

EFFICIENT STATIC ANALYSIS OF
EXECUTABLES FOR DETECTING MALICIOUS BEHAVIORS

THESIS

Submitted in Partial Fulfillment
of the REQUIREMENTS for the

Degree of

MASTER OF SCIENCE (Computer Science)

at the

POLYTECHNIC UNIVERSITY

by

Konstantin Rozinov

June 2005

Advisor

Date

Department Head

Date

Copy No. _____

Vita

Oct 25, 1981..... Born - Lvov, Ukraine

Sep 1999 – May 2003..... Bachelor of Science (Information Management),
Polytechnic University, Brooklyn, NY

Sep 2003 – May 2005..... Master of Science (Computer Science),
Polytechnic University, Brooklyn, NY

Acknowledgements

I thank Professor Nasir Memon, my advisor, for his helpful opinions, guidance, and support during the course of this thesis and throughout my undergraduate and graduate studies at Polytechnic University. I also thank Bjoern Luettmann, Ted Wroblicka, and Tom Reddington for their technical expertise and assistance during my internship at Bell Labs, where I began my preliminary research. I also want to thank the SFS Program at Polytechnic University for its support during my graduate studies.

AN ABSTRACT

EFFICIENT STATIC ANALYSIS OF

EXECUTABLES FOR DETECTING MALICIOUS BEHAVIORS

by

Konstantin Rozinov

Advisor: Dr. Nasir Memon

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science (Computer Science)

June 2005

Today, many anti-virus (AV) scanners primarily detect viruses by looking for simple virus signatures within the file being scanned. The signature of a virus is typically created by disassembling the virus into assembly code, analyzing it, and then selecting those sections of code that seem to be unique to the virus. The binary bits of those unique sections become the signature for the virus. However, this approach can be easily subverted by simply changing the virus's code (and thus the virus signature) in trivial ways. The FFSig is a virus signature that would encompass the sequence of API calls, instead of just a small bit of binary code (as shown above) which could be easily modified. Thus, the problem we try to solve in this thesis is as follows: Can an executable be efficiently and statically analyzed

to construct (extract), store, and compare a signature based on the sequence of Win32 API calls made by the executable?

The first step is the disassembly of the target executable, which yields an assembly version of the code. The first step also includes the analysis of the assembly code to simplify it and improve the analyzability of it. This results in abstractions and high-level representations of the assembly code. The second step is to extract certain malicious parts of the code in order to analyze it more closely. This step reduces and focuses the amount of code that has to be analyzed (in effect it decreases the complexity of the detection process). This is accomplished by using slicing techniques on certain areas of the assembly code. The third is to store the information from the slice using a finite state automaton (this becomes the signature). The fourth and final step is to use various similarity measures to compare different signatures and then produce a report.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Statement.....	2
1.3	Contributions of the Thesis.....	4
1.4	Assumptions and Background Knowledge.....	4
1.5	Organization of the Thesis.....	11
2	Related Work.....	12
3	Algorithm and Architecture Design	15
3.1	Disassembly and Analysis of Executable.....	16
3.2	Slicing Algorithm to Extract the Signature	25
3.3	An Automaton-Based Method for Storing the Signature	44
3.4	Similarity Measures for the Signatures	47
4	Chapter 4.....	55
4.1	Conclusion.....	55
4.2	Future Work.....	56
5	Appendixes.....	58
5.1	Appendix A: Functional Flows.....	58
6	Bibliography	63

List of Figures

Figure 1. Using <code>dumpbin</code> to show the various segments of an executable.....	8
Figure 2. Overview of the system architecture.	16
Figure 3. Unpacking an executable using UPX.	18
Figure 4. Two parallel arrays of pointers.....	24
Figure 5. The original program (left), and some example slices of the program...26	
Figure 6. A group of statements with a single successor. Nodes C, D, and E form a set with a single successor, F, not in the set. The flowgraph is shown before (left) and after (right) removing this set.....	27
Figure 7. Extract assembly code with an indexed jump statement.	28
Figure 8. Disassembled code for <code>main()</code> procedure of the test program in Figure 9. It is annotated with Use-Definition chains.....	30
Figure 9. Sample Test C program.	31
Figure 10. Control Flow Graph, Program Dependence Tree, and Control Dependence Graph for the program in Figure 8.	31
Figure 11. Slice of program given in Figure 8 w.r.t. Register <code>si</code> at Instruction 11.	32
Figure 12. Control flow graph of the program (left) and the assembly program code (right).....	35
Figure 13. A graph-based model.....	37

Figure 14. One possible partial preliminary signature extracted from a slice of the Bagle virus. It contains API functions that are in the suspicious API database, as well as non-suspicious API functions that are not in that database.	42
Figure 15. One possible final FFSig for the Bagle virus is constructed by augmenting the signature in Figure 14. The augmentation involves deleting non-suspicious API functions and adding program state (the address of the call site) and loop information.....	43
Figure 16. Derived FSA from the FFSig in Figure 15.....	46
Figure 17. The Euclidean Distance Measure.	49
Figure 18. The Needleman-Wunsch Algorithm used in aligning DNA sequences.	52
Figure 19. The sequence alignment algorithm in action on the sequences from Table 5. Match = +1, Mismatch = 0.	52
Figure 20. Cosine similarity measure.	53
Figure 21. Jaccard Coefficient.	53
Figure 22. Examples of Pearson's Correlations. (a) has a perfect positive linear relationship (+1). (b) has a perfect negative linear relationship (-1). (c) has a strong, but not perfect positive linear relationship. (d) has no linear relationship (0).....	54
Figure 23. Pearson's Correlation.	54

List of Tables

Table 1. Intel x86 registers.....	7
Table 2. Segments commonly found inside executables.....	8
Table 3. Sequence of system calls with and without program state information...44	
Table 4. Euclidean distance measure between FFSigs from Bagle.A and Bagle.B.	50
Table 5. The effect of misalignment in the Euclidean distance measure.....	51

List of Code

Code 1. The section table of Bagle.B virus showing UPX compression is used. ..	18
Code 2. The MS-DOS header defined inside winnt .h.	18
Code 3. Checking for the MS-DOS and PE signatures.	19
Code 4. The address of the entry point into an executable is defined.	20
Code 5. Calculating the address of the entry point.	21
Code 6. Output form PEDUMP.EXE showing imported functions.	23
Code 7. The subroutine in Bagle.A which makes contact with attacker websites. The gray portions of the code are the resulting intraprocedural slice w.r.t. hMem at location 00402D97.	39

Chapter 1

1 Introduction

1.1 Motivation

Today, many anti-virus (AV) scanners primarily detect viruses by looking for simple virus signatures within the file being scanned. The signature of a virus is typically created by disassembling the virus into assembly code, analyzing it, and then selecting those sections of code that seem to be unique to the virus. For example, the virus signature for the Chernobyl/CIH virus for one AV vendor is the following hexadecimal sequence [4]:

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

The binary bits of those unique sections become the signature for the virus. However, this approach can be easily subverted by simply changing the virus's code (and thus the virus signature) in trivial ways.

Most viruses in the wild today are of the "simple" type [1] - not encrypted or polymorphic, and many of them have variants that come out afterwards. These variants are inherently similar to the original virus [2], yet current signatures fail to detect these variants [3] without further updates from AV vendors. This indicates

that present-day signatures are too weak to withstand simple changes to the virus body (i.e. dates, port numbers, variable names, etc).

The motivation for this research started in June 2004, when the author began to reverse code engineer (RCE) the first variant of the Bagle (a.k.a. Beagle) virus. By August 2004, the author had fully reverse-engineered the virus by hand and noticed that the bodies of subsequent variants of the Bagle virus (there were 18 such variants by July 2004) were very similar to the original [2]. The differences lay in variables such as the port number that the virus listened on, the dates it was active, the websites it contacted, how often it would contact those sites, filenames that the virus dropped into the Windows system directory, icon pictures, and the like. The thing that changed much less often is the sequence of Win32 API calls made by the variants. This constitutes the functional flow (or code flowchart) that the virus goes through to achieve its goal. The steps and their order may vary slightly, but are generally very similar if not identical between successive variants.

This discovery led to the idea of what the author dubbed Functional Flow Signature (FFSig). The FFSig is a virus signature that would encompass the sequence of system calls, instead of just a small bit of binary code (as shown above) which could be easily modified. This leads us to the problem statement below.

1.2 Problem Statement

Analysis of binaries and executables can be generally classified into two classes: *static analysis* and *dynamic analysis*. Similarly, intrusion detection

techniques can be divided into the following two classes: *misuse detection* and *anomaly detection*. Static analysis can be defined as the process of evaluating an executable based on its form, structure, and content, without actually executing the program. Similarly, misuse detection is based on identifying and pattern-matching known signatures. On the other hand, dynamic analysis of an executable can be defined as the process of evaluating that executable based solely on its behavior during execution. Similarly, anomaly detection is based on learning normal behavior and then monitoring for abnormal deviations from the normal behavior. It is generally accepted that signature-based schemes are very accurate, but suffer from the inability to detect new attacks, while schemes based on dynamic analysis and anomaly detection are able to detect new attacks, but suffer from the difficulty of learning normal behavior and a high false positive rate.

Although there are several ways in which the FFSig could be generated, we focus our research on statically analyzing an executable. We avoid dynamic analysis because we do not want to run a potentially dangerous executable on a live system for the obvious reasons. We also avoid making any changes to the host system (Microsoft Windows) as is usually required when doing dynamic analysis and API hooking [9]. We utilize the research done by [5] in order to automatically and efficiently build a finite-state automaton (FSA) in order to remember the sequences of system calls.

Thus, the problem we try to solve in this thesis is as follows: Can an executable be efficiently and statically analyzed to construct (extract) and store a signature based on the sequence of Win32 API calls made by the executable,

followed by a slew of similarity measures to compare known and unknown signatures?

1.3 Contributions of the Thesis

Although much research has been conducted in the fields of static and dynamic analysis (see Related Work chapter), we believe that this thesis contributes something novel, namely a methodology for efficiently and statically analyzing a Windows executable, extracting specific sequences of Win32 API calls made by the executable, storing them efficiently by utilizing the method presented in [5], and then using several similarity measures to compare known and unknown signatures. Our algorithm does not require any modification to the host operating system and does not require the executable to be run as it would if it was dynamically analyzed. In addition, by only doing static analysis, this allows us to scan a large number of executables in a short period of time. If this was done via dynamic analysis, one would have to wait until the executable was executed to analyze it.

1.4 Assumptions and Background Knowledge

There are several assumptions we make about the executable and the surrounding environment. First, we assume that the host operating system is Microsoft Windows 2000 or XP. Second, we assume that the executable is malicious in nature and cannot be executed on the system without infecting the

system. Third, we assume that the executable is not compressed (packed) or encrypted. The most popular compression algorithms are UPX, PEX, and ASPack, and it is trivial to decompress executables compressed with these packers. We assume that the executable is written in C/C++ and is in the Portable Executable (PE) file format.

To have a good understanding of the research presented in this thesis, we briefly review some of the most important concepts. Mastery of the assembly language for the Intel chipset (x86) is by far the most important thing to have. With that, a solid understanding of how the various registers are used and their purposes is also required. It's important to note that the Intel processor accesses and stores memory in Little Endian order. Little Endian means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. For example, the following assembly instruction copies the value 1 into the EDX register:

Assembly	Hexadecimal
<code>MOV EDX, 1</code>	<code>BA 01 00 00 00</code>

In hexadecimal, 1 would be represented as 00000001h (4 bytes). However, since the Intel processor uses Little Endian order, it is stored and accessed as (lowest address) 01 00 00 00 (highest address). The BA above is the hexadecimal representation of the `MOV EDX, <immediate>` instruction on the Intel processor.

In addition, we assume that the reader is familiar with other basic x86 concepts, including registers, runtime data structures, and the stack. For the

purpose of completeness, we include the following table which presents a summary of the various registers typically used and encountered in x86 assembly:

Register	Size (in bits)	Purpose	
AX (EAX)	16 (32)	Main register used in arithmetic calculations. Also known as accumulator, as it holds results of arithmetic operations and function return values .	
BX (EBX)	16 (32)	The Base Register. Used to store the base address of the program.	
CX (ECX)	16 (32)	The Counter register is often used to hold a value representing the number of times a process is to be repeated. Used for loop and string operations.	
DX (EDX)	16 (32)	A general purpose register. Also used for I/O operations. Helps extend EAX to 64-bits.	
SI (ESI)	16 (32)	Source Index register. Used as an offset address in string and array operations. It holds the address from where to read data.	
DI (EDI)	16 (32)	Destination Index register. Used as an offset address in string and array operations. It holds the implied write address of all string operations.	
BP (EBP)	16 (32)	Base Pointer. It points to the bottom of the current stack frame. It is used to reference local variables.	
SP (ESP)	16 (32)	Stack Pointer. It points to the top of the current stack frame. It is used to reference local variables.	
IP (EIP)	16 (32)	The instruction pointer holds the address of the next instruction to be executed.	
CS	16	Code segment register. Base location of code section (.text section). Used for fetching instructions.	These registers are used to break up a program into parts. As it executes, the segment registers are assigned the base values of each segment. From here, offset values are used to access each command in the program. ¹
DS	16	Data segment register. Default location for variables (.data section). Used for data accesses.	
ES	16	Extra segment register. Used during string operations.	
SS	16	Stack segment register. Base location of the stack segment. Used when implicitly using SP or ESP or when explicitly using BP, EBP.	
EFLAGS	32	This register's bits represent several single-bit Boolean values, such as the sign, overflow, carry, and zero flags. It is modified after every mathematical operation. See	

¹ Modern operating system and applications use the (unsegmented or flat) memory model: all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space [26]. In the old days (DOS and Windows 3.1), a segmented memory model was used, whereby the memory was broken up into 64KB chunks called segments. Each of the segment registers would then be loaded with different values to point to different segments. A linear address would be calculated by taking the segment address, adding a

	below for more information.
--	-----------------------------

Table 1. Intel x86 registers.

