY1 = %System%\$VFILE1 ;

    registry_add "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce",
            $VKEY1 = "%System%\$VFILE1 %1" ;


    # creates several other files used for its own purposes
    file_create %System%\bcegfds.lll ;
    file_create %System%\cvqaikxt.apk ;

    file_create %System%\datsobex.wwr ;

    file_create %System%\wincheck32.dats ;
    # contains list of email gather from the infected machine
```

```
    file_create %System%\winexpoder.dats ;
    # contains a list of corresponding recipient names to the email gathered

    file_create %System%\winzweier.dats ;
    # contains a list of randomly generated email addresses

    file_create %System%\xdatxzap.zxp ;

    file_create %System%\zhcarxxi.vvx ;

    file_create %System%\NoSpam.readme ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # searches local disks for files with ext
    search_loc @LOCAL_DIRS ;

    find\_data\_files
         *.abc | *.abd | *.abx | *.adb | *.ade | *.adp | *.adr | *.asp | *.bak
         | *.bas | *.cfg | *.cgi | *.cls | *.cms | *.csv | *.ctl | *.db | *.dbx
         | *.dhtm | *.doc | *.dsp | *.dsw | *.eml | *.fdb | *.frm | *.hlp
         | *.imb | *.imh | *.imh | *.imm | *.inbox | *.ini | *.jsp | *.ldb
         | *.ldif | *.log | *.mbx | *.mda | *.mdb | *.mde | *.mdw | *.mdx
         | *.mht | *.mmf | *.msg | *.nab | *.nch | *.nfo | *.nsf | *.nws
         | *.ods | *.oft | *.php | *.pl | *.pmr | *.pp | *.ppt | *.pst | *.rtf
         | *.shtml | *.slk | *.sln | *.stm | *.tbb | *.txt | *.uin | *.vap
         | *.vbs | *.vcf | *.wab | *.wsh | *.xhtml | *.xls | *.xml ;
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # discards any e-mail that contains
    grepv -dav | .dial. | .kundenserver. | .ppp. | .qmail@ | .sul.t- | @arin
          | @avp | @ca. | @example. | @foo. | @from. | @gmetref | @iana
          | @ikarus. | @kaspers | @messagelab | @msn | @nai. | @panda | @smtp.
          | @sophos | @spiegel. | @www | abuse | announce | antivir | anyone
          | anywhere | bellcore. | bitdefender | clicks. | clock | detection
          | domain. | emsisoft | ewido. | free-av | freeav | ftp. | gold-certs
          | google | host. | icrosoft. | ipt.aol | law2 | linux | mailer-daemon
          | me@ | members. | mozilla | msdn. | mustermann@ | nlpmail01. | nothing
          | office | password | postmas | reciver@ | redaktion | refer. | secure
          | service | smtp- | somebody | someone | spybot | sql. | subscribe
          | support | t-dialin | t-ipconnect | time | track. | user@ | variabel
          | verizon. | viren | virus | whatever@ | whoever@ | winrar | winzip
          | www. | you@ | yourname ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # gathers email addresses from files
    grep $EMAIL_PATTERN @_ ;

    # stores gathered email
    file_append &_, %System%\Wincheck32.dats ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # connects directly to the SMTP server to send messages
    sendmail @_EMAILS ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # downloads a copy of itself and saves in system directory
    URL = "home.arcor.de
            | people.freenet.de
            | home.pages.at
            | scifi.pages.at
            | free.pages.at" ;

    download_inet http://${URL}/DOERKGGG.EXE ;
    file_create %System%\DOERKGGG.EXE ;

    #and launch files
    file_execute %System%\DOERKGGG.EXE ;

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
    # tests live connection by resolve the following URLs
    dns_resolve microsoft.com ;
    dns_resolve bigfoot.com ;
    dns_resolve yahoo.com ;
    dns_resolve t-online.de ;

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```