Obfuscated Malicious Executable Scanner

Jianyun Xu

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA

Andrew H. Sung, Srinivas Mukkamala and Qingzhong Liu

Department of Computer Science, New Mexico Tech, Socorro, NM 87801, USA
Institute for Complex Additive Systems Analysis, New Mexico Tech, Socorro, NM 87801, USA
dennisxu@microsoft.com {sung,srinivas,liu}@cs.nmt.edu

The proliferation of malware (viruses, Trojans, and other malicious code) in recent years has presented a serious threat to individual users, enterprises, and organizations alike. Current static scanning techniques for malware detection have serious limitations; on the other hand, sandbox testing fails to provide a complete satisfactory solution either due to time constraints (e.g., time bombs cannot be detected before its preset time expires). What is making the situation worse is the ease of producing polymorphic (or variants of) computer viruses that are even more complex and difficult than their original versions to detect.

In this paper, we propose a new approach for detecting polymorphic malware in the Windows platform. Our approach rests on an analysis based on the Windows API calling sequence that reflects the behaviour of a particular piece of code. The analysis is carried out directly on the PE (portable executable) code. It is achieved in two basic steps: construct the API calling sequences for both the known virus and the suspicious code, and then perform a similarity measurement between the two sequences after a sequence realignment operation is done. An alternative technique based on comparing the bags of API calls, and the technique's performance, are also studied. Favourable (in terms of time and accuracy of detection) experimental results are obtained and presented.

Keywords: Polymorphic malware detection, API sequence, sequence realignment, similarity measure

ACM Classification: K.6.5 (Security and Protection)

1. INTRODUCTION

The risk of using the Internet has increased tremendously. The connectivity of the Internet that is vital to individual users and enterprises may also expose their critical, valuable information and/or information systems to attacks and adversaries. One of the greatest recent threats to security has come from automated, pre-scanned, self-propagating attacks such as blended viruses and worms. These attacks scan at random until they are able to place a harmful program on the victim server using a maliciously crafted request. The program then uses the now-infected server as a base from which to attack other vulnerable servers. The result is an exponential growth in the number of

Copyright© 2007, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 4 October 2005
Communicating Editor: Phoebe Chen

attackers leading to load-induced network failure. Though organizations have a wide variety of protection mechanisms (firewalls, anti virus tools, and intrusion detection systems) against cyber attacks, recent hybrid and blended malware like Sasser, Blaster, Slammer, Nimda and CodeRed have worked their way past the current security mechanisms. Due to the seemingly increasing number and intensity of malware attacks, information security companies and researchers are hard-pressed to find effective new ways to help thwart or defend against such attacks.

Bergeron *et al* (1999) gave a classification of malware into three generations, based on payload, enabling vulnerability.

A. *First Generation*
- Malware that share the properties of a virus
- Requires human action to trigger replication and spreading
- Propagates via email and file sharing

Examples: Melissa, LoveLetter, VBScript worm, SoBig

B. *Second Generation*
- Malware that shares the properties of a worm
- Does not require human intervention for replication and spreading
- Automatic scanning of victims for vulnerabilities
- Hybrid in nature, blended with viruses and trojans
- Propagates via Internet

Examples: Slapper worm, SQL Slammer worm, and Blaster worm

C. *Third Generation*
- Pre-complied vulnerable targets
- Exploits known and unknown vulnerabilities to the broader security communities
- Employs multiple attack vectors
- Geographical region- or organization-specific malware
- Attack security technologies and products

Theoretical studies by Cohen (1987) and Chess *et al* (2000) on virus detection have shown that there is no algorithm that can detect all types of viruses; while heuristics-based static analysis techniques have been proposed by researchers for virus detection. The rise in the number of variants of original malware and their effects have shown the inability of many commercial-grade antivirus scanners to detect even slight modifications to the original virus; thus making obfuscation and de-obfuscation of vicious executables an interesting problem, as presented in Strehl's work (2002). Detection techniques using a program annotator have been proposed by Bergeron *et al* (1999) and Collberg (2002), however, the amount of time taken for analysis by these techniques as reported by the authors for even simple malware is too high for them to be suitable for real time detection.

Polymorphic and metamorphic viruses and polymorphic shellcodes (Detristan *et al,* 2003) are designed to bypass detection tools. There is strong evidence that commercial malware detectors are susceptible to common evasion techniques used by malware writers: previous testing work has shown that malware detectors cannot cope with obfuscated versions of worms (Christodorescu and Jha, 2004), and there are numerous examples of obfuscation techniques designed to avoid detection (Mohanty, 2004 published online *http://www.hackingspirits.com/eth-hac/papers/whitepapers.asp*; Ször, 2005). Christodorescu *et al* (2005) describes the design and implementation of a *malware normalizer* that undoes the obfuscations performed by a malware writer. The experimental evaluation demonstrates that a *malware normalizer* can drastically improve detection rates of

commercial malware detectors. Some advanced pattern recognition methods are applied in a few commercial scanners to distinguish a virus from a non-virus (IBM Antivirus Research "Digital Immune System" available at *http://www.research.ibm.com/antivirus/*; Chu *et al* "Virus A Retrospective" available at *http://cse.stanford.edu/class/cs201/projects-00-01/viruses/anti-virus.html/*.). Kruegel *et al* (2004) presents novel binary analysis techniques for static disassembly of obfuscated Intel x86 binaries based on control flow graph information and statistical methods. Udupa *et al* (2005) shows that it may be possible to bypass much of the effects of some code obfuscations by a combination of static and dynamic analyses. However, to the best of our knowledge, to this date little research seems to have been performed on detecting obfuscated malware.

In the remainder of this paper, the term "polymorphic malware" means specifically obfuscated malware. Our work for polymorphic or obfuscated malware detection is based on the assumption that an original malware, M, contains a malicious sequence of API calls, S. A variant of the original malware that is obtained by obfuscation retains the functionality of the original malware and contains a set of system calls S'. The problem is to find the similarity measure between S and S'. Since the original functionality is preserved, it is reasonable to assume that the "difference" between S and S' is not "large". We implement an antivirus scanner, SAVE (Static Analyzer for Vicious Executable), to prove our assumption. Section 2 and 3 introduce the obfuscation techniques. Sections 4, 5 and 6 discuss how SAVE works including the description of PE binary parser and the similarity measure algorithms. The performance study is presented in Section 7. Finally, Section 8 presents our conclusions and proposes our idea for future work.