We also include the following table which describes some common segments found inside executable images:

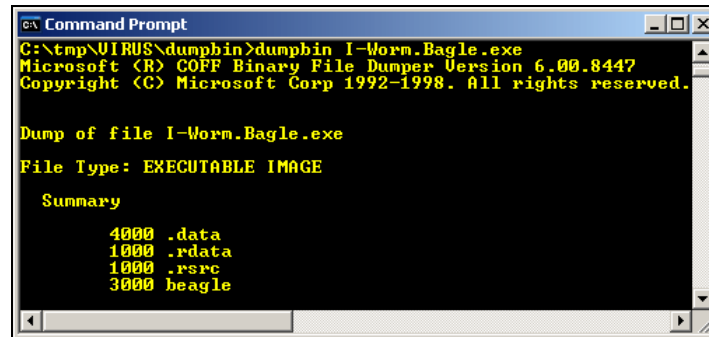
Segment	Segment Description
.text	This segment contains the executable instructions and is shared among every process running the same binary. This segment usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rdata	Sometimes known as .rodata (read-only data) segment. This contains constants and string literals.
.bss	BSS stands for "Block Started by Symbol." It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data segment) doesn't take up any actual space in the object file.
.reloc	Stores the information required for relocating the image while loading.
Heap	<p>The heap area is for dynamically allocated memory (malloc(), realloc(), calloc()) and is accessed through a pointer. Everything on a heap is anonymous, thus you can only access parts of it through a pointer. A malloc() request may be rounded up in size to some convenient power of two. Freed memory goes back to the heap, but there is no easy way to give up that memory back to the OS. The heap usually grows up toward the stack.</p> <p>The end of the heap is marked by a pointer known as the "break." You cannot reference past the break. You can, however, move the break pointer (via brk and sbrk system calls) to a new position to increase the amount of heap memory available. This is usually done automatically for you by the system if you use malloc often enough.</p>
Stack	<p>The stack holds local (automatic) variables, temporary information, function parameters, and the like. It acts like a LIFO (Last In First Out) object as it grows downward toward the heap.</p> <p>When a function is called, a stack frame (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called (and where to jump back to when the function is finished (return address)), parameters, local variables, and any other information needed by the invoked function. The order of the information varies by system and compiler, but on Solaris it is described in /usr/include/sys/frame.h. When a function returns, the stack frame is POPped from the stack. The current instruction that is running is pointed to by the IP (Instruction</p>

hexadecimal 0 to it, and then adding the offset. The 20-bit addresses were held by two 16-bit registers. In addition, the flat memory model on the x86 uses only near pointers (32 bits), while far pointers (48 bits) were needed with a segmented memory model in order to specify the segment and offset within the segment.

	Pointer). The address of the next instruction is held in the PC (Program Counter).
--	--

Table 2. Segments commonly found inside executables.

For example, Figure 1 shows the various sections of the Bagle.A virus.



```

C:\tmp\VIRUS\dumpbin>dumpbin I-Worm.Bagle.exe
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file I-Worm.Bagle.exe
File Type: EXECUTABLE IMAGE

Summary
 4000 .data
 1000 .rdata
 1000 .rsrc
 3000 beagle

```

Figure 1. Using dumpbin to show the various segments of an executable.

At this point we want to give a brief overview of how the original variant (A) of the Bagle virus works. We describe the major steps that the virus takes during its execution. It is a summary of what the virus does and how it does it. A more detailed explanation is found in Appendix A of [2].

The first thing Bagle does is initialize the COM (Component Object Model), which is needed for any non-trivial program running on the Microsoft Windows platform. COM is a platform-independent, distributed, object-oriented system for creating binary software components that can interact [30].

The very next thing it does is check that the current local date is no later than January 28, 2004. If it's after January 28, 2004, the virus exits immediately without doing any damage; otherwise it continues. This means that systems with the wrong time may still continue to be infected and help the virus spread. If the system was infected prior to January 28, 2004 and it is now after January 28, 2004, the virus will automatically kill its own process and delete its file from the

Windows system directory. However, it will not remove its Registry entries, but that is not an issue since Windows will ignore them after the virus is deleted.

It then creates a registry entry "uid" = "[Random Value]" in the registry key HKEY_CURRENT_USER\Software\Windows98. [Random Value] in this case is replaced by 8 random bytes. Following this, it initializes the Windows sockets library in order to make use of the network, and creates a mutex which will be used later to synchronize threads. It then proceeds to copy itself to the %system% (C:\WINDOWS\system32) directory and execute that copy of the virus, while killing the currently running process. If the virus is not run from %system%\bbeagle.exe, it executes calc.exe, which helps it conceal itself from user suspicion. After all, the virus has an icon of a calculator and so a user expects it to open up the Calculator program. If it is run from %system%\bbeagle.exe, it will not execute calc.exe. It also adds a new value, "d3dupdate.exe" = "%system%\bbeagle.exe" to the key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run, which restarts the virus during boot time, and the value "frun" = "1" to the registry key HKEY_CURRENT_USER\Software\Windows98, which means the virus has been successfully run on the machine for the first time.

With a new thread it creates a listening socket on port 6777, which accepts various commands and allows an attacker to upload files and execute them. This allows the attacker to update his virus with newer versions at will. The attacker can also send a specially crafted byte sequence that will force the virus to kill its own

process and delete itself from the file system. Thus, the attacker (and anyone else) has the ability to remove the virus remotely.

Another thread starts up and its purpose is to contact a list of hard coded websites every 10 minutes to inform them of the infection on the current machine. It sends the [Random Value] and port number the virus is listening on to each web site. Of course, the IP of the infected machine is logged as well.

Another thread is created and its purpose is to search all fixed drives for files that contain .wab, .txt, .htm, or .html in their filenames for valid email addresses. When an email address is found, the virus uses its own SMTP engine to send itself to the newly found email address. The source address in the email will be spoofed to try to prevent suspicion. Finally, the executing virus goes to sleep and runs every 1 second in the background. The virus has the process name `bbeagle.exe` in task manager.

One of the results that came about from the disassembly process was the discovery of the functional flow of the virus, which is presented in Appendix A.

Additionally, we assume that the reader is familiar with different types of viruses/worms. Finally, knowledge of the Win32 Portable Executable (PE) file format is necessary and this will be presented in Chapter 3 with the design details of the algorithm. For detailed information on these topics please refer to [2, 6, 7, 8].

1.5 Organization of the Thesis

The remainder of the thesis is organized into three chapters. In Chapter 2, we present related work in the fields of executable and binary analysis (static and dynamic), give an overview of the FSA-based method developed by Sekar *et. al.* in [5], and also present some related works to program slicing. In Chapter 3 we describe in detail our methodology for analyzing the binary executable, extracting and constructing the signature (FFSig), storing the signature (FFSig), and finally comparing different signatures for verification purposes. In Chapter 4, we conclude and offer ideas for future research.

Chapter 2

2 Related Work

Analysis of binaries and executables can be generally classified into two classes: *static analysis* and *dynamic analysis*. Similarly, intrusion detection techniques can be divided into the following two classes: *misuse detection* and *anomaly detection*. Much of the research done in the analysis of executables falls into the realm of dynamic analysis and anomaly detection [5, 10, 11, 12, 13]. The concept of detecting attacks by analyzing sequences of system calls is not new in the field of host-based anomaly detection systems. In fact, most Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) systems today support this technique and monitor for abnormal sequences of system calls and flag that as a possible attack.

Cohen [14] and Chess-White [15] showed that an algorithm that can detect all possible viruses cannot exist, while Landi [16] showed that static analysis can be undecidable and incomputable. However the problem considered in this thesis is different. We are not attempting to create an algorithm that will detect all possible viruses. Instead we describe an algorithm that will allow us to statically extract specific sequences of Win32 API calls from an executable and store them in an efficient manner by using the ideas presented by Sekar *et. al.* in [5]. In addition, the results of Barak *et. al.* [17] shows that obfuscation is generally impossible to

achieve in a computationally bounded environment. So, in theory, it is impossible to completely hide the malicious behavior of a virus.

Much of the research done in regards to static analysis requires that the source code of the executable be available [18, 19, 20]. This is obviously not possible with many of the executables found on today's computer systems and the Internet, especially if the executable is potentially malicious. There is some research that has been conducted in statically analyzing binary executables with no access to source code [3, 21, 22, 27, 28]. However, there are several differences between the current state of research and our proposed algorithm. In [3], the authors first unpack (if needed) the PE binary file and then extract the API calling sequence in a semi-manual manner. In fact, they use a disassembler (W32Dasm – a commercial disassembler) to disassemble the PE binary file and then run that output through a parser to get the API calling sequence. They store this new sequence by storing a sequence of 32-bit integers, which represent the various Dynamically Loaded Libraries (DLLs) and the particular API within that DLL. They store this sequence of 32-bit numbers in a vector. Our methodology looks at an unpacked executable and can automatically and efficiently extract a suspect slice and store it in a FSA utilizing the methods developed in [5].

Our methodology makes use of a splicing algorithm to extract the relevant parts of the executable. In [3], they appear to capture every single API call made by the executable, which could quickly run into the thousands, making it too big to store and too long to analyze. This splicing algorithm is described in detail in Section 3. Program slicing was originally introduced by Mark Weiser in [25]. He

defined program slicing as a method for abstracting and reducing a program to a minimal form, while still preserving its behavior. The reduced program, or “slice,” is an autonomous program independent of the original, but is guaranteed to have the same behavior as the original within the domain of the specified subset of behavior. His approach for decomposing a program consisted of analyzing its control and data flow (via the program’s source code). Weiser further states that the behavior of interest can be specified as values of specific sets of variables at some set of statements and he calls this the slice criterion. This algorithm is further described in the following section. Static program slicing of binary executables is described by Kiss *et. al.* in [27] and Cifuentes *et. al.* in [28]. We further describe their methods in section 3 and show how it can be applied to our methodology with detailed examples.

Finally, our methodology also makes use of similarity measures. Extensive research in this field has been done by many including Nambiar *et. al.* [34], Noreault *et. al.* [35], and Cohen [36]. We make use of efficient techniques that are able to compare finite state automata and graphs, based on the research conducted by D. Zhang *et. al.* [39] and L.P. Cordella *et. al.* [40]. We also present some other techniques that can be used to measure similarity between sequences. They include Euclidean distance, cosine measures, the Jaccard measures, and the Pearson correlation measure.

Chapter 3

3 Algorithm and Architecture Design

In this section, we present our methodology for efficiently and statically analyzing an executable to construct (extract) and store a signature based on the sequence of Win32 API calls made by the executable, followed by a slew of similarity measures to compare known and unknown signatures. Our approach consists of 4 major steps. The first step is the disassembly of the target executable, which yields an assembly version of the code. The first step also includes the analysis of the assembly code to simplify it and improve the analyzability of it. This results in abstractions and high-level representations of the assembly code. The second step is to extract certain malicious parts of the code in order to analyze it more closely. This step reduces and focuses the amount of code that has to be analyzed (in effect it decreases the complexity of the detection process). This is accomplished by using slicing techniques on certain areas of the assembly code. The third is to store the information from the slice using a finite state automaton (this becomes the signature). The fourth and final step is to use various similarity measures to compare different signatures and then produce a report. The overall architecture of our methodology is reported in Figure 2.

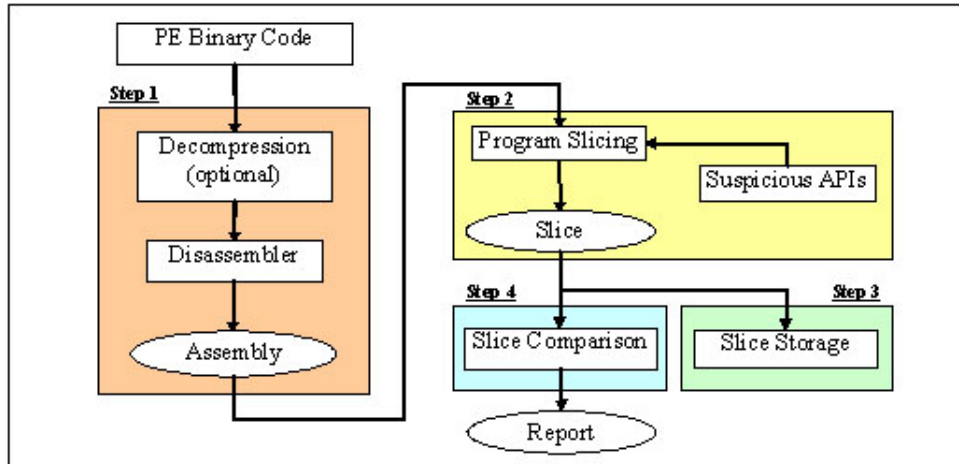


Figure 2. Overview of the system architecture.

3.1 Disassembly and Analysis of Executable

A binary executable is stored as a sequence of bytes. To be able to analyze the control flow of the program, the sequence of bytes has to be processed. As we will see, trying to detect instruction boundaries from binary code is not a trivial task. On architectures with variable length instructions the boundaries may not be detected unambiguously. On architectures with multiple instruction sets it may be difficult to determine the instruction set used. If the binary representation mixes code and data their separation may be also difficult [27]. Once the instruction boundaries have been determined, the control flow of the program still has to be determined. This is more troublesome than it seems at first. The behavior of the instructions have to be established and since instructions at the binary level are so much more numerous (as compared to the source-code equivalent) and jumps and other control transfer instructions are sometimes ambiguous in where they transfer

control, it requires more than simply scanning and tokenizing the binary. Another important problem that needs to be addressed is the determination of the function boundaries. Function call sites have to be detected and the targets of the function calls need to be determined. The detection of function boundaries is not an easy task in general, but indirect function calls, where the target of the call cannot be determined unambiguously, and overlapping and cross-jumping functions (where the control flow can cross function boundaries) present further problems [27]. These problems are mitigated slightly by auxiliary data stored within the executable image itself. In the PE format, for example, there is a PE header which contains supplementary data that helps the operating system load and execute the executable image. This information includes section tables, individual section data, sizes, offsets, relocation data, symbol information, etc. Needless to say, this information is highly dependent on the specific format of the executable image and the operating system itself, so generalizing the extraction process of this information is not possible. Below, we go through the process of dissecting a PE executable.

Once the executable has been acquired, we can begin the analysis. We first check if the executable is packed (compressed). This is done by reading the section names in the section table of the PE header. For example, if the section name is UPX0, UPX1, and so on, then we know that the PE file has been packed with the UPX algorithm. We can then manually unpack it. For example, Variant B of the Bagle virus was compressed with UPX as shown in Code 1.

Section Table			
01 UPX0	VirtSize: 00009000	VirtAddr: 00001000	
raw data offs:	00000400	raw data size:	00000000
relocation offs:	00000000	relocations:	00000000
line # offs:	00000000	line #'s:	00000000
characteristics:	E0000080		
	UNINITIALIZED_DATA	EXECUTE	READ WRITE ALIGN_DEFAULT(16)

02 UPX1	VirtSize: 00002000	VirtAddr: 0000A000
raw data offs:	00000400	raw data size: 00001A00
relocation offs:	00000000	relocations: 00000000
line # offs:	00000000	line #'s: 00000000
characteristics:	E0000040	
INITIALIZED_DATA EXECUTE READ WRITE ALIGN_DEFAULT(16)		

Code 1. The section table of Bagle.B virus showing UPX compression is used.

We then unpack it manually using the UPX program [29] as shown in Figure 3.

```

C:\dell\ida\b>..\upx\upx.exe -d -o I-Worm.Bagle.b.decompressed.exe I-Worm.Bagle.b.exe
Ultimate Packer for executables
Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004
UPX 1.25v Markus F.X.J. Oberhumer & Laszlo Molnar Jun 29th 2004

  File size      Ratio      Format      Name
  -----
  16896 <-      11264      66.67%     win32/pe     I-Worm.Bagle.b.decompressed.exe

Unpacked 1 file.
C:\dell\ida\b>

```

Figure 3. Unpacking an executable using UPX.

Once the executable has been unpacked, we can begin to process the PE file. Every PE file begins with a small DOS stub program (a.k.a. MS-DOS header), whose sole purpose is to print out an error message that says that Windows is required to run this executable (the error message is “This program cannot be run in DOS mode.”) Within this MS-DOS header, there is field called `e_lfanew`, which contains the offset of the PE header. The MS-DOS header is defined in `winnt.h`, as shown in Code 2.

```

typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD    e_magic;           // Magic number
    WORD    e_cblp;           // Bytes on last page of file
    WORD    e_cp;             // Pages in file
    WORD    e_crlc;           // Relocations
    WORD    e_cparhdr;        // Size of header in paragraphs
    WORD    e_minalloc;       // Minimum extra paragraphs needed
    WORD    e_maxalloc;       // Maximum extra paragraphs needed
    WORD    e_ss;             // Initial (relative) SS value
    WORD    e_sp;             // Initial SP value
    WORD    e_csum;           // Checksum
    WORD    e_ip;             // Initial IP value
    WORD    e_cs;             // Initial (relative) CS value
    WORD    e_lfarlc;         // File address of relocation table
    WORD    e_ovno;           // Overlay number
    WORD    e_res[4];         // Reserved words
    WORD    e_oemid;          // OEM identifier (for e_oeminfo)
    WORD    e_oeminfo;        // OEM information; e_oemid specific
    WORD    e_res2[10];       // Reserved words
    LONG    e_lfanew;         // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

Code 2. The MS-DOS header defined inside `winnt.h`.

So we need to find this value within the MS-DOS header, as shown in Code 3.

```

// MakePtr is a macro that allows you to easily add to values
// (including pointers) together without dealing with C's pointer
// arithmetic. It essentially treats the last two parameters as
// DWORDs. The first parameter is used to typecast the result to
// the appropriate pointer type.
#define MakePtr(cast, ptr, addValue) (cast)((DWORD_PTR)(ptr) +
(DWORD_PTR)(addValue))

// IMAGE_NT_SIGNATURE and IMAGE_DOS_SIGNATURE are defined in
// winnt.h
#define IMAGE_NT_SIGNATURE 0x00004550 // PE00
#define IMAGE_DOS_SIGNATURE 0x5A4D // MZ

// declare a pointer to the DOS Header and fill in its data via
// g_pMappedFileBase, which is a mapped view of the opened
// executable.
PIMAGE_DOS_HEADER dosHeader;
dosHeader = (PIMAGE_DOS_HEADER)g_pMappedFileBase;

// check the for the MZ signature
if ( dosHeader->e_magic == IMAGE_DOS_SIGNATURE )
{
    // Make pointers to the 32 bit version of the header.
    pNTHHeader = MakePtr(PIMAGE_NT_HEADERS, dosHeader, dosHeader->e_lfanew);

    // check the signature to make sure its PE00
    if (pNTHHeader->Signature != IMAGE_NT_SIGNATURE)
    {
        printf("Not a Portable Executable (PE) EXE\n");
        return;
    }
}
}

```

Code 3. Checking for the MS-DOS and PE signatures.

The important thing to remember here is that the data structures of a PE file on disk are the same as the data structures used in memory. So if you know how to find something inside the PE file on disk, you will be able to find it when it is loaded in memory. The next step is to find the Relative Virtual Address (RVA) of the entry point within the PE Header (`pNTHHeader`). This is the address at which the code section begins (and where the executable begins to execute). Within a PE file, there are many places where memory addresses need to be specified (i.e. the address of a global variable). Although PE files have a preferred loading address (this is known as the `ImageBase` and the default for Windows executables is `0x00400000`), the operating system can load them anywhere in memory. This means that a PE file needs to have a way of specifying addresses within its image

independently of where the file is actually loaded. This is achieved via RVAs. Instead of hard-coded values, the PE file uses RVAs, which are just offsets from the address where the image was loaded (known as HMODULE in Windows-speak). The RVA of the entry point of the image is specified within the structure called IMAGE_OPTIONAL_HEADER. It is defined inside winnt.h, as shown in Code 4.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Code 4. The address of the entry point into an executable is defined.

The code to extract the RVA of the entry point can be as follows:

```
printf("%X\n", &pNtHeader->OptionalHeader->AddressOfEntryPoint);
```

This memory address is the address when the image is loaded into memory by the Windows loader. To find the corresponding address inside the file, when it is on the hard disk, we have to subtract the base address of the code section from this RVA and then add the raw offset of the code section. Once again, this value can be gotten from the structure called `IMAGE_OPTIONAL_HEADER`. This will give us the address of the entry point within the file when it is still on the hard disk (see Code 5).

```
PIMAGE_SECTION_HEADER section;
unsigned cSections;
DWORD rawOffset;

// initialize the sections and the number of sections
section = IMAGE_FIRST_SECTION(pNTHHeader);
cSections = pNTHHeader->FileHeader.NumberOfSections;

// go through the sections, finding those that are executable
for (unsigned i=1; i <= cSections; i++, section++)
{
    if (section->Characteristics == (IMAGE_SCN_CNT_CODE + IMAGE_SCN_MEM_EXECUTE
+ IMAGE_SCN_MEM_READ))
    {
        // if it's execute,read, and contains code then use
        // its raw offset
        rawOffset = section->PointerToRawData;
        break;
    }
}
printf("raw base of code: %X\n", rawOffset);
printf("raw entrypoint: %X\n", ((&pNTHHeader->OptionalHeader)->AddressOfEntryPoint -
(&pNTHHeader->OptionalHeader)->BaseOfCode + rawOffset));
```

Code 5. Calculating the address of the entry point.

Next we want to read in the Import Address Table (IAT). The IAT holds an array function pointers which point to all API functions that are imported from various DLLs. Each PE file contains an IAT and each imported API has its own reserved spot within the IAT. The IATs for each imported DLL appear sequentially in memory. Inside the PE file, there is only one place where an imported API's address is stored and that's in the IAT. Let's look at what a call to an imported API looks like. There are two possibilities here. We can call through a function pointer, which is a very efficient way of calling a Win32 API:

```
CALL DWORD PTR [0x004031E8]
```

Or we can call an API by transferring control to a small stub. The stub is simply a JMP to the address whose value is at 0x004040A4. This address is within the IAT. This is less efficient because it uses 5 more bytes of code and an extra JMP and thus takes longer. For example, let's look at some actual binary code of the Bagle.A virus:

```
0040318C: E8 95 01 00 00    call    00403326
00403326: FF 25 C4 40 40 00 jmp     dword ptr ds:[004040C4h]
```

The above representation is how it would look in memory. The memory address 0x00403326 is gotten by adding 0x0195 to 0040318c + 5 bytes (the length of this code). To find the same code within the file on disk, we would do the following. Whenever we run into a CALL instruction (E8), we would look at the next 4 bytes and add the value of those 4 bytes to the current location inside the PE file (and add 5 bytes to compensate for the CALL. For example:

```
00002180: 00 33 c0 5b 5f 5e c9 c2 04 00 6a 00 e8 95 01 00 .3.[_^....j.....
00002190: 00 e8 9f e6 ff ff 83 3d 03 50 40 00 00 75 14 68 .....=.P@..u.h
```

Above the CALL instruction is made from raw address 218C. 218C + 5 + 0195 gives us 2326. Let's look at what's at the raw address 2326 inside the PE file (highlighted in red):

```
00002320: ff 25 2c 41 40 00 ff 25 c4 40 40 00 ff 25 c8 40 .%,A@..%.@..%.@
```

It's the same JMP instruction. Looking it up inside [23], we see that this instruction is a jump to an absolute, indirect address, exactly as expected (since it's a pointer to a function). Now the question becomes which API is at the address

0x004040C4? To answer this question, let's first look at what IDA Pro [24] (a commercial disassembler) tell us:

```

beagle:0040318A      public start
beagle:0040318A start      proc near
beagle:0040318A      push    0      ; pvReserved
beagle:0040318C      call   CoInitialize

```

IDA Pro has determined that the API at address 0x004040C4 is CoInitialize. Looking at what the PEDUMP (described in detail in [7, 8]) tells us about the imported API, we can see that the Import Address Table (IAT) RVA starts at address 0x000040C4 (see Code 6). At this address is a pointer to the beginning of the IAT and the first thunk (in this case CoInitialize).