2. MALWARE USED FOR ANALYSIS

In this paper, several recent viruses (executables) were used for analysis; specifically, we employed four viruses and their variants. The description of the viruses below is given based on the payload, enabling vulnerability, propagation medium, and the systems infected.

W32.Mydoom: A mass mailing worm and a blended back door that arrives with as an attachment with file extensions .bat, .cmd, .exe, .pif, .scr or .zip according to the report from Symantec company, a well known anti virus software company. The pay load performs a denial of service against www.sco.com and creates a proxy server for remote access using TCP ports 3127 through 3198. **W32.Mydoom** infects all Windows systems.

W32.Blaster: Exploits windows DCOM RPC vulnerability using TCP port 135. The pay load launches a denial of service attack against windowsupdate.com, might cause systems to crash and opens a hidden remote cmd.exe shell. **W32.Blaster** propagates via TCP ports 135, 4444 and UDP port 69 and infects only Windows 2000 and XP.

W32.Beagle: A mass mailing worm blended with a back door. The worm contains large scale email with extensions, .wab, .htm, .xml, .nch, .mmf, .cfg, .asp, and etc, according to the report from Symantec Company. Uses its own SMTP engine, TCP port 2745 to spread and also tries to spread via file sharing networks like Kazza. **W32.Beag** infects all Windows systems.

Win32.Bika: According to virus library it is a harmless per-process memory resident parasitic Win32 virus. It infects only Win32 applications. The virus writes itself to the end of the file while infecting. Once the host program is infected it starts the virus hooks "set current directory" Win32 API functions, SetCurrentDirectoryA, SetCurrentDirectoryW that are imported by the host program and stays as a background thread of the infected process, and then infects files in the directories when the current directory is being changed. The virus does not manifest itself.

3. OBFUSCATION

In its simplest form, obfuscation is obscuring some information such that another person cannot construe its true meaning. This is certainly true for code obfuscation where the objective is to hide the underlying logic of a program.

According to the research of Collberg *et al* (2002), code obfuscation has been compared to code optimization where code optimization is a transformation that minimizes a program's certain metric such as execution time or execution size, while code obfuscation has the additional requirement that the code transformation also maximizes obscurity. When we optimize for speed we generally try to take advantage of hardware pipelines, memory buffers, etc., while leaving the program essentially the same. Any optimization that changes the program's functionality or logic cannot be applied blindly and is generally avoided.

Obfuscation has also been applied to program watermarking and is a well-known technique to prevent reverse engineering, for example Krishnaswamy *et al* (2002) proposed an approach on software watermarking based on obfuscation. In general, obfuscating a program to prevent reverse engineering is similar to a classic cryptography game: you try to make reversing your obfuscation hard enough such that it is impractical to attack. Given enough time and resources any obfuscation can be reversed; but as long as it requires excessive amounts of resources and effort to be infeasible, most will not bother and so it may be considered secure. By obfuscating one can prevent another individual from gaining knowledge about his program. With respect to malware, code obfuscation is an appealing technique to hinder detection. A simple obfuscation technique may render a known virus completely invisible to conventional scanners with very little effort on the part of the virus writer.

Applying an obfuscation transformation to a program has the advantage that it is essentially self-decrypting encryption. The code is rendered incomprehensible while still remaining a viable program.

3.1 Obfuscation Theory

An elegant method to describe the code obfuscation problem was presented by Collberg and Thomborson (2002). Given a program P and a set of obfuscation transformations T we want to generate program P' such that:

- P' retains the functionality of P
- P' is difficult to reverse engineer
- P' performs comparably to P, i.e. the cost of obfuscation is minimized

Obfuscation transformations need to be resilient. After applying transformation Ti to program statement Sj and generating an obfuscated statement Sj', it must be prohibitively hard to build an automated tool that can generate Sj from Sj'. As long as the meaning or logic of the transformed statement is sufficiently ambiguous a series of transformations will deter reverse engineering.

3.1.1 Data

Data obfuscation changes the look of a program by modifying the constants or encapsulated bits of data. An example would be to split a string hello world into smaller strings, such as he, ll, and o. Another method would be to separate a Boolean variable into two integers and use comparisons between the two to emulate the True / False properties of the original.

In general, complex data transformations require the addition of "helper code" if the original functionality is to be maintained. In the example above, we would need to generate code to concatenate the small strings together to get the original "hello world" before using it.

3.1.2 Control Flow

Control flow transformations focus on obscuring how the program runs. For example, inserting junk code into a program changes its appearance considerably but does not change the logic. A more complex example would be to use global pointers for control flow. If we use pointers p and q and insert a statement like *if* ($p == q$) then it is nearly impossible to determine if this statement is true or false using static analysis. Such a combination of pointers and control flow statements is considered opaque because of the difficulty inherent in pointer alias analysis.

This type of obfuscation is particularly appealing to malware authors because of its strength. We see control flow transformations implemented in polymorphic and metamorphic engines where the code changes with each host infected.

3.1.3 Other techniques

Data and control flow are not the only techniques that can be used to obscure a program's meaning or prevent reverse engineering. Many software authors make use of antidisassembly and anti-debugging techniques to hinder analysis. In general, these are "tricks" that slow down automated tools such as disassembles. Byte code scramblers are also used to obfuscate strongly typed bytecode such as Java's. All of these techniques, combined with a generous helping of data and control flow obfuscation, help make code analysis exorbitantly difficult.

3.2 Classification

For simplicity we have separated the obfuscation techniques into six general categories. Because of the complexity in implementing and detecting pointer aliases we gave them their own category. As a general rule the complexity and robustness of the technique increases the greater the type. Straight control flow obfuscation is (in general) not as robust as both data and control flow obfuscation together. These types assume a low level language such as x 86 assemblies.

Type 1: Null operations and dead code insertion

NOPs are inserted into the code. There is virtually no modification to data or control flow. An example of a type 1 transformation is presented in Figure 1 below. On the left we have the original code and on the right we have the modified code with null operations inserted every two lines.