```

ole32.dll
Import Lookup Table RVA: 000042C4
TimeStamp: 00000000
ForwarderChain: 00000000
DLL Name RVA: 0000469E
Import Address Table RVA: 000040C4
Ordin Name
 49 CoInitialize
107 CreateStreamOnHGlobal

```

Code 6. Output form PEDUMP.EXE showing imported functions.

The anchor of the imports data is the `IMAGE_IMPORT_DESCRIPTOR` structure. The `DataDirectory` entry for imports points to an array of these structures. There's one `IMAGE_IMPORT_DESCRIPTOR` for each imported executable. Each `IMAGE_IMPORT_DESCRIPTOR` typically points to two essentially identical arrays. These arrays have been called by several names, but the two most common names are the Import Address Table (IAT) and the Import Name Table (INT). Figure 4 shows an executable importing some APIs from `USER32.DLL`. [8]

Both arrays have elements of type `IMAGE_THUNK_DATA`, which is a pointer-sized union. Each `IMAGE_THUNK_DATA` element corresponds to one

imported function from the executable. The ends of both arrays are indicated by an `IMAGE_THUNK_DATA` element with a value of zero. In the executable file, they contain either the ordinal of the imported API or an RVA to an `IMAGE_IMPORT_BY_NAME` structure. The `IMAGE_IMPORT_BY_NAME` structure is just a `WORD`, followed by a string naming the imported API. [8]

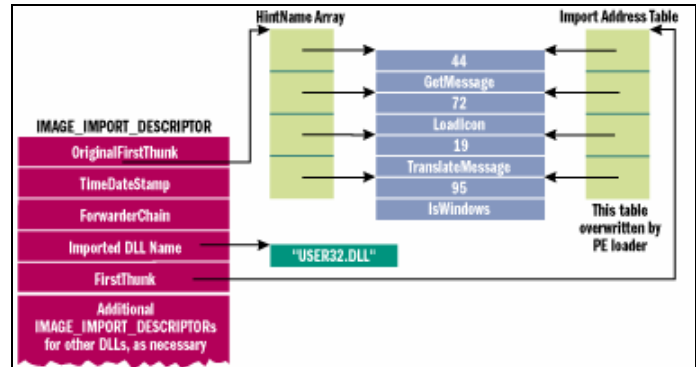


Figure 4. Two parallel arrays of pointers.

Now that we know where the starting point is, we have to move to that position and begin reading and processing (disassembling) the executable. This is a simplified view of how disassemblers work. They begin to read to the executable one byte at a time and determine which instruction the byte is and how many operands it takes (how many of the following bytes belong to the current instruction). We can expand our method of statically reading and analyzing the binary executable to fully disassembling the entire executable. Instead, we can use tools like IDA Pro [24] to disassemble to the executable quickly.

² This figure taken from [8].

3.2 Slicing Algorithm to Extract the Signature

Our methodology makes use of a slicing algorithm to extract the relevant parts of a malicious binary executable that can be used for the signature. Cifuentes *et. al.* in [28] correctly suggest that the ability to slice binary executables could aid in the analysis of worms and viruses because once a critical area of code has been flagged (i.e. the code that searches for emails on the hard drive), it can be sliced out and significantly simplify the analysis. We will begin by introducing the idea of program slicing as it was described by Mark Weiser in [25]. This will be followed by an explanation of the techniques used to slice binary executables followed by an application of the techniques to our research.

Program slicing was originally introduced by Mark Weiser in [25]. He defined program slicing as a method for abstracting and reducing a program to a minimal and simple form, while still preserving its behavior. The reduced program, or “slice,” is an autonomous program independent of the original, but is guaranteed to have the same behavior as the original within the domain of the specified subset of behavior. Weiser’s approach for decomposing a program consisted of analyzing its control and data flow (via the program’s source code). Weiser further states that the behavior of interest can be specified as values of specific sets of variables at some set of statements and he calls this the slice criterion. Figure 5 (compiled from images in [25]) gives examples of some slicing criterion and their corresponding slices.

<pre> The original program: 1 BEGIN 2 READ(X,Y) 3 TOTAL := 0.0 4 SUM := 0.0 5 IF X <= 1 6 THEN SUM := Y 7 ELSE BEGIN 8 READ(Z) 9 TOTAL := X*Y 10 END 11 WRITE(TOTAL,SUM) 12 END. </pre>	<pre> Slice on the value of Z at statement 12. BEGIN READ(X,Y) IF X < 1 THEN ELSE READ(Z) END. Slice on the value of X at statement 9. BEGIN READ(X,Y) END. Slice on the value of TOTAL at statement 12. BEGIN READ(X,Y) TOTAL := 0 IF X <= 1 THEN ELSE TOTAL := X*Y END. </pre>
---	--

Figure 5. The original program (left), and some example slices of the program.

Weiser's algorithm works on block-structured Pascal-like source code where variables are uniquely named and all procedures are assumed to be single-entry, single-exit. The slicing criterion for such a program can be thought of as a window for observing its behavior and this window is specified as a statement and the values of a set of variables (as shown in Figure 5). The slicing criterion can be specified as a pair $\langle i, v \rangle$, where i is the number of the statement at which to observe, and v is the set of variables to observe. Weiser further states that each slice has the following two properties: the slice must be obtained from the original program via statement deletion and that the behavior of the slice must be the same as the original program when the slice criterion is applied. Because statement deletion will often make a program ungrammatical, Weiser makes use of flowgraph³ to represent the program, where each node will represent a single

³ From [25]: A flowgraph is a structure $G = \langle N, E, n_0 \rangle$, where N is the set of nodes, E is a set of edges in $N \times N$, and n_0 is the distinguished initial node. If (n, m) is an edge in E then n is an immediate predecessor of m , and m is an immediate successor of n . A path of length k from n to m is a set of nodes $p(0), p(1), \dots, p(k)$ such that $p(0) = n$, $p(k) = m$, and $(p(i), p(i+1))$ is in E for all $i, 0 < i < k-1$. There is a path from n_0 to every other node in N . A node n is nearer than a node m to some node q if the shortest path from n to q has length less than the shortest path from m to q . A node m is dominated by a node n if n is on every path from n_0 to m . An inverse dominator is a dominator on the flowgraph obtained by reversing the direction of all edges and making the final node the initial node.

source language statement. Deleting nodes from a flowgraph produces a new, yet meaningful flowgraph as long as the nodes that were deleted have only one successor. The deleted node's successors become the successors of the predecessors of the deleted nodes, as shown in Figure 6 (taken from [25]).

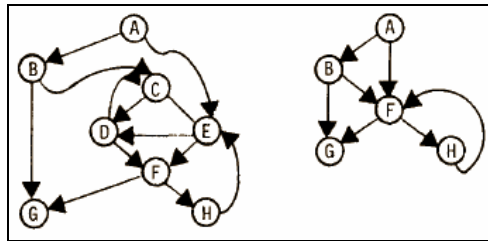


Figure 6. A group of statements with a single successor. Nodes C, D, and E form a set with a single successor, F, not in the set. The flowgraph is shown before (left) and after (right) removing this set.

The second property of slices is that they exhibit the equivalent behavior as the original program with the slicing criterion applied. To be more precise, equivalent behavior is defined as behavior that is equivalent when the original program terminates. The slicing criterion has the form $\langle i, v \rangle$ and v can be used in both the slice and the original program. However, i , may not exist in the slice, so $\langle \text{SUCC}(i) \rangle$ is used instead. $\langle \text{SUCC}(i) \rangle$ is the nearest successor to i and it exists in the original program as well as the slice. To find slices using dataflow analysis requires one to know all the possible statements and variables that can affect the variables being observed through the window of the slicing criterion. This is known as reachability analysis. Further information on this dataflow analysis algorithm can be found in [25].

Static program slicing of binary executables is described by Kiss *et. al.* in [27] and Cifuentes *et. al.* in [28]. In the next few pages we further describe their methods and show how it can be applied to our methodology. We start with a

description of the research presented in [28] which is based on the Intel 80286 CISC chipset (the techniques can be applied to the Intel Pentium chipset, but would take longer since it has more instructions). Cifuentes *et. al.* focus on intraprocedural slicing, which means that they look at calls made within a single procedure (although they make use of some of the ideas presented in [31]). Cifuentes *et. al.* stipulate that when faced with an indexed JMP instruction or an indirect CALL or JMP instruction on the value of a register, statically it is not possible to determine the value in the register and thus it is not possible to determine the target address. For example, in Figure 7 (taken from [28]), the indexed JMP at instruction 71 depends on the contents of the BX register. Statically, we do not know the value in the BX register. However, we can use backward splicing on the BX register at instruction 71 to figure out the possible ranges of values in the BX register.

```

[...]
038 L6: XOR      ah, ah
039     MOV      dx, ax
040     MOV      bx, ax
041     SUB      b1, 20h
042     CMP      b1, 60h
043     JAE      L7
044     MOV      b1, [bx+291h]
045     CMP      bx, 17h
046     JBE      L8
047     JMP      L9
048 L9: MOV      si, [bp-10h]
049     MOV      di, [bp-4]
050     MOV      al, 25h
[...]
070 L8: SHL      bx, 1
071     JMP      word ptr cs:[bx+9B3h]
[...]
```

Figure 7. Extract assembly code with an indexed jump statement.

The algorithm is divided into three steps:

1. Determine the slice using the algorithm presented by Horwitz *et. al.* in [31].
2. Add unconditional jumps and returns to the slice.
3. Fix jump labels.

The first step is described here. Program slicing as presented by Weiser in [25] uses individual statements as the nodes in the control flow graph (CFG) (See Figure 10). The corresponding machine code or assembly instructions are too numerous and thus basic blocks of them are used as the nodes in the CFG. The construction of the control dependence graph (CDG) is based on the post dominator tree (PDT) and the CFG. A CDG is a graph that represents control dependencies in a graph. For example, if statement X determines whether statement Y is executed, then statement Y is control dependent on statement X . Statements that are guaranteed to execute are said to be dependent on program entry. An example of a CDG is shown in Figure 10. A PDT is also sometimes called a forward dominance tree (FDT) and represents the forward dominance of various statements within the program. For example, if all paths from statement X include statement Y , then statement Y forward dominates statement X . An example of a PDT is shown in Figure 10. The PDT can be constructed in $O(N\alpha(N))$ time, where N is the number of nodes in the CFG. The CDG can be built in N^2 time by walking the PDT [28].

Data dependencies are represented in terms of UD-chains (Use-Definition chains) as is shown in Figure 8 (taken from [28]), which also shows the disassembly of the test C program shown in Figure 9 (taken from [28]). Prior to creating the UD-chains, Cifuentes *et. al.* analyze the code and replace various instructions with different, yet equivalent instructions in order to reduce unnecessary dependencies. For example, the `xor si, si` instruction is replaced with `mov si, 0`. The UD-chains are then generated for each register and condition code used in an instruction.

```

main PROC NEAR
000 PUSH bp ; ud(bp)=-1
001 MOV bp, sp ; ud(sp)=-1
002 SUB sp, 2 ; ud(sp)=-1
003 PUSH si ; ud(si)=-1
004 PUSH di ; ud(di)=-1
005 XOR si, si
006 MOV word ptr [bp-2], 0
007 MOV di, 0FFFBh
008 JMP L1
009 L1: CMP di, 0Ah ; ud(di)=7
010 JL L2 ; ud(cc)=9
011 PUSH si ; ud(si)=5,32,50,56
012 MOV ax, 194h
013 PUSH ax ; ud(ax)=12
014 CALL near ptr printf
015 POP cx
016 POP cx
017 PUSH word ptr [bp-2]
018 MOV ax, 19Eh
019 PUSH ax ; ud(ax)=18
020 CALL near ptr printf
021 POP cx
022 POP cx
023 POP di
024 POP si
025 MOV sp, bp ; ud(bp)=1
026 POP bp
027 RET
028 L2: OR di, di ; ud(di)=7,34
029 JG L3 ; ud(cc)=28
030 MOV ax, si ; ud(si)=32,50,56,5
031 ADD ax, di ; ud(ax)=30 ud(di)=7,34
032 MOV si, ax ; ud(ax)=31
033 JMP L4
034 L4: INC di ; ud(di)=7,34
035 JMP L1
036 L3: MOV ax, [bp-2] ; ud([bp-2])=6,38
037 INC ax ; ud(ax)=36
038 MOV [bp-2], ax ; ud(ax)=37
039 MOV ax, di ; ud(di)=7,34
040 MOV bx, 2
041 CWD ; ud(ax)=39
042 MOV tmp, dx:ax ; ud(dx:ax)=41
043 IDIV bx ; ud(bx)=40 ud(tmp)=42
044 MOD bx ; ud(bx)=40 ud(tmp)=42
045 OR dx, dx ; ud(dx)=43
046 JNE L5 ; ud(cc)=45
047 MOV ax, di ; ud(di)=7,34
048 SHL ax, 1 ; ud(ax)=47
049 ADD ax, si ; ud(si)=5,32,50,56 ud(ax)=48
050 MOV si, ax ; ud(ax)=49
051 JMP L4
052 L5: MOV ax, di ; ud(di)=7,34
053 MOV dx, 3
054 MUL dx ; ud(dx)=53 ud(ax)=52
055 ADD ax, si ; ud(si)=32,50,56 ud(ax)=54
056 MOV si, ax ; ud(ax)=55
057 JMP L4
main ENDP

```

Figure 8. Disassembled code for `main()` procedure of the test program in Figure 9. It is annotated with Use-Definition chains.

```

void main()
{ int sum, positives, x;
  sum = 0;
  positives = 0;
  for (x = -5; x < 10; x++) {
    if (x <= 0)
      sum = sum + x;
    else {
      positives = positives + 1;
      if (x % 2 == 0)
        sum = sum + 2 * x;
      else
        sum = sum + 3 * x;
    }
  }
  printf ("sum = %d\n", sum);
  printf ("positives = %d\n", positives);
}

```

Figure 9. Sample Test C program.

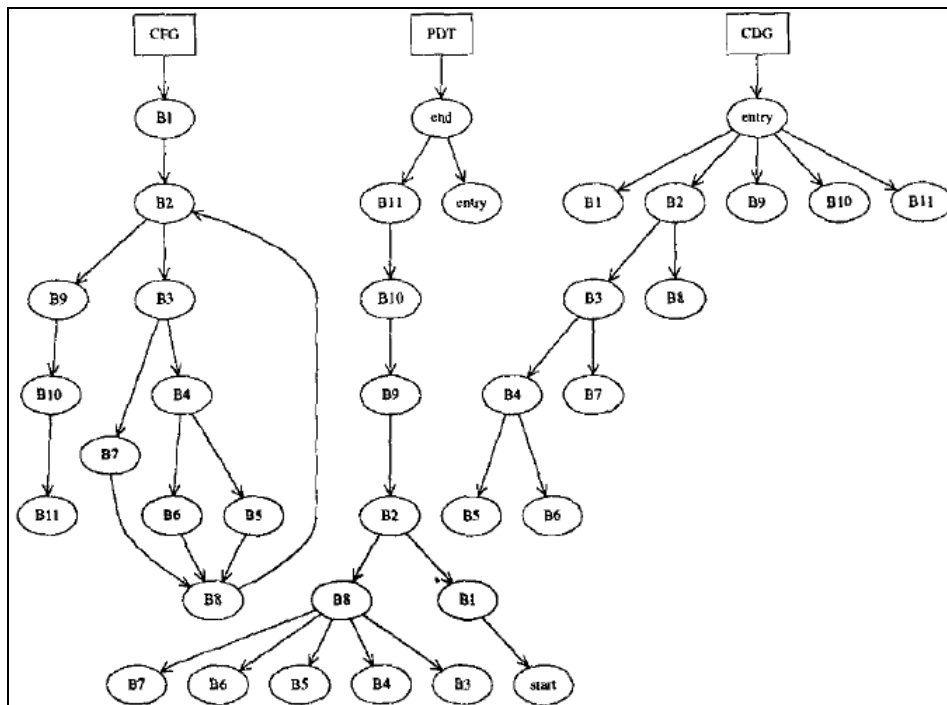


Figure 10. Control Flow Graph, Program Dependence Tree, and Control Dependence Graph for the program in Figure 8.

In order to accommodate conditional jumps (those jumps that are not dependant on registers (i.e. `jne`, `jl`, etc.)), UD-chains on conditional codes (denoted by `cc`) need to be used. For example, in the following code:

```

cmp dx, bx ; dF = CCF,ZF,SF)
jg L10    ; uF = (SF)

```

the compare instruction `cmp dx, bx` sets the condition codes CF (carry flag), ZF (zero flag) and SF (sign flag). The conditional jump instruction `jg L10` uses the sign flag to determine whether the jump is to be taken or not. Dead-condition code elimination would determine that the zero and carry flag set by the compare instruction are dead and therefore irrelevant to the analysis; however, the sign flag is set by this instruction and then used in the conditional jump, hence making the conditional jump data dependent on the comparison instruction [28].