Inserting null operation is similar to inserting white space in a document: it may take longer to read but no more difficult as the content remains the same.

Type 2: Data modification

Some data obfuscation transformation is applied, such as string splitting or variable type replacement. For example, we could replace a Boolean variable with two integers. If they are equal,

Original code	After transformation

`mov eax, -44(ebp)`
`mov -44(ebp), ebx`
`sub 12, esp`
`lea -24(ebp)`
`push eax`
	`mov eax, -44(ebp)`
	`mov -44(ebp), ebx`
	nop
	`sub 12, esp`
	`lea -24(ebp)`
	nop
	`push Eax`

Figure 1: Example of null operation insertion

Original code and meaning		
cmpb	0, x	If (x == true)
je	.sub	goto sub
Transformed code and meaning		
mov	a, eax	If (a < b)
cmpl	b, eax	goto sub
jge	.sub	

Figure 2: Example of data flow obfuscation

the statement is true, otherwise it is false. In the example above (Figure 2), ***x*** is a Boolean variable and ***a*** and ***b*** are integers. The code on the left is the original control flow and the code on the right performs exactly the same but has a different signature.

Type 3: Control flow modification
Control flow transformations are applied. Code is swapped around and jump instructions are inserted. For example, we could copy the contents of a subroutine to another location in the file and add jumps to and from the subroutine. The code would function exactly the same but look quite different. In Figure 3 below, three lines of code have been shifted to some location (denoted as [shift]) and helper code has been inserted.

Type 4: Data and control flow obfuscation
We pull out all the stops and combine data and control flow transformations. At this level junk code is inserted and variables can be completely replaced with large sections of needless code. For example, we can modify all integer variables as above and transpose the program's entry point as in Figure 4.

Type 5: Pointer aliasing
The final step is to introduce pointer aliasing. Variables are replaced with global pointers and functions are referred to by arrays of function pointers. This type of transformation is relatively easy to implement using high level languages that allow pointer references but tricky (at best) using

Original code		After transformation	
cmp	24, eax	**Jmp**	**[shift]**
jne	.sub	**Nop**	
sub	12, eax	**Nop**	
push	eax	Push	Eax
	…		…
		Cmp	24, eax
		Jne	.sub – **[shift]**
		Sub	12, eax
		Jmp	**–[shift]**

Figure 3: Example of control flow obfuscation

Original code	After transformation
Cmp 24, eax	**jmp [shift]**
Jne .sub	**nop**
Sub 12, eax	**nop**
Push eax	push Eax
…	…
	mov 24, eax
	cmpl b, eax
	jle .dead_code
	jne .sub – **[shift]**
	sub 12, eax
	jmp **-[shift]**

Figure 4: Example of data and control flow obfuscation

assembly languages. Pointer aliasing can be as simple as changing $\boldsymbol{a} = \boldsymbol{b}$ into $*\boldsymbol{a} = **\boldsymbol{b}$ or as complex as converting all variables and functions into an array of pointers referenced by pointers to pointers.

3.3 Obfuscation Used in this Project

In our research we discovered that most commercial virus scanners can be defeated with very simple obfuscation techniques. For example, simple program entry point modifications consisting of two extra jump instructions effectively defeated most scanners. Therefore, we only used the bare minimum level of obfuscation needed to prevent detection. Our goal was to show how trivial it is to modify recent malware to defeat existing scanning techniques using only the compiled executable and a few tools.

The binary code is disassembled into a more readable format so that we may understand what the program is doing. Someone with fore-knowledge about the malware need not spend so much time analyzing the program. Once we have the disassembled program and have studied it, we pick out an area to attack. The first target when applying a control flow transformation is to attack the program's entry point. But when using a data transformation we generally have to take a guess. We decide where and what modifications need to be performed and change the binary file directly, using the disassembled version as a guide or map. Once all modifications have been made, the file is examined using the anti-virus scanners.

All variants with the exception of the *MyDoom* virus were generated using of the shelf hex editing tools. We were fortunate enough to have a copy of the *MyDoom.A* source code and made all our modifications using the *Microsoft Visual* development suite.

4. SYSTEM ARCHITECTURE

Our approach is performed directly on PE (Windows Portable Executable) binary code, which is designed as a common file format for all flavours of windows, on all supported CPUs. Our approach is structured into two major steps illustrated in Figure 5.

Firstly, a suspicious PE file is optionally decompressed if it is compressed by using a third party binary compress tool, for example USP Shell, created by Oberhumer (2000), and then passed via a PE binary parser. The output of PE binary parser is a sequence of Windows APIs. An API calling

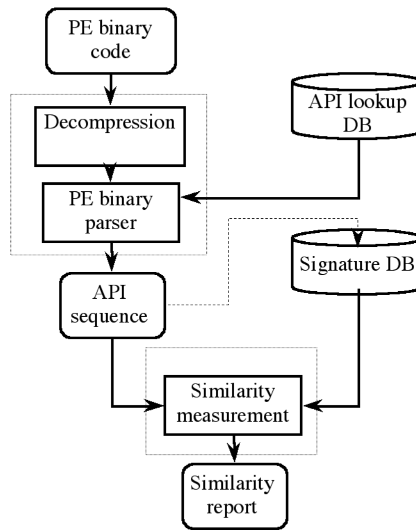


Figure 5: The architecture of our anti-virus system

sequence consists of a group of 32-bit global ids which represent the static calling sequence of corresponding API functions. The signature of known Win32 virus is also an API sequence generated by the same technique. The signature is stored in a signature database together with other known Win32 virus signatures.

To determine whether a suspicious PE file, s, is a variant of a particular virus, v, or not, we pass the suspicious API sequence together with every signature sequence of v from the signature database through a similarity measure module, which calculates the similarity between the suspicious API sequence and API sequence of v. The output is compared to a threshold to determine whether s is a variant of v. If the similarity value is larger than the threshold, a positive flag is raised. In order to report a non virus, s should be measured against all v in the signature database, and get no positive output it is observed.

5. BINARY PARSER

We developed a PE binary parser instead of using a third party disassembler, in order to improve the overall system performance. Most third party disassemblers output a text file for the disassembled code, which should be parsed to extract the necessary features for further analysis. The text processes greatly degrade the overall system performance. Furthermore, we are only interested in Win32 API calling sequence in a PE file instead of all disassembled instructions, which the third party disassemblers spend much time to process. We decided to develop a special purpose PE binary parser to reduce the intermediate text process and get only the necessary information from a PE file, since a virus scanner is highly speed sensitive.

The major structure of a PE file is illustrated in Figure 6. Every PE file begins with a small MS-DOS executable. After the DOS header is a PE header, which includes a main file header and an optional header to specify how the PE file is stored. Immediately following the PE header is the section table. The section table provides information about all sections names, locations, lengths and characteristics. There is at least one section following the section table. A PE section represents

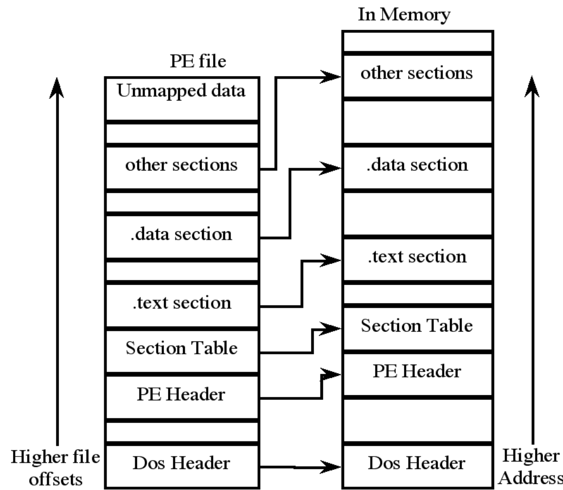


Figure 6: Overview of PE file structure

a code or data of some sort. While code is just code, there are multiple types of data. Besides read/write program data, other types of data in sections include API import and export tables, resources and relocations.

Our PE parser is built to extract static API calling sequence. Three major steps should be performed to achieve that goal. First of all, we locate the Imported Address Table (IAT), which contains the pointers to the imported API hints and names. Then generate binary lookup tree from all imported API. Secondly, we scan code section(s) to extract the CALL instructions and their target addresses. Then search the target address in the binary lookup tree to find the corresponding API. Finally, we map the API name with its module name to a 32 bits unique global API id by a lookup table.

5.1 Locating the Imported Address Table

Within a PE file, there is an array of data structures, one per imported module. Each of these structures gives the name of the imported module and points to an array of function pointers. The array of function pointers is known as IAT, discussed by Pietrek (2002). To locate the IAT we should firstly extract the optional header, which appears at the end of PE header. At the end of the optional header, there is a DataDirectory array, which is the address book for important locations within the executable. We can find the DataDirectory entry for imports by specifying the index IMAGE_DIRECTO RY_ENTRY_IMPORT in the DataDirectory array. The imports entry points to an array of IMAGE_ IMPORT_DESCRIPTOR structure. There is one such structure for each imported module. Figure 7 shows a PE file importing some APIs from KERNEL.DLL.

After getting the IAT, for each API we calculate its relative virtual address (RVA) by RVA(APIi) = address(import descriptor) + image_base + offset(APIi), where address is the address of import descriptor, image_base is specified in the PE header. All the relative virtual address should be biased by the image base, shown in the format specification from Microsoft. RVA is a very important feature that allows us to find the target API name from the target address of CALL instruction. We store the set of API names, API module names and the corresponding RVAs in a binary tree for efficient looking up.

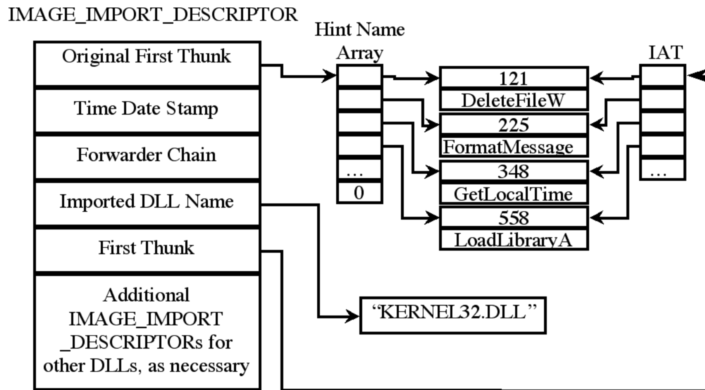


Figure 7: IMAGE_IMPORT_DESCRIPTOR

5.2 Scanning Code Sections

By referring to the section table immediately following the PE header we can locate all the code sections in the PE file. For each code section, we scan the whole section for CALL instructions. A piece of machine codes from malware Mydoom and the corresponding assembly codes are shown in Figure 8. The CALL instruction we are interested in is FF1504104A00. The corresponding assembly instruction could be checked from Intel’s instruction reference. The instruction in this example is ‘CALL dword ptr [004A1004]’, where 004A1004 is the RVA of the target API. We then search the RVA in the look up binary tree to find the corresponding API and its module, which is ‘ADVAPI32.RegOpenKeyExA’. After scanning the whole code section we get a set of strings, which stores the names of the called APIs and the names of their modules.

5.3 Mapping API

We map each API name with its module name to a global id through a lookup table, which is a fixed table storing the most frequently used Win32 DLL and their APIs and the corresponding pre-assigned id. The global id is a 32-bit integer. The first (most significant) 12 bits of the integer represent a particular Win32 dynamic link library, and the rest 20 bits specify a particular API in this module. For example, the API ‘ADVAPI32.RegOpenKeyExA’ described in the last section is encoded as 0x00500E13. By using integer representation, we can avoid the costly string comparison operations. It will be discussed in detail in the next section.

6. SIMILARITY MEASURE

By using the mechanism described above, we construct the API sequence of a suspicious PE binary file. Let’s denote it *Vu* (vector of unknown). The API sequence of a known virus is called signature.

```
:004A40F5 03C6         add eax, esi
:004A40F7 50           push eax
:004A40F8 FF1504104A00 CALL dword ptr [004A1004]
:004A40FE 85C0         test eax, eax
```

Figure 8: A sample of machine code from the code section of MyDoom. The corresponding API is ADVAPI32.RegOpenKeyExA.