```

main PROC NEAR
005     XOR     si, si
007     MOV     di, 0FFFBh
009 L1:  CMP     di, 0Ah
010     JL      L2
011     Start:
028 L2:  OR      di, di
029     JG      L4
030     MOV     ax, si
031     ADD     ax, di
032     MOV     si, ax
034 L3:  INC     di
035     JMP     L1
036 L4:
039     MOV     ax, di
040     MOV     bx, 2
041     CWD
042     MOV     tmp, dx:ax
044     MOD     bx
045     OR      dx, dx
046     JNE     L5
047     MOV     ax, di
048     SHL     ax, 1
049     ADD     ax, si
050     MOV     si, ax
051     JMP     L3
052 L5:  MOV     ax, di
053     MOV     dx, 3
054     MUL     dx
055     ADD     ax, si
056     MOV     si, ax
057     JMP     L3
main ENDP

```

Figure 11. Slice of program given in Figure 8 w.r.t. Register `si` at Instruction 11.

The second step is to add unconditional jumps and return instructions. In typical program slicing, the lexical successor tree (LST) is used to determine the next high-level statement to go to (i.e. the lexical successor of the last instruction). However, in binary code, the same thing can be represented by many different assembly instructions. Here it is safe to assume that the end of a procedure is not at

the end (i.e. the next physical instruction), but is interleaved within the code somewhere else. Unconditional jumps and return instructions introduce this break in the flow of control of the program. The time required for this step is linear to the number of basic blocks. The last step is used to fix the target labels. This is done by checking all of the jumps that are part of the slice. As an example, consider Figure 11 (taken from [28]), which shows the slice obtained from the test program in Figure 8 when the slice criterion is $\langle 11, si \rangle$.

The next logical step in slicing binary executables would be to look at how a way to create an interprocedural slice. That is, generating a slice of the entire program where the slice spans different procedures (i.e. crosses the boundaries of procedure calls). Kiss *et. al.* in [27] describe a method for interprocedural static slicing of binary executables. Through their algorithm, they claim they were able to produce slices that were 56-68% of the original program size.

Kiss *et. al.* present a list of problems that exist when trying to slice binary executables as opposed to slicing structured source-code, as in [25]. A binary executable is stored as a sequence of bytes. To be able to analyze the control flow of the program, the sequence of bytes has to be processed. As we have seen earlier, trying to detect instruction boundaries from binary code is not a trivial task. On architectures with variable length instructions the boundaries may not be detected unambiguously. On architectures with multiple instruction sets it may be difficult to determine the instruction set used. If the binary representation mixes code and data their separation may be also difficult [27]. Once the instruction boundaries have been determined, the control flow of the program still has to be determined.

This is more troublesome than it seems at first. The behavior of the instructions have to be established and since instructions at the binary level are so much more numerous (as compared to the source-code equivalent) and jumps and other control transfer instructions are sometimes ambiguous in where they transfer control, it requires more than simply scanning and tokenizing the binary. Another important problem that needs to be addressed is the determination of the function boundaries. Function call sites have to be detected and the targets of the function calls need to be determined. The detection of function boundaries is not an easy task in general, but indirect function calls, where the target of the call cannot be determined unambiguously, and overlapping and cross-jumping functions (where the control flow can cross function boundaries) present further problems [27]. These problems are mitigated slightly by auxiliary data stored within the executable image itself. In the PE format, for example, there is a PE header which contains supplementary data that helps the operating system load and execute the executable image. This information includes section tables, individual section data, sizes, offsets, relocation data, symbol information, etc. Needless to say, this information is highly dependent on the specific format of the executable image and the operating system itself, so generalizing the extraction process of this information is not possible.

Once these problems have been overcome, *basic block leaders* are generated. *Leaders* can be defined as instructions following branching or call instructions (JMP, CALL, etc.), the targets of these instructions, the first instructions of a function, and instructions following set switch instructions.

2. Perform a control and data dependence analysis on the CFG, which results in a program dependence graph (PDG).
3. Compute the intraprocedural and interprocedural slices.

Since the first step has just been explained in detail, we proceed to the second step, whose goal is to build the PDG. The PDG is composed of the CDG, which in this case represents the different control dependencies between the basic blocks of a function. The CDG is further explained above. Kiss *et. al.* construct the CDG with the PDT and the CFG using the algorithms described in [32] and [33].

The other part of the PDG is the data dependence graph (DDG), which represents the dependence among instructions according to their formal and actual parameters. See Figure 13. In order to create the DDG, every instruction is analyzed to see which registers, flags, and memory addresses⁴ it uses. The analysis results in the sets u_j and d_j for each instruction j , which contain all used and defined arguments of j , respectively. During the analysis we also determine the sets $u_j^{(a)}$ for every $a \in d_j$, which contain the arguments of j actually used to compute the value of a . Obviously $u_j = \bigcup_{a \in d_j} u_j^{(a)}$ for each instruction j , but instructions may exist where $u_j^{(a)} \subset u_j$ for a defined argument a . Unlike that in high-level programs, in binaries the parameter list of procedures is not defined explicitly but has to be determined via a suitable interprocedural analysis. We use a fix-point iteration to collect the sets of *input* and *output parameters* of each function. We compute the sets U_f and D_f representing the used and defined arguments of all instructions in function f itself and in functions called (transitively) from f . I_f is the set of instructions in f and C_f is the set of functions called from f . The resulting set D_f is

⁴ The algorithm only determines if the instruction reads from or writes to memory.

called the set of output parameters of function f , while $U_f \cup D_f$ yields the set of input parameters of f [27].

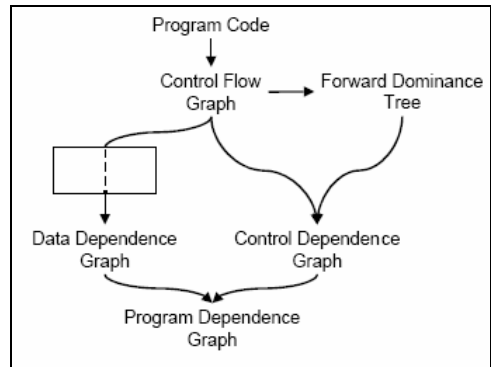


Figure 13. A graph-based model.

Finally, the CDG is extended to create the DDG. Nodes that represent the instructions of the program, the used and defined arguments of each instruction, the parameters of the called function, and the formal input (formal-in) and output (formal-out) parameters are added to the graph.

Once the individual PDGs for each individual function have been constructed, intraprocedural and interprocedural slices can be computed. Here we concentrate on computing interprocedural slices. The individual PDGs have to be interconnected and this is achieved by connecting all actual-in and actual-out parameter nodes with the appropriate formal-in and formal-out nodes using parameter-in and parameter-out edges. Actual-in and actual-out parameters are nodes created for all input and output parameters of the called function, respectively (in effect, each function call in a PDG is connected with the corresponding PDG of the called function). The resulting graph is now called a system dependence graph (SDG). The SDG is further augmented with summary

⁵ Image taken from slides by Judith A. Stafford (University of Colorado at Boulder).

edges to represent dependencies between actual-in and actual-out parameters. For more details, refer to [27].

At this point, let's take a look at actual Bagle virus assembly code. This assembly code was derived from version A of the Bagle virus by using the commercial disassembler IDA Pro. Here we will look at the part of the code that is responsible for searching all hard drives for email addresses. According to the system architecture, we slice the target executable based on suspicious APIs. It is assumed that this database of suspicious APIs has been previously compiled and has been kept up to date. Examples of suspicious APIs that we will use in the following pages include `InternetOpenUrlA`, `FindFirstFileA`, and `FindNextFileA`.

The first suspicious API we will look at is `InternetOpenUrlA`. This API function is imported `wininet.dll` and is used to open a resource specified by the URL (typically an HTTP URL). Code 7 contains the assembly code for the subroutine from where `InternetOpenUrlA` is called and the slice with respect to variable `hMem` at location `00402D97` (in gray).

```

beagle:00402D3D
beagle:00402D3D ; ||| S U B R O U T I N E |||
beagle:00402D3D
beagle:00402D3D ; Attributes: bp-based frame
beagle:00402D3D
beagle:00402D3D sub_402D3D      proc near          ; CODE XREF: sub_402DC2+1F p
beagle:00402D3D
beagle:00402D3D hMem          = dword ptr -8
beagle:00402D3D var_4         = dword ptr -4
beagle:00402D3D arg_0        = dword ptr  8
beagle:00402D3D
beagle:00402D3D      push     ebp
beagle:00402D3E      mov      ebp, esp
beagle:00402D40      add     esp, 0FFFFFF94h
beagle:00402D43      push     ebx
beagle:00402D44      push     esi
beagle:00402D45      push     400h          ; dwBytes
beagle:00402D4A      push     40h          ; uFlags
beagle:00402D4C      call    GlobalAlloc
beagle:00402D51      mov     [ebp+hMem], eax ; result of GlobalAlloc moved into
                                     ; hMem

```

```

beagle:00402D54    push    offset Data      ; registry_value of uid (38174321)
beagle:00402D59    push    dword_405003     ; port_number = 1A79h = 6,777
                                ; (hardcoded elsewhere in the body)
beagle:00402D5F    push    [ebp+arg_0]      ; argument passed insite =
                                ; http://www.elrasshop.de/1.php
beagle:00402D62    push    offset aS?pLuIdS ; "%s?p=%lu&id=%s"
beagle:00402D67    push    [ebp+hMem]       ; buffer just allocated with
                                ; GlobalAlloc()

beagle:00402D6A    call   wsprintfA
beagle:00402D6F    add    esp, 14h
beagle:00402D72    call   sub_402D22
beagle:00402D77    push    0
beagle:00402D79    push    0
beagle:00402D7B    push    0
beagle:00402D7D    push    1
beagle:00402D7F    push    offset aBeagle_beagle ; "beagle_beagle"
beagle:00402D84    call   InternetOpenA
beagle:00402D89    mov    [ebp+var_4], eax
beagle:00402D8C    push    0                ; context
beagle:00402D8E    push    40000000h        ; flag = INTERNET_FLAG_RAW_DATA
beagle:00402D93    push    0                ; header_length
beagle:00402D95    push    0                ; header
beagle:00402D97    push    [ebp+hMem]       ; http://www.elrasshop.de/1.php?p=6...
beagle:00402D9A    push    eax              ; handle from InternetOpen
beagle:00402D9B    call   InternetOpenUrlA
beagle:00402DA0    xchg   eax, ebx
beagle:00402DA1    or     ebx, ebx
beagle:00402DA3    jz     short loc_402DAB
beagle:00402DA5    push  ebx
beagle:00402DA6    call   InternetCloseHandle
beagle:00402DAB
beagle:00402DAB  loc_402DAB:                ; CODE XREF: sub_402D3D+66 j
beagle:00402DAB    push  [ebp+var_4]
beagle:00402DAE    call   InternetCloseHandle
beagle:00402DB3    push  [ebp+hMem]         ; hMem
beagle:00402DB6    call   GlobalFree
beagle:00402DBB    xchg   eax, ebx
beagle:00402DBC    pop    esi
beagle:00402DBD    pop    ebx
beagle:00402DBE    leave
beagle:00402DBF    retn   4
beagle:00402DBF  sub_402D3D    endp
beagle:00402DBF

```

Code 7. The subroutine in Bagle.A which makes contact with attacker websites. The gray portions of the code are the resulting intraprocedural slice w.r.t. hMem at location 00402D97.

The assembly code shown in Code 7 is explained here. The Bagle virus has a hard-coded list of websites within its body that it tries to contact every 10 minutes once an infection has occurred. In this case, the subroutine sub_402D3D was called by sub_402DC2. sub_402DC2 calls sub_402D3D(website) for each website in that hard-coded list. sub_402D3D builds the URL string that will be used by our suspicious API via wsprintf and then checks for an Internet connection every 2 seconds via a call to InternetGetConnectedState. Once an Internet connection has been detected, sub_402D3D calls

`InternetOpen("beagle_beagle", 1, 0, 0, 0)`, which initializes an application's use of the `WinINet` functions. It tells the Internet DLL to initialize internal data structures and prepare for future calls from the application.⁶ It is imported from `wininet.dll`. The string "beagle_beagle" becomes the user agent in the HTTP protocol. The second parameter, 1, represents the type of access. In this case it means the virus will connect to the sites by trying to resolve all the hostnames locally. It is defined in `include\wininet.h`:

```
#define INTERNET_OPEN_TYPE_DIRECT 1 // direct to net
```

The third parameter is the `ProxyName`, but is `NULL` in this case because there was a direct connection to the Internet. The fourth parameter is the `ProxyBypass` addresses that will be not be routed through the proxy. The fifth parameter is the `Flags` parameter.

Two other suspicious API functions include `FindFirstFileA` and `FindNextFileA` since normally most executables do not make use of these API function calls. `FindFirstFileA(lpString1, lpFindFileData)` searches for `lpString1` (i.e. "C:*.*"), stores information about the file or directory (such as file name, and creation, access, and write times) in `lpFindFileData`, and returns a handle (`hFindFile`) to the file or directory. If the handle is invalid, meaning no files were found, the function frees up the allocated memory and exits. However, in the much more likely case that a file is found, a valid handle is returned. `FindNextFileA` is similar except that it continues a file search from where `FindFirstFileA` left off. Both are

⁶<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wininet/wininet/internetopen.asp>

imported from `kernel32.dll`. But in this case, let's do something different. It is well known that most viruses and worms spread by sending themselves to other email addresses. It is logical to assume that they will have some ability to either search for email addresses or create random ones. In our case, the Bagle virus searches all hard drives for email addresses. Searching for the "@" symbol (0x40) and the instructions used to compare values (i.e. `cmp`, `xor`, etc), we come upon the following two lines of assembly code:

<code>beagle:004029CC</code>	<code>cmp</code>	<code>al, 40h</code>
<code>beagle:00402A05</code>	<code>cmp</code>	<code>al, 40h</code>

If we do a backward slice on the `al` register at either one of those locations (they come from the same subroutine), it becomes clear that we can get a narrowed API call sequence (or FFSig) that will in some way be related to this critical instruction. This FFSig will only contain those instructions and API call sequences that are related to the register at the specified location. For example, if we do a backward slice on the value of the `al` register at address `00402A05`, a typical preliminary signature will include *at least* (it will include more, but for the sake of space, we list only some of the API functions) the API functions listed in Figure 14. Prior to each API function name is the DLL that exports it into the API. Following each API function name is a real number, which signifies the DLL the API function came from and the location within that DLL that the API function exists at (ordinal). See Section 3.4 for more details.

<code><Exporting DLL>:<API name>:<DLL>.<Ordinal></code>
<code>kernel32.dll:GlobalAlloc:1.491</code>
<code>kernel32.dll:GetLogicalDriveStringsA:1.366</code>
<code>kernel32.dll:GetDriveTypeA:1.332</code>
<code>kernel32.dll:GlobalAlloc:1.491</code>
<code>kernel32.dll:lstrcpyA:1.942</code>