Let's denote the signature sequence *Vs* (vector of signature). To decide if the new executable file is an obfuscated version of the virus represented by *Vs*, we measure the similarity between *Vs* and *Vu*. One of the most common measures is the Euclidean distance (1).

$$D(Vs,Vu) = \left[\sum_{i=1}^{\min(|Vs|,|Vu|)} (v_{s_i} - v_{u_i})^2 \right]^{1/2} \quad (1)$$

However, Euclidean distance may not be a good similarity measure at times. For example, consider below three vectors:

V1 = (1, 9, 1, 9, 1, 9, 1, 9, 1, 9, 1, 9)
V2 = (9, 1, 9, 1, 9, 1, 9, 1, 9, 1, 9, 1)
V3 = (5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5)

Most people would perceive *V1* and *V2* as the closer pair of sequence.

If the Euclidean distance is used, however, the distance between *V1* and *V2* is greater than that between *V2* and *V3*, as *D*(*V1*, *V2*) = 27.71 while *D*(*V2*, *V3*) = 13.86

We use a sequence alignment technique similar to the algorithm proposed by Wilson (1998) to solve this problem. Consider the following two sequences: "WANDER" and "WADERS", the best alignment should be

```
WANDER-
WA-DERS
```

The optimal alignment algorithm can be conceptualized by considering a matrix with the first sequence placed horizontally at the top and the second sequence placed vertically on the side. Each position in the matrix corresponds to a position in the first and second sequence. Any alignment of the sequences corresponds to a path through the grid, as in Figure 9.

Using paths in the grid to represent alignments provides a method of computing the best alignments. The score of the best path up to that position can be placed in each cell. Beginning at the top left cell, the scores are calculated as the sum of the score for the element pair determined by the score of the row and column heading (0 for mismatches and 1 for matches) and the highest score in the grid above and to the left of the cell.

Figure 10 shows the alignment algorithm step by step. Let's take a deeper look at shadowed 4 and 3. 4 is generated as a max score in left above matrix plus the score for a matching, that is 3 plus 1. 3 is calculated as a max score in left above matrix plus score for mismatching, that is 3 plus 0.

In our case, API sequences *Vs* and *Vu* are inserted with some zeros to generate *Vs'* and *Vu'*, which have optimal alignment.

	W	A	N	D	E	R
W	X					
A		X				
D				X		
E					X	
R						X
S						

Figure 9: Sequence alignment represented by path through grid

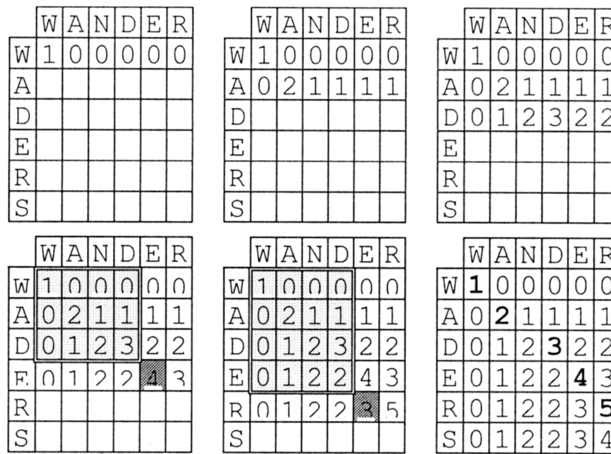


Figure 10: An example of realignment algorithm

Next, we apply the traditional similarity functions on *Vs'* and *Vu'*. Cosine measure (2), extended Jaccard measure (3) and Pearson correlation measure (4) are the popular measures of similarity for sequences. The cosine measure captures a scale-invariant understanding of similarity.

$$S^{(C)}(Vs',Vu') = \frac{Vs'^{T}Vu'}{\|Vs'\|_2 \cdot \|Vu'\|_2}, \text{ where } \|V\|_p = \left[\sum_i |v_i|^p\right]^{\frac{1}{p}} \tag{2}$$

The extended Jaccard measure is computed as (3), which measures the ratio of the number of shared attributes of *Vs'* and *Vu'* to the number of attributes possessed by *Vs'* or *Vu'*.

$$S^{(J)}(Vs',Vu') = \frac{Vs'^{T}Vu'}{\|Vs'\|_2^2 + \|Vu'\|_2^2 - Vs'^{T}Vu'} \tag{3}$$

The Pearson's correlation measure is defined as (4), which measures the strength and direction of the linear relationship between *Vs'* and *Vu'*.

$$S^{(P)}(Vs',Vu') = \frac{1}{2}\left(\frac{(Vs' - \overline{Vs'})^{T}(Vu' - \overline{Vu'})}{\|Vs' - \overline{Vs'}\|_2 . \|Vu' - \overline{Vu'}\|_2} + 1\right) \tag{4}$$

The reason to utilize three different measures is that none of them can output the effective results for all sequence measures. Table 1 illustrates three examples to demonstrate how these measures mutually correct each other.

The first example shows a shortened version of the most common cases in our experiments. Tiny changes in the API sequence indicate that two files perform very similar functions, that is to say the suspicious executable is an obfuscated virus. The effective output in this case is 1.0. In this case the cosine measure gives the best output. The second and third examples show two exceptions that cannot be measured correctly by the cosine measure. The second shows two different sequences, whose effective output is 0.0. The Jaccard measure outputs the best result. In the third example the Pearson measure gives the expected result.

In the current version, we calculate the mean value of $S^{(C)}(Vs', Vu')$, $S^{(J)}(Vs',Vu')$, and $S^{(P)}(Vs', Vu')$. For a particular measure between a virus signature and a suspicious binary file, let's denote this

$[Vs'],[Vu']$	$S^{(C)}(Vs',Vu')$		$S^{(J)}(Vs',Vu')$		$S^{(P)}(Vs',Vu')$	
[1,2,3,4,5,6], [1,2,3,9,5,6]	0.9316	√	0.8160	?	0.8631	?
[1,2,1,2,1,2], [8,9,8,9,8,9]	0.9656	×	0.2097	√	1.0000	×
[1,1,1,1,1,2], [1,1,1,1,1,100]	0.6832	×	0.0204	×	1.0000	√

√ indicates the best output, × indicates the false output,
and ? indicates the acceptable but not the best output

Table 1: Mutual correction between measures

mean value as $S^{(m)}(Vs'_i, Vu')$, which stands for the similarity between virus signature *I* and suspicious file.

Our similarity report is generated by calculating the $S^{(m)}(Vs'_i, Vu')$ value for every signature in the signature database. The index of the largest entry in the similarity report indicates the most possible virus the suspicious file intends to be. Let's denote the index i_{max}. By comparing this largest entry with a threshold, we can make a decision: if the largest entry is higher than the threshold, then the suspicious file is the virus in the signature database with the index i_{max}; otherwise the suspicious file is not a known virus (or perhaps a new virus that doesn't yet have a signature in our database). In our experiment, threshold .90 works quite well.

By utilizing the above algorithm we implemented our polymorphic virus scanner SAVE1 (Static Analyzer for Vicious Executable). We also created another version's scanner SAVE2. The major difference between SAVE1 and SAVE2 is that they use the different signatures. SAVE1 uses the API sequence of a Win32 PE as a signature. Meanwhile SAVE2 uses the imported API set as a signature. The corresponding similarity measure methods are different too. SAVE1 uses sequence measure, SAVE2 uses the set match.

7. PERFORMANCE STUDY

Several recent Win32 viruses were used for performance evaluation: Mydoom, Blaster, Beagle, Bika from Symantec company. For each virus we created a set of polymorphic versions by the obfuscation techniques described in Section 3. For example, Mydoom V1 and Beagle V1 are created by modifying data segment; Mydoom V2 and Beagle V2 are created by modifying control flow; Bika V1 is created by inserting dead code. We then scan these polymorphic versions by using eight different virus scanners and our new scanners. Table 2 shows the experimental results. As can be seen from the last two columns, our scanners, SAVE1 and SAVE2, perform the most accurate detection.

We also considered a suite of benign PE files (All Win32 PE under the directories of 'Windows', and 'Program Files'). We executed SAVE1 and SAVE2 on these benign programs; our scanner reported "negative" in all cases.

In our example, no false positive was found, however, the sensitivity of the scanner is quite different. The detection sensitivity of SAVE1 is better than SAVE2, shown in Figure 11. The horizontal axis represents the file index. In this experiment we used more than three hundred benign Win32 PE files, shown in the left side of the vertical line. And from left to right the file size increases. We also used 15 versions' of Mydooms by different obfuscated techniques, shown in the right side of the vertical line. The detection outputs by SAVE2, shown in Figure 12(b), are scattered

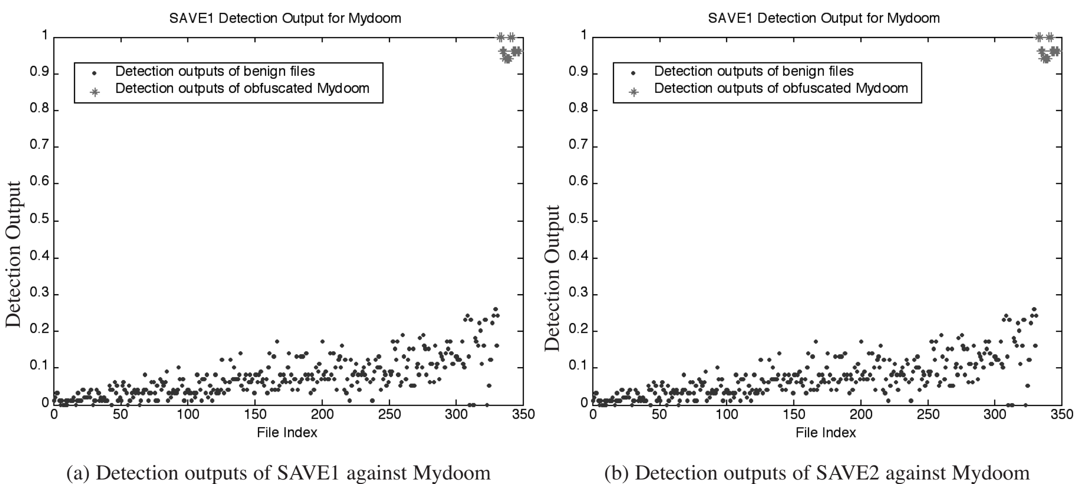
	N	M^1	M^2	D	P	K	F	A	S1	S2
Mydoom.A	√	√	√	√	√	√	√	√	√	√
Mydoom.A V1	×	√	√	×	×	√	√	×	√	√
Mydoom.A V2	√	×	×	×	×	×	×	×	√	√
Mydoom.A V3	×	×	×	×	×	×	×	×	√	√
Mydoom.A V4	×	×	×	×	×	×	×	×	√	√
Mydoom.A V5	×	?	×	×	×	×	×	×	√	√
Mydoom.A V6	×	×	×	×	×	×	×	×	√	√
Mydoom.A V7	×	×	×	×	×	×	×	×	√	√
Bika	√	√	√	√	√	√	√	√	√	√
Bika V1	×	×	×	√	×	√	√	√	√	√
Bika V2	×	×	×	√	×	√	√	√	√	√
Bika V3	×	×	×	√	×	√	√	√	√	√
Beagle.B	√	√	√	√	√	√	√	√	√	√
Beagle.B V1	√	√	√	×	×	√	√	×	√	√
Beagle.B V2	√	×	×	×	×	×	×	×	√	√
Blaster	√	√	√	√	√	√	√	√	√	√
Blaster V1	×	√	√	√	√	√	√	×	√	√
Blaster V2	√	√	√	×	×	√	√	×	√	√
Blaster V3	√	√	√	√	√	×	×	×	√	√
Blaster V4	×	×	×	×	×	√	√	×	√	√

N – Norton, **M^1** – McAfee UNIX Scanner, **M^2** – McAfee, **D** – Dr. Web, **P** – Panda, **K** – Kaspersky, **F** – F-Secure, **A** – Anti Ghostbusters, **S1** – NMT developed **S**tatic **A**nalyzer for **V**icious **E**xecutable, which uses the methods described in Section 4. **S1, S2** – NMT developed **S**tatic **A**nalyzer for **V**icious **E**xecutable, which uses the API bag as signature.