```

kernel32.dll:LocalAlloc:1.584
kernel32.dll:LocalAlloc:1.584
kernel32.dll:lstrlenA:1.948
kernel32.dll:lstrcatA:1.933
kernel32.dll:FindFirstFileA:1.209
kernel32.dll:lstrcatA:1.933
shlwapi.dll:StrStrIA:3.816
kernel32.dll:CreateFileA:1.80
kernel32.dll:GetFileSize:1.348
kernel32.dll:CreateFileMappingA:1.81
kernel32.dll:MapViewOfFile:1.600
....
....
kernel32.dll:FindNextFileA:1.218
kernel32.dll:FindClose:1.205

```

Figure 14. One possible partial preliminary signature extracted from a slice of the Bagle virus. It contains API functions that are in the suspicious API database, as well as non-suspicious API functions that are not in that database.

Once this preliminary signature has been extracted, we augment it to create the final FFSig. The augmentation involves adding program state information and loop and branch information (via the CFG) and deleting non-suspicious API function calls from the preliminary signature. Loops and branches are detected by using the CFG created earlier in the process (and listed in numerical form in Appendix A). A loop in a CFG will be shown via a cycle (i.e. a back edge to a dominator or as in Appendix A, an arrow back up). The fact that there is a loop at this point in the code is noted within the signature by adding another parameter, the loop number(s) – multiple loop entries are comma separated – to which the API function belongs to. A CFG also represents all alternatives of control flow (i.e. branches caused by statements like `jmp`, `jz`, `jnz`, `jl`, etc.). This allows us to create different FFSigs by following different paths within the CFG. By looking into the suspicious API database, which not only includes individual suspicious APIs, but also suspicious *sequences* of APIs, we construct a signature that is devoid of any non-suspicious APIs by deleting them from the preliminary signature. This

database is originally built and maintained by people, so there is more flexibility allowed here. We also add program state information in order to facilitate the construction of a finite state automaton, which will be used later in the process to store the signature (see Section 3.3). The program state information is the address from where the call to the API function is made. The final FFSig is shown in Figure 15.

<Address of Call Site>:<Instruction Number>:<Exporting DLL>:<API name>:<DLL>.<Ordinal>:<loop number>
00402CED:1:kernel32.dll:GetLogicalDriveStringsA:1.366:
00402CFB:2:kernel32.dll:GetDriveTypeA:1.332:
00402C0D:3:kernel32.dll:FindFirstFileA:1.209:
00402BA6:4:shlwapi.dll:StrStrIA:3.816:1
00402A73:5:kernel32.dll:CreateFileA:1.80:1
00402A83:6:kernel32.dll:GetFileSize:1.348:1
00402A9B:7:kernel32.dll:CreateFileMappingA:1.81:1
00402AAF:8:kernel32.dll:MapViewOfFile:1.600:1
00402C77:9:kernel32.dll:FindNextFileA:1.218:1
00402C83:10:kernel32.dll:FindClose:1.205:

Figure 15. One possible final FFSig for the Bagle virus is constructed by augmenting the signature in Figure 14. The augmentation involves deleting non-suspicious API functions and adding program state (the address of the call site) and loop information.

The final FFSig shown in Figure 15 is present in all four versions of the Bagle virus we analyzed (See Appendix A, specifically 6). The only thing that may change within different versions of the Bagle virus is the first parameter, <Address of Call Site>, but in that case, we use the second parameter, <Instruction Number>. It is safe to assume that it is present in the rest of the 20+ versions. Once an FFSig has been extracted, we have to store it and subsequently compare it to either known signatures or new signatures.

3.3 An Automaton-Based Method for Storing the Signature

At this point we have extracted a slice and a signature. Now we need a compact way of storing it in a fully automatic and efficient manner. We utilize the finite-state automaton (FSA) based method presented by Sekar *et. al.* in [5] to do this. The space requirements for the FSA are low – of the order of a few kilobytes for typical programs [5]. These requirements will be even lower for the slice that was produced in the previous section, since a slice is a reduced version of the original program (usually up to 56-68%). Additionally, an FSA can capture an unbounded number of sequences of arbitrary length within a finite storage area, as well as structures like loops and branches.

The central difficulty in learning an FSA from strings is that the strings do not provide any direct information about internal states of the automaton [5]. For example, if we saw an API function call (i.e. `CreateFileA`) multiple times within the slice, we would not know whether to treat the multiple occurrences as being from the same automaton state or from different states if we did not have any other information. Sekar *et. al.* [5] solve this problem by utilizing the operating system (in their case it was Linux) to extract extra information, particularly the value of the program counter (PC), at the place where the system call occurs. This is shown in Table 3.

Sequence of system calls:	$S_0 S_1 S_2 S_4 S_2$
Sequence of system calls (with auxiliary data – PC):	$\frac{S_0}{1} \frac{S_1}{3} \frac{S_2}{4} \frac{S_4}{6} \frac{S_2}{7}$

Table 3. Sequence of system calls with and without program state information.

The algorithm described in [5] is based on tracing system calls during normal program execution and capturing the system state (the Program Counter) at the point of the system call. The key difference between this algorithm and our needs is that we never execute the potentially malicious program (or slice), so we have to adapt the algorithm to work statically. Sekar *et. al.* in [5] states that the technique they use to construct the FSA is similar to those used in compilers to capture control-flow graphs. This makes it possible to learn the FSA statically, without any runtime training. This is the reason why we augmented the preliminary signature with `<Address of Call Site>` and `<Instruction Number>`, instead of relying on the Program Counter, which we won't have access to. In Figure 14, we see multiple instances of the same API function calls, without any program states so we can't tell if the instances refer to the same API function call or to different ones. However, in Figure 15, the final FFSig has the necessary information (`<address of site call>`, `<instruction number>`, `<loop number>`) to allow us to differentiate between different calls to the same API function. This enables us to create an FSA from the FFSig in Figure 15, as shown in Figure 16. If there were branches within this part of the code, they would be present inside the FSA because there would be multiple FFSigs following the different possible branches, as would be shown in the CFG.

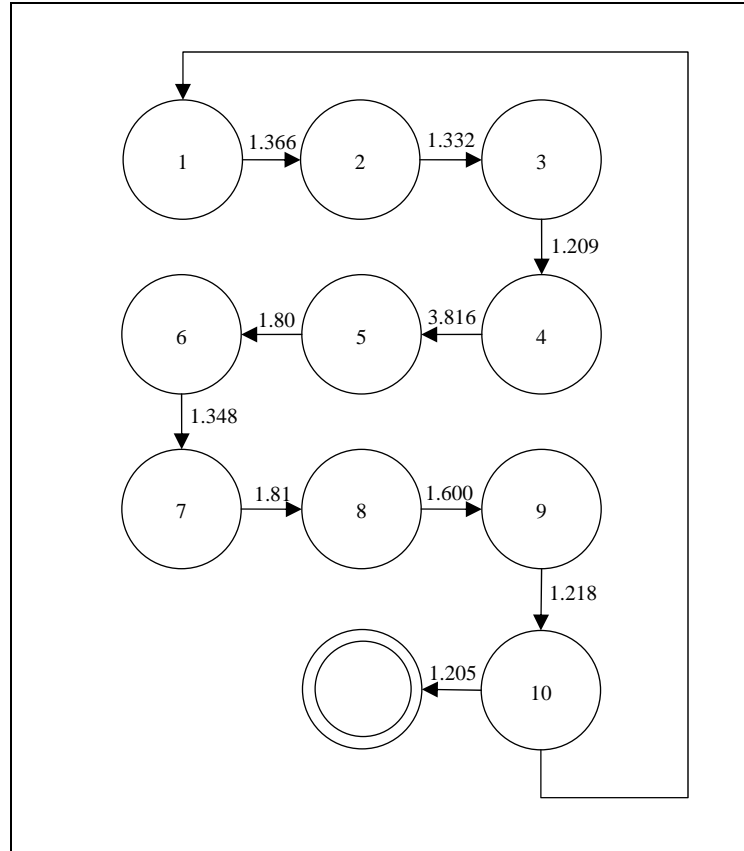


Figure 16. Derived FSA from the FFSig in Figure 15.

The FSA version of the FFSig has several advantages over an N-gram (i.e. string) version of the FFSig, including faster learning, better detection of certain classes of attacks, reduction in false positives, compact representation, and fast detection. For more information on each of these advantages, see [5].

As a side note, even if this process is not used to store the signature, they can be stored in more traditional ways such as arrays, vectors, or N-grams. These alternative methods can be used because of the following two reasons. The signatures can afford to be larger because of the evermore powerful machines used by users and the high-speed Internet connections available at most homes. Those two factors will compensate for the potentially longer processing time it would require to download, install, and compare longer signatures.

3.4 Similarity Measures for the Signatures

Once we have disassembled and analyzed the executable, extracted the signature, and stored the signature using the adapted automaton-based method presented by Sekar *et. al.* in [5], we are ready to utilize the signature in the detection phase and produce a report on the executable.

One way to compare two FSAs is to use a third FSA, similar to the way it is done in [5]. Sekar *et. al.* does real-time monitoring while the program is executing, as follows:

1. Obtain the corresponding location from where the call was made.
2. Check if there exists a transition from the current state to the new state that is labeled with the system call name that was intercepted.
3. Update the state of the automaton to correspond to the new state.

These steps can be modified to work on static signatures. Let's assume that the known virus signature is S_A and the new signature is S_B . The comparison would work as follows:

1. Obtain the corresponding location from where the call was made in S_B . The location is stored in the `<Address of Call Site>` and/or `<Instruction Number>` parameters.
2. Check if there exists a transition inside S_A from the current state (found in step 1) to the next state (the next instruction address/number) that is labeled with the API function id that was intercepted inside S_B . If it does not exist inside S_A , then we can transition to a sink state and compare S_B to the next known signature. If it exists, go to step 3.
3. Move to the next state within the automaton of S_B . If the next state is not in the automaton of S_A then move to a sink state and repeat this process with the next known signature.

There are other ways of comparing two FSAs. If we decide to view the FSA as an Attributed Relational Graph (ARG) then we can use the techniques presented

by D. Zhang *et. al.* [39] and L.P. Cordella *et. al.* [40] to compare them. An FSA is really just a directed, labeled graph. An Attributed Relational Graph (ARG)⁷ generalizes the ordinary graph by attaching attributes to its vertexes and edges. This problem is also closely related to graph and subgraph⁸ matching, which is the problem of finding correspondences between graphs. Part of graph and subgraph matching requires being able to compare two matched vertices of the two graphs with a distance function (see below), which will measure how similar the two vertices are. There are even more techniques and implementations (i.e. toolkits) available in the public domain including GraphGrep [41], which allows you to query for a graph (i.e. an FSA) in database of graphs. Given a collection of graphs and a pattern graph, GraphGrep finds all the occurrences of the pattern in each graph. The pattern is a subgraph and it can be also a tree, a path, or a node. The pattern is expressed as a list of nodes and a list of edges [41].

If we decide not to use the FSA created in Section 3.3 as the basis for the comparison, we can instead use a vector version of the FFSig (i.e. a list of elements, like a string) with the following techniques. There are many techniques that can be used to measure similarity among two data sequences (i.e. two FFSigs). Extensive research in this field has been done by many including Nambiar *et. al.* [34], Noreault *et. al.* [35], and Cohen [36]. There are many types of techniques that

⁷ D. Zhang *et. al.* [39] defines an ARG is defined as the following: An attributed relational graph is a triple $G = (V, E, A)$, where V is the vertex set, E is the edge set, and A is the attribute set that contains unary attribute a_i attaching to each node $n_i \in V$, and binary attribute a_{ij} attaching to each edge $e_k = (n_i, n_j) \in E$.

⁸ mathworld.com defines a subgraph as follows: A graph G' whose graph vertices and graph edges form subsets of the graph vertices and graph edges of a given graph G . If G' is a subgraph of G , then G is said to be a supergraph of G' .

can be used to measure similarity between sequences. They include the traditional similarity measures such as cosine measures, Euclidean distance, Pearson Product Moment Correlation (called Pearson's Correlation for short) measure, overlap coefficient, and the Jaccard measures. In fact, multiple measures can be used when comparing signatures to get a more precise result.

One of the most common measures of similarity is the Euclidean distance formula. In general, the distance between two points x and y in Euclidean space (sometimes called n -space or the space of all n -tuples of real numbers), is given Figure 17.

$$d = |x - y| = \sqrt{\sum_{i=1}^n |x_i - y_i|^2}$$

Figure 17. The Euclidean Distance Measure.

Using this formula, one can easily and quickly find the similarities between two vectors (i.e. a set of zero or more points). For example, say the following API function call sequence `GetLogicalDriveStringsA`, `GetDriveTypeA`, `FindFirstFileA`, `StrStrIA`, `CreateFileA`, `GetFileSize`, `CreateFileMappingA`, `MapViewOfFile`, `FindNextFileA`, and `FindClose` could be mapped into the following vector (1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218, 1.205). The integral part of the number (to the left of the decimal point) corresponds to the DLL from which the API function is imported from, which in this case 1 stands for `kernel32.dll` and 3 for `shlwapi.dll`. The fractional part (to the right of the decimal point) simply represents the location at which the API function exists within the DLL. Alternatively, each suspicious API function can be mapped into a