Table 2: Polymorphic Malware detection using different scanners

much wider than that of SAVE1. It is obvious that SAVE1 is more effectively to distinguish benign files from obfuscated malicious files.

Since SAVE2 is much faster than SAVE1. The performance comparison is shown in Figure 12. We used SAVE2 as a pre-scanner to perform extremely efficient batch scanning. For the files have a highly suspicious output, we scan them by SAVE1 again to generate the highly confident output.



(a) Detection outputs of SAVE1 against Mydoom (b) Detection outputs of SAVE2 against Mydoom

Figure 11: Sensitivity comparison between SAVE1 and SAVE2

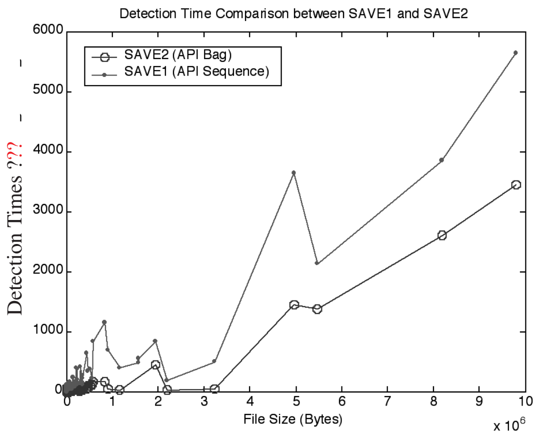


Figure 12: Speed comparison between SAVE1 and SAVE2. The experiment is performed on more than three hundred PE files having the various sizes from 10K bytes to 10M bytes.

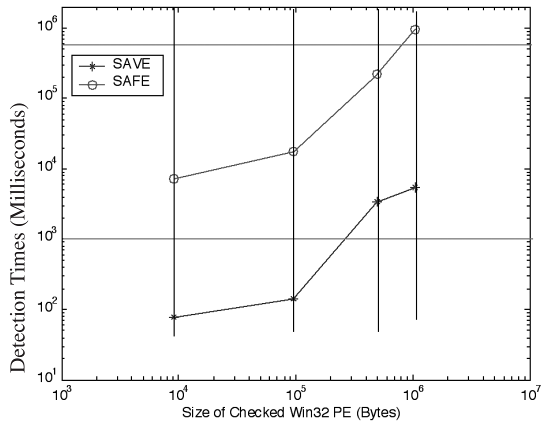


Figure 13: Performance comparison on four executables. (The values near curves are detection time, in millisecond)

We also compare our scanner with another static executable scanner, SAFE (Static Analyzer for Executables), which was introduced by Christodorescu and Jha (2003), and was claimed to be able to detect obfuscated versions of virus. We did experiments on an environment, Intel 1 GHz, 1 GB of RAM plus MS Windows 2000, which is very similar to the environment (AMD Athlon 1GHz, 1 GB of RAM plus MS Windows 2000) described by Christodorescu and Jha (2003). We also ran our scanner against the same executable codes in their experiments.

Figures 13 and 14 compare the performance between SAVE1 and SAFE. The higher curve shows the total time of SAFE for checking four benign Win32 PE files, which are listed in the bottom of the diagram. The corresponding performance of our detector, SAVE1, is shown in the lower curve. As can be seen, SAVE1 is about one hundred times faster than SAFE for checking middle size PE files. As shown in Figure 13, with respect to Win32 PE size, the detection time taken by SAFE increases at a much higher rate than our detector.

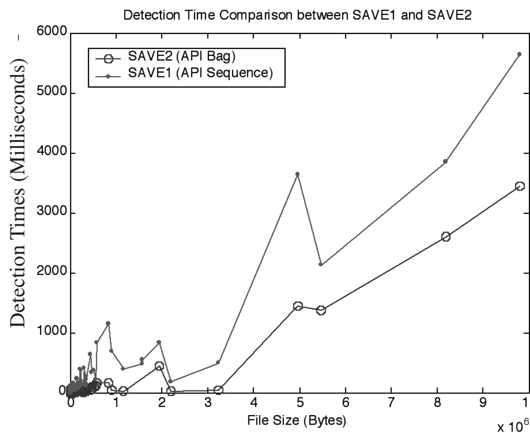


Figure 14: Detection time taken by SAFE increases in a much higher rate than that of SAVE with respect to size

8. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, a methodology for composing signatures of Win32 PE malicious codes is presented aimed at supporting polymorphic malicious code detection. The key assumption is that to preserve its functionality, a polymorphic malware should contain a sufficiently similar API calling sequence. We described in detail the implementation of a PE scanner for creating API calling sequence as the signature, and an algorithm for computing similarity measure for signature comparison. Experiments (performed before May 2005) on a large set of polymorphic malware showed that our scanner, SAVE, is accurate and efficient in detecting polymorphic malware.

In the future, we hope to concentrate on the optimizations of the signature creation process. Since not all the APIs are meaningful in profiling a malicious code, we plan to investigate developing a new method to pick up the most important API calls as signature for the malicious code. Using a smaller vector as a signature may significantly improve the overall performance. We also plan to optimize the realignment algorithm, which is now a bottle neck of our algorithm.

ACKNOWLEDGEMENT

The authors would like to acknowledge New Mexico Tech's ICASA (Institute for Complex Additive Systems Analysis) division and the Information Technology Program for partial support of this work. We would also like to acknowledge Mr Kartikeeyan Ramamurthy for his insightful suggestions and Dr. Xie Tao for his initial studies of obfuscation that stimulated our interest in the subject.

REFERENCES

ANTIVIRUS RESEARCH (1999): Digital immune system, http://www.research.ibm.com/antivirus/.

BERGERON, J., DEBBABI, M., ERHIOUI, M. and KTARI, B. (1999): Static analysis of binary code to isolate malicious behaviors. *Proc. The IEEE 4th International Workshops on Enterprise Security (WETICE'99)*, Stanford University, California, USA.

CHESS, D.M. and WHITE, S.R. (2000): An undetectable computer virus. *Proc. Virus Bulletin Conference, 2000*.

CHRISTODORESCU, M. and JHA, S. (2003): Static analysis of executables to detect malicious patterns. *Proc. The 12th USENIX Security Symposium*, Berkeley, CA.

CHRISTODORESCU, M. and JHA, S. (2004): Testing malware detectors. *Proc. the ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04)*. 34–44. Boston, MA, USA, July 2004. ACM SIGSOFT, ACM Press.

CHRISTODORESCU, M., KINDER, J., JHA, S., KATZENBEISSER, S. and VEITH, H. (2005): Malware normalization. *Technical Report #1539*, Computer Sciences Department, University of Wisconisin. Madison. Nov. 2005. http://www.cs.wisc.edu/wisa/papers/tr1539/.

CHU, S.S., DIXON, B., LAI, P., LEWIS, D., VALDES, C. and ROBERTS, E. (2001): Virus a retrospective, Stanford University: Class project for Computers, Ethics, and Social Responsibility. http://cse.stanford.edu/class/cs201/projects-00-01/viruses/anti-virus.html.

COHEN, F. (1987): Computer viruses: Theory and experiments. *Computers and Security*, 6:22–35.

COLLBERG, C. and THOMBORSON, C. (2002): Watermarking, tamper-proofing, and obfuscation – Tools for software protection, *IEEE Transactions on Software Engineering* 28(8): 735–746.

DETRISTAN, T., ULENSPIEGEL, T., MALCOM, Y. and UNDERDUK, M. (2003): Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61).

ESCHELBECK, G. (2003): Worm and virus defense: How can we protect the nation's computers from these threats? Before the Subcommittee on Technology, Information Policy, Inter-governmental Relations and the Census House Government Reform Committee.
http://a257.g.akamaitech.net/7/257/2422/10may20041230/www.access.gpo.gov/congress/house/pdf/108hrg/92654.pdf

INTEL CORPORATION (2004): IA-32 Intel® Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference.

KRISHNASWAMY, S., KWON, M., MA, D., SHAO, Q. and ZHANG, Y. (2000): Experience with software watermarking. *Proc. 16th Annual Computer Security Applications Conference (ACSAC'00)*, 308–316.

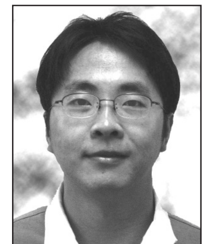
KRUEGEL, C., ROBERTSON, W., VALEUR, F. and VIGNA, G. (2004): Static disassembly of obfuscated binaries. *Proc. the 13th Usenix Security Symposium (USENIX'04)*, San Diego, CA, USA, August. USENIX.

MICROSOFT CORPORATION (2004): Portable Executable Formats, Formats specification for Windows.

MOHANTY, D. (2004): Anti-virus evasion techniques and countermeasures. Published online at http://www. Hacking spirits.com/eth-hac/papers/whitepapers.asp.

OBERHUMER, M. and MOLNAR, L. (2000): http://upx.sourceforge. Net/.
PIETREK, M. (2002): Inside Windows: An in-depth look into the Win32 portable executable file format. *MSDN Magazine* 17(2–3).
STREHL, A. and GHOSH, J. (2000): Value-based customer grouping from large retail datasets, *Proc. SPIE Conference on Data Mining and Knowledge Discovery*, Orlando, 4057: 33–42.
SUNG, A., XU, J., CHAVEZ, P. and MUKKAMALA, S. (2004): Static analyzer for vicious executables (SAVE), *Proc. The 20th Annual Computer Security Applications Conference*. 326–334.
SYMANTEC COOPERATION (2005): http://securityresponse.Symantec.com.
SZÖR, P. (2005): Advanced code evolution techniques and computer virus generator kits. *The Art of Computer Virus Research and Defens*. Addison-Wesley Professional, 1st edition, February.
UDUPA, S.K., DEBRAY, S.K. and MADOU, M. (2005): Deobfuscation: Reverse engineering obfuscated code. *Proc. The 12th Working Conference on Reverse Engineering*, 45–54.
VIRUS LIBRARY (2004): http://www.viruslibrary.com/ virusinfo/Win32.Bika.htm
WILSON, W.C. (1998): Activity pattern analysis by means of sequence-alignment methods, *Environment and Planning*, A(30):1017–1038.

BIOGRAPHICAL NOTES


Jianyun Xu

Jianyun Xu received his B.Sc. in computer science and engineering in 1999 from Zhejiang University, Hangzhou, P.R.C. and M.Sc. and Ph.D. in computer science in 2005 from New Mexico Tech, Socorro, USA.

In January 2006, he joined the Identity Service Group, Microsoft Corporation, Redmond, USA, where he is currently a software design engineer working on network security.

His research interests include fraud detection, malware detection, network security, steganography, steganalysis, watermarking, imaging processing and computer graphics.

Andrew H. Sung

Andrew H. Sung is a professor and the chairman of the Computer Science Department of New Mexico Tech. He received his Ph.D. in 1984 from the State University of New York at Stony Brook. His current research interests include information security, application algorithms, and soft computing and its engineering applications.

Srinivas Mukkamala

Srinivas Mukkamala is a senior research scientist with New Mexico Tech's Institute for Complex Additive Systems Analysis and Adjunct faculty at the Department of Computer Science, New Mexico Tech. He is a technical manager of the information assurance research group at New Mexico Tech.

He received his B.E. in Computer Science and Engineering from University of Madras in 1999, M.Sc. in Computer Science and Ph.D. from New Mexico Tech.

He is a frequent speaker on information assurance in conferences and tutorials. He is currently working in the areas of information assurance and security and has over 65 publications in the areas of information assurance, digital forensics, applied soft computing and data mining.

Qingzhong Liu

Qingzhong Liu received his B.E. in 1993 from Northwestern Polytechnical University and M.Sc. in 1997 from Sichuan University of China. Currently he is pursuing his Ph.D. in the Computer Science Department of New Mexico Tech, USA. His research interests include data mining and computational intelligence for information security, bioinformatics, and image analysis.