more appropriate id (i.e. 32-bit numbers). Once another signature is constructed, it can be compared to this one using the Euclidean formula, as is shown in Table 4.

Known virus signature (S_A):	(1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218, 1.205)
New signature (S_B):	(1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218, 1.205)
Euclidean Distance (S_A, S_B):	0

Table 4. Euclidean distance measure between FFSigs from Bagle.A and Bagle.B.

The Euclidean distance measure has several flaws. First, the decision on whether two signatures matched does not depend on the length of the signature. This is bad because two signatures could be identical, except for one part, which could result in a large Euclidean distance measure (see the section on Pearson's Correlation measure). Identical signatures would have a Euclidean distance of 0. Ideally, the longer the vector, the less each variation should effect the final similarity measure. The second flaw has to do with sequence alignment. A new call sequence that is identical to a known signature, but is not properly aligned with the known signature will likely not have a low Euclidean distance measurement. For example, in Table 5, the Euclidean distance is not as low as it ideally would be if the two signatures were properly aligned.

Known virus signature (S_A):	(1.491, 1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218)
New signature (S_B):	(1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218, 1.205)
Aligned known virus signature (S_A'):	(1.491, 1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218, -----)
Aligned new signature (S_B'):	(-----, 1.366, 1.332, 1.209, 3.816, 1.80, 1.348, 1.81, 1.600, 1.218, 1.205)
Euclidean Distance (S_A, S_B):	3.361
(before alignment)	
Euclidean Distance (S_A', S_B'):	1.917

(after alignment)

Table 5. The effect of misalignment in the Euclidean distance measure.

This global and formal⁹ misalignment problem can be solved by applying the techniques described by Wilson in [37]. This technique is similar to the Needleman-Wunsch Algorithm used in comparing sequences of DNA samples (See Figure 18). In addition, this is also similar to the problem of finding the Longest Common Sequence (LCS) as is described by Cormen *et. al.* in [38]. Additionally, it has the same cost of $O(nm)$, where n and m are the lengths of the two strings. The technique can be conceptualized as loading the two signatures inside a matrix, with one signature placed on the horizontal axis and the second signature placed on the vertical axis, and then computing the score of the best path to each cell, as is shown in Figure 19. Any matches are marked off with an (X). Beginning with the uppermost left cell, the score of the best path to that cell is calculated by adding the highest score to the top and left of the cell with a 1 if a match occurs and a 0 if match does not occur. Once the scores have been calculated, the highest score is where the best alignment occurs. The correct alignment is constructed by working backwards from the maximum match and adding gaps or performing deletions. When finished, the best alignment has been achieved. See Table 5. It is important to note that even when the best alignment has been achieved, it is possible that the Euclidean distance will be larger than when the vectors were not aligned.

⁹ Formal alignment is a precise quantitative scoring system for matches and gaps.

	M	P	R	C	L	C	Q	R	J	N	C	B	A
P	0	1	0	0	0	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1	1	1	1	1	2
R	0	0	2	1	1	1	1	2	1	1	1	1	2
C	0	0	1	3	2	3	2	2	2	2	3	2	2
K	0	0	1	2	3	3	3	3	3	3	3	3	3
C	0	0	1	3	3	4	3	3	3	3	4	3	3
R	0	0	2	2	3	3	4	5	4	4	4	4	4
N	0	0	1	2	3	3	4	4	5	6	5	5	5
J	0	0	1	2	3	3	4	4	6	5	6	6	6
C	0	0	1	3	3	4	4	4	5	6	7	6	6
J	0	0	1	2	3	3	4	4	6	6	6	7	7
A	0	0	1	2	3	3	4	4	5	6	6	7	8

MP-RCLCQR-JNCBA
 | | | | | | |
 -PBRCKC-RNJ-CJA

Figure 18. The Needleman-Wunsch Algorithm used in aligning DNA sequences.

	1.1	1.2	1.3	1.4	1.5	1.6	
2.1							→
1.1	(X)						→
1.2		(X)					→
1.3			(X)				→
1.4				(X)			→
1.5					(X)		→

	1.1	1.2	1.3	1.4	1.5	1.6
2.1	0	0	0	0	0	0
1.1	1	0	0	0	0	0
1.2	0	2	1	1	1	1
1.3	0	1	3	2	2	2
1.4	0	1	2	4	3	3
1.5	0	1	2	3	5	4

Figure 19. The sequence alignment algorithm in action on the sequences from Table 5. Match = +1, Mismatch = 0.

We can use the other classical similarity measures including cosine measure, Pearson’s Correlation, and the Jaccard coefficient, for greater accuracy when comparing two signatures. Each of the similarity measures has advantages and disadvantages. The cosine measure is given in Figure 20 and it measures the cosine of the angle between the two vectors. The cosine measure does not depend on the length of the vectors (i.e. signatures), meaning that signatures of different

¹⁰ This image taken from slides at <http://www.maths.tcd.ie/~lily/pres2/sld007.htm>

lengths, but similar composition will be treated as similar as opposed to radically different.

$$\text{Cosine Measure} = \frac{S_1 S_2}{|S_1| \cdot |S_2|}$$

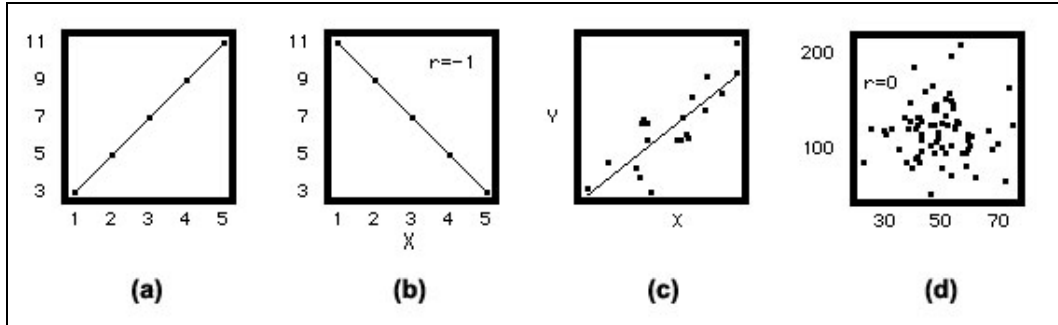
Figure 20. Cosine similarity measure.

The Jaccard coefficient measures the proportion of overlap (the shared elements) between two vectors to the total number of elements shared between the two vectors. The formula for the Jaccard coefficient is shown in Figure 21.

$$\text{Jaccard Coefficient} = \frac{S_1 S_2}{S_1 + S_2 - S_1 S_2}$$

Figure 21. Jaccard Coefficient.

Pearson's Correlation measure is another popular similarity measure. It measures the degree to which two variables (or vectors) are related. It reflects the strength of linear relationship (i.e. how close the points are to forming a straight line) between the two variables. It ranges from +1 to -1, with +1 meaning a perfect positive linear relationship, and -1 means the opposite. Figure 22 has examples of various Pearson's Correlations and linear relationships. The formula for Pearson's Correlation is given in Figure 23. Pearson's Correlation measure has the advantage of minimizing the effects of a single wildly different element if the rest of the elements are similar.



11

Figure 22. Examples of Pearson's Correlations. (a) has a perfect positive linear relationship (+1). (b) has a perfect negative linear relationship (-1). (c) has a strong, but not perfect positive linear relationship. (d) has no linear relationship (0).

$$\text{Pearson's Correlation} = r = \frac{\sum S_1 S_2 - \frac{\sum S_1 \sum S_2}{N}}{\sqrt{\left(\sum S_1^2 - \frac{(\sum S_1)^2}{N} \right) \left(\sum S_2^2 - \frac{(\sum S_2)^2}{N} \right)}}$$

Figure 23. Pearson's Correlation.

¹¹ Image composed from images found at <http://davidmlane.com/hyperstat/index.html>

4 Chapter 4

4.1 Conclusion

Current virus signatures used by AV scanners are alarmingly weak as demonstrated by [2, 3, and 21]. A need for stronger signatures based on different information exists. One solution to this problem is to use Functional Flow Signature (FFSig), which is a virus signature that would encompass a specific sequence of API calls, instead of the small bit of binary code currently used for signatures which could be easily modified.

In this thesis, we present and describe a methodology for efficiently and statically analyzing a potentially malicious Windows executable, extracting specific sequences of Win32 API calls made by the executable, storing them efficiently by utilizing the method presented in [5], and then using several similarity measures to compare known and unknown signatures. This forms the basis for FFSig. Our algorithm does not require any modification to the host operating system and does not require the executable to be run as it would if it was dynamically analyzed. In addition, by only doing static analysis, this allows us to scan a large number of executables in a short period of time.

The first step of this process is the disassembly of the target executable, which yields an assembly version of the code. The first step also includes the analysis of the assembly code to simplify it and improve the analyzability of it.

This results in abstractions and high-level representations of the assembly code. The second step is to extract certain malicious parts of the code in order to analyze it more closely. This step reduces and focuses the amount of code that has to be analyzed (in effect it decreases the complexity of the detection process). This is accomplished by using slicing techniques on certain areas of the assembly code. The third is to store the information from the slice using a finite state automaton (this becomes the signature). The fourth and final step is to use various graph matching techniques and similarity measures to compare different signatures and then produce a report.

4.2 Future Work

There are several interesting problems for future research in the area of static analysis of executables for detecting malicious behaviors. They include the following research areas. Can frequency information (i.e. how many times does a loop run, how many times a certain API function was called, etc.) be incorporated into the data contained by the FFSig in order to provide a better profile of the executable? This may allow us to get a more accurate results and stronger signatures. Another possibility involves the use of API function call argument values (i.e. by doing backward slicing on them). This again may strengthen the signature (i.e. the signature can contain information on which files are accessed) and reduce the number of false positives.

Another set of possible research topics include the following questions. Can slicing be used on the virus to “sterilize” it so that it is no longer malicious or

dangerous to the system? Can slicing be used to extract only the malicious parts of the virus in order to analyze and study that part much more effectively? Can virus signatures encompass any other information that would make them stronger against changes? What is the proper threshold for the ratio of similarities between different signatures? Can this process be adapted to deal with encrypted or polymorphic viruses? Can this entire process be automated in an efficient way so human intervention is minimized or not needed at all?

5 Appendixes

5.1 Appendix A: Functional Flows

The following is the functional flow of variant A of the Bagle virus. This was created by manual inspection of the virus through the use of IDA Pro. For more details on all the Win32 API calls that are called below, see Appendix A. The sub_ functions are also described in detail in Appendix A of [2].

1. CoInitialize – initialize the COM library.
2. sub_401835 – this function does many things; see below for details.
 - 2.1. sub_401669 – check that the current date is earlier than January 28, 2004, otherwise exit.
 - 2.1.1. GetLocalTime
 - 2.1.2. sub_401000 – zeroes out number of bytes from starting address.
 - 2.1.3. SystemTimeToFileTime
 - 2.1.4. SystemTimeToFileTime
 - 2.1.5. CompareFileTime
 - 2.2. GetTickCount
 - 2.3. sub_40126F – fills memory with random data using the result from GetTickCount as the random seed.
 - 2.4. sub_4015A5 – check/create a registry entry. (uid)
 - 2.4.1. RegCreateKey
 - 2.4.2. RegQueryValueEx
 - 2.4.3. sub_4012AA – returns a random value less than passed argument.
 - 2.4.4. RegSetValueEx
 - 2.4.5. RegCloseKey
 - 2.5. WSASStartup – initialize the use of Windows Sockets.
 - 2.6. sub_402ADD – allocate heap memory.
 - 2.6.1. sub_401524 – wrapper function.
 - 2.6.1.1. GlobalAlloc
 - 2.7. CreateMutex
 - 2.8. sub_402737 – creates a mutex and allocates heap memory.
 - 2.8.1. CreateMutex
 - 2.8.2. GlobalAlloc
 - 2.9. sub_4016CA – make a base64-encoded copy of the virus for use with email.
 - 2.9.1. GlobalAlloc
 - 2.9.2. GetModuleFileName
 - 2.9.3. CreateFile
 - 2.9.4. GetFileSize

- 2.9.5. CreateFileMapping
 - 2.9.6. MapViewOfFile
 - 2.9.7. GlobalAlloc
 - 2.9.8. sub_4010DD – wrapper function.
 - 2.9.9. strlen
 - 2.9.10. UnmapViewOfFile
 - 2.9.11. CloseHandle
 - 2.9.12. GlobalFree
 - 2.10. GetSystemDirectory
 - 2.11. GetModuleFileName
 - 2.12. lstrcat
 - 2.13. sub_401625 – check/create a registry entry. (d3dupdate.exe)
 - 2.14. StrStrI
 - 2.15. GetCommandLine
 - 2.16. WinExec – if the virus is not run from %system%\bbeagle.exe, execute calc.exe.
 - 2.17. CopyFile
 - 2.18. WinExec – run the virus from the system directory. (*to continue executing following functions*)
 - 2.19. sub_4017DC – check/create a registry entry. (frun)
 - 2.20. sub_40179B – check/create a registry entry. (frun)
3. If port number is 0, choose a random port between 5000 and 50000.
4. sub_401C78 – creates a new thread that listens on port 6777 and accepts and processes connections.
- 4.1. GlobalAlloc
 - 4.2. CreateThread
 - 4.2.1. StartAddress – starting address of newly created thread.
 - 4.2.1.1. sub_401000 – see Appendix A.
 - 4.2.1.2. socket
 - 4.2.1.3. GlobalFree
 - 4.2.1.4. bind
 - 4.2.1.5. listen
 - 4.2.1.6. accept
 - 4.2.1.6.1. CreateThread
 - 4.2.1.6.1.1. sub_4030F6 – receives and processes data from attacker.
 - 4.2.1.6.1.1.1. sub_4013D2 – wrapper function.
 - 4.2.1.6.1.1.1.1. CreateStreamOnHGlobal
 - 4.2.1.6.1.1.2. sub_4019CF – receives data from socket.
 - 4.2.1.6.1.1.2.1. sub_401972 – wrapper function.
 - 4.2.1.6.1.1.2.1.1. select
 - 4.2.1.6.1.1.2.2. recv
 - 4.2.1.6.1.1.3. sub_40146E – wrapper function.
 - 4.2.1.6.1.1.3.1. sub_4013F7 – wrapper function.
 - 4.2.1.6.1.1.3.1.1. call to unknown function in ole32.dll.
 - 4.2.1.6.1.1.4. sub_401000 – see Appendix A.
 - 4.2.1.6.1.1.5. sub_402E2B - allows uploading and executing of files.
 - 4.2.1.6.1.1.5.1. WaitForSingleObject
 - 4.2.1.6.1.1.5.2. sub_401000 – see Appendix A.
 - 4.2.1.6.1.1.5.3. sub_401481 – wrapper function.
 - 4.2.1.6.1.1.5.3.1. sub_40146E – wrapper function.
 - 4.2.1.6.1.1.5.3.2. call to unknown function in ole32.dll.
 - 4.2.1.6.1.1.5.4. sub_4019CF – see Appendix A.
 - 4.2.1.6.1.1.5.5. sub_40146E – see Appendix A.
 - 4.2.1.6.1.1.5.6. sub_401481 – see Appendix A.

- 4.2.1.6.1.1.5.7. sub_401A38 – see Appendix A.
- 4.2.1.6.1.1.5.8. sub_40146E – see Appendix A.
- 4.2.1.6.1.1.5.9. sub_401481 – see Appendix A.
- 4.2.1.6.1.1.5.10. lstrcmpi
- 4.2.1.6.1.1.5.11. send
- 4.2.1.6.1.1.5.12. sub_4019CF – see Appendix A.
- 4.2.1.6.1.1.5.13. sub_40146E – see Appendix A.
- 4.2.1.6.1.1.5.14. sub_401481 – see Appendix A.
- 4.2.1.6.1.1.5.15. sub_4019CF – see Appendix A.
- 4.2.1.6.1.1.5.16. sub_40146E – see Appendix A.
- 4.2.1.6.1.1.5.17. GetWindowsDirectory
- 4.2.1.6.1.1.5.18. sub_401023 – create random letters.
- 4.2.1.6.1.1.5.18.1. sub_4012AA – see Appendix A.
- 4.2.1.6.1.1.5.19. lstrcat
- 4.2.1.6.1.1.5.20. CreateFile
- 4.2.1.6.1.1.5.21. WriteFile
- 4.2.1.6.1.1.5.22. WinExec
- 4.2.1.6.1.1.5.23. sub_401184 – kill and delete the currently
executing virus.
- 4.2.1.6.1.1.5.24. closesocket
- 4.2.1.6.1.1.5.25. ReleaseMutex
- 4.2.1.6.1.1.6. sub_4013E5 – wrapper function.
- 4.2.1.6.2. CloseHandle
- 4.2.1.7. closesocket
- 4.3. CloseHandle

- 5. sub_402E07 – creates a new thread that contacts a list of websites every 10 minutes to inform of infection.

- 5.1. CreateThread
 - 5.1.1. sub_402DED – wrapper function.
 - 5.1.1.1. sub_402DC2 – wrapper function.
 - 5.1.1.1.1. sub_401669 – see Appendix A.
 - 5.1.1.1.2. sub_402D3D – loop through each hard coded site and contact
them.
 - 5.1.1.1.2.1. GlobalAlloc
 - 5.1.1.1.2.2. wsprintf
 - 5.1.1.1.2.3. sub_402D22 – checks that the Internet connection is up.
 - 5.1.1.1.2.3.1. InternetGetConnectedState
 - 5.1.1.1.2.3.2. Sleep (for 2 seconds)
 - 5.1.1.1.2.4. InternetOpen
 - 5.1.1.1.2.5. InternetOpenUrl
 - 5.1.1.1.2.6. InternetCloseHandle
 - 5.1.1.1.2.7. GlobalFree
 - 5.1.1.1.2. Sleep (for 10 minutes)
 - 5.1.2. CloseHandle

- 6. sub_402CCE – searches fixed drives for email addresses and emails itself to them.

- 6.1. GlobalAlloc
- 6.2. GetLogicalDriveStringsA
- 6.3. GetDriveTypeA
- 6.4. sub_402C9D – wrapper function.
 - 6.4.1. GlobalAlloc
 - 6.4.2. lstrcpyA
 - 6.4.3. sub_402BCB – wrapper function.
 - 6.4.3.1. LocalAlloc
 - 6.4.3.2. LocalAlloc

- 6.4.3.3. lstrlenA
- 6.4.3.4. lstrcatA
- 6.4.3.5. FindFirstFile
- 6.4.3.6. lstrcatA
- 6.4.3.7. sub_402B8F – see Appendix A.
 - 6.4.3.7.1. StrStrIA
 - 6.4.3.7.2. sub_402A5A – see Appendix A.
 - 6.4.3.7.2.1. CreateFileA
 - 6.4.3.7.2.2. GetFileSize
 - 6.4.3.7.2.3. CreateFileMappingA
 - 6.4.3.7.2.4. MapViewOfFile
 - 6.4.3.7.2.5. sub_402985 – finds an email address in a file.
 - 6.4.3.7.2.5.1. Sleep
 - 6.4.3.7.2.5.2. sub_4028A5 – see Appendix A.
 - 6.4.3.7.2.5.3. sub_4028F3 – see Appendix A.
 - 6.4.3.7.2.5.4. lstrlenA
 - 6.4.3.7.2.5.5. sub_40293D – see Appendix A.
 - 6.4.3.7.2.5.6. sub_40295A – see Appendix A.
 - 6.4.3.7.2.5.7. sub_402B2C – see Appendix A.
 - 6.4.3.7.2.6. UnmapViewOfFile
 - 6.4.3.7.2.7. CloseHandle
 - 6.4.3.7.2.8. CloseHandle

- sub_402B2C – makes sure the email address is not to certain domains/usernames.
 - sub_402AF6 – see Appendix A.
 - sub_4014F3 – see Appendix A.
 - sub_40153E – see Appendix A.
 - sub_402465 – finds out which DNS server to use.
 - StrRChrA
 - sub_4020B1 – see Appendix A.
 - sub_401CBC – see Appendix A.
 - GlobalAlloc
 - GetNetworkParams
 - GlobalFree
 - sub_4013D2 – see Appendix A.
 - CreateStreamOnHGlobal
 - sub_401D2C – see Appendix A.
 - sub_401000 – see Appendix A.
 - call to unknown function in
 - lstrlenA
 - call to unknown function in
 - call to unknown function in
 - call to unknown function in
 - call to unknown function in
 - call to unknown function in
 - sub_401E1A – finds the MX record for e-mail address.
 - sub_401B25 – see Appendix A.
 - sub_401426 – see Appendix A.
 - sub_40146E – see Appendix A.
 - sub_401481 – see Appendix A.
 - sub_4019CF – see Appendix A.
 - sub_40146E – see Appendix A.

- sub_401481 – see Appendix A.
- sub_4019CF – see Appendix A.
- closesocket
- sub_4013E5 – see Appendix A.
 - call to unknown function in
- sub_40280C – wrapper function.
 - WaitForSingleObject
 - StrDupA
 - StrDupA
 - sub_40249F – see Appendix A.
 - lstrlenA
 - GlobalAlloc
 - GlobalAlloc
 - lstrcpyA
 - CreateThread
 - sub_402778 - creates the infected email and send it.
 - CloseHandle
 - ReleaseMutex
- GlobalFree
- lstrcpyA
- Sleep
- FindNextFileA
- FindClose
- LocalFree
- LocalFree
- GlobalFree
- lstrlenA
- GlobalFree

7. Sleep (for 1 second)

Important and similar parts of Bagle variant B: variant B is identical to variant A, except that it is compressed and several names and variables are changed.

Important and similar parts of Bagle variant C: variant C has many identical parts with variant A, the most important of which include the thread that searches the hard drive for emails, the thread that creates the email and sends it out, and the thread that contacts various websites to alert the attackers of the infection.

Important and similar parts of Bagle variant D: variant D is identical to variant C except for variable names and values, and thus has identical parts with variant A.

6 Bibliography

- [1] J. O. Kephart and W. C. Arnold. *Automatic Extraction of Computer Virus Signatures*. Proceedings of the 4th Virus Bulletin International Conference, R. Ford, ed., Virus Bulletin Ltd., Abingdon, England, 1994, pp. 178-184.
- [2] K. Rozinov. *Reverse Code Engineering: An In-Depth Analysis of the Bagle Virus*. August 2004, <http://rozinov.sfs.poly.edu>.
- [3] A. H. Sung, J. Xu, P. Chavez, S. Mukkamala. *Static Analyzer of Vicious Executables (SAVE)*. Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004), 2004.
- [4] R. Wang. *Flash in the pan?* Virus Bulletin, July 1998. Virus Analysis Library.
- [5] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni. *A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors*. IEEE Symposium on Security and Privacy, 2001.
- [6] K. Rozinov. *PE File Infection Techniques*. February 2005, <http://rozinov.sfs.poly.edu>.
- [7] M. Pietrek. *An In-Depth Look into the Win32 Portable Executable File Format*. MSDN Magazine, February 2002.
- [8] M. Pietrek. *An In-Depth Look into the Win32 Portable Executable File Format, Part 2*. MSDN Magazine, March 2002.
- [9] I. Ivanov. *API Hooking Revealed*. The Code Project. <http://www.codeproject.com/system/hooks.asp>.
- [10] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes. *Next-generation Intrusion Detection Expert System (NIDES): A Summary*. SRI-CSL-95-07, SRI International, 1995.
- [11] S. Forrest, S. A. Hofmeyr, A. Somayaji. *Intrusion Detection using Sequences of System Calls*. Journal of Computer Security Vol. 6 (1998) pg 151-180.

- [12] P. A. Porras and P. G. Neumann. *Emerald: Event monitoring enabling responses to anomalous live disturbances*. In Proceedings of the 20th National Information Systems Security Conference, pages 353-365, October 1997.
- [13] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, W. Gong. *Anomaly Detection Using Call Stack Information*. IEEE Symposium on Security and Privacy, 2003.
- [14] F. Cohen. *Computer Viruses: Theory and Experiments*. Computers and Security, 6:22-35, 1987.
- [15] D. M. Chess, S. R. White. *An Undetectable Computer Virus*. In Proceedings of Virus Bulletin Conference, 2000.
- [16] W. Landi. *Undecidability of Static Analysis*. ACM Letters on Programming Languages and Systems, 1(4):323-337, December 1992.
- [17] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. *On the (Im)possibility of Obfuscating Programs*. In Advances in Cryptology -CRYPTO'01, volume 2139 of Lecture Notes in Computer Science, pages 1 – 18, Santa Barbara, CA, August 2001. Springer-Verlag.
- [18] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In 2002 IEEE Symposium on Security and Privacy, pages 143–159, May 2002.
- [19] M. Bishop and M. Dilger. *Checking For Race Conditions in File Accesses*. Computing Systems, 9(2), 1996.
- [20] B.V. Chess. *Improving Computer Security Using Extended Static Checking*. In 2002 IEEE Symposium on Security and Privacy, pages 160–173, May 2002.
- [21] M. Christodorescu, S. Jha. *Static Analysis of Executables to Detect Malicious Patterns*. In Proceedings of the 12th USENIX Security Symposium, Washington, DC, August 2003.
- [22] J. Bergeron, M. Debbabi, M. M. Erhioui, B. Ktari. *Static Analysis of Binary Code to Isolate Malicious Behaviors*. In Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises, pages 184 – 189, 1999.
- [23] *IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*.
<ftp://download.intel.com/design/Pentium4/manuals/25366614.pdf>
- [24] IDA Pro. <http://www.datarescue.com/idabase/>

- [25] M. Weiser. *Program Slicing*. In the Proceedings of the 5th international conference on Software Engineering, pages 439 - 449, 1981.
- [26] K. Li. *Modes, Registers and Addressing and Arithmetic Instructions* (CS217 Class Slides).
<http://www.cs.princeton.edu/courses/archive/spring04/cos217/notes/IA32-I.pdf>
- [27] A. Kiss, J. Jasz, G. Lehotai, T. Gyimothy. *Interprocedural Static Slicing of Binary Executables*. In the Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, pages 118 – 127, Sep. 2003.
- [28] C. Cifuentes, A. Fraboulet. *Intraprocedural Static Slicing of Binary Executables*. In the Proceedings of the Third IEEE International Conference on Software Maintenance, pages 188 – 195, Oct. 1997.
- [29] UPX - the Ultimate Packer for eXecutables. <http://upx.sourceforge.net/>
- [30] The Component Object Model.
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/html/f5f66603-466c-496b-be29-89a8ed9361dd.asp>
- [31] S. Horwitz, T. Reps, D. Binkley. *Interprocedural Slicing Using Dependence Graphs*. ACM Transactions on Programming Languages and Systems, 12(1):26-60, January 1990.
- [32] T. Lengauer, R. E. Tarjan. *A Fast Algorithm for Finding Dominators in a Flowgraph*. ACM Transactions on Programming Language and Systems, 1(1):121-141, July 1979.
- [33] J. Ferrante, K. J. Ottenstein, J. D. Warren. *The Program Dependence Graph and Its Use in Optimization*. ACM Transactions on Programming Languages and Systems, 9(3):319-349, July 1987.
- [34] U. Nambiar, S. Kambhampati. *Answering Imprecise Database Queries*. Presentation at WIDM 2003, New Orleans, LA. Arizona State University.
- [35] T. Noreault, M. McGill, M. B. Koll. *A Performance Evaluation of Similarity Measures, Document Term Weighting Schemes and Representations in a Boolean Environment*. In the Proceedings of the 3rd Annual ACM Conference on Research and Development in Information Retrieval, pages 57-76, 1980.
- [36] W. Cohen. *The WHIRL Approach to Information Integration*. In the Proceedings of IEEE Intelligent Systems, pages 20-23, Sept/Oct 1998.

- [37] W.C. Wilson. *Activity Pattern Analysis by Means of Sequence-Alignment Methods*. Journal of Environment and Planning. Vol 30, pages 1017 – 1038, 1998.
- [38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, Cambridge, MA, and McGraw-Hill Book Company, New York, NY, pages 350 – 356, 2001.
- [39] D. Zhang, S. Chang. *Stochastic Attributed Relational Graph Matching for Image Near-Duplicate Detection*. DVMM Technical Report 2004.
- [40] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. *An Efficient Algorithm for the Inexact Matching of ARG Graphs Using a Contextual Transformational Model*. In the Proceedings of the International Conference on Pattern Recognition (ICPR '96). Volume 3, pages 180 – 184, Aug 25 - 29, 1996.
- [41] R. Giugno and D. Shasha. *GraphGrep: A Fast and Universal Method for Querying Graphs*. In the Proceedings of the 16th International Conference on Pattern Recognition (ICPR'02) Volume 2. Pages 112 – 115, 2002.