Polymorphic Malicious Executable Scanner by API Sequence Analysis

J-Y. Xu, A. H. Sung, P. Chavez, S. Mukkamala
Department of Computer Science, New Mexico Tech
{dennisxu, sung, pchavez, srinivas}@cs.nmt.edu

Abstract

The proliferation of malware (viruses, Trojans, and other malicious code) in recent years has presented a serious threat to enterprises, organizations, and individuals. Polymorphic (or variant versions of) computer viruses are more complex and difficult than their original versions to detect, often requiring antivirus companies to spend much time to create the routines needed to catch them. 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 behavior of a piece of particular code. The analysis is carried out directly on the PE (portable executable) code. It is achieved in two major steps: construct the API calling sequences for both the known virus and the suspicious code, and perform a similarity measurement between the two sequences after a sequence realignment operation is done. Favorable experimental results are obtained and presented.

1. Introduction

Internet connectivity that is vital to enterprises and individuals may also expose critical information to adversaries and result in frequent attacks. One of the greatest threats to cyber security has come from automatic, pre-scanned, self-propagating attacks such as blended viruses and worms. These attacks scan at random until they are able to place a malign program on the victim server using a maliciously crafted request. The program uses the now-infected server as a base from which to attack other vulnerable servers. The end result is exponential growth in the number of attacked servers leading to load-induced performance degradation or network failure. Though organizations have a wide variety of protection mechanisms (firewalls, antivirus tools, and intrusion detection systems) against cyber attacks, recent hybrid, and blended malware like Sasser, Blaster, Slammer, Nimda and CodeRed worked their way past the current security mechanisms. Since the number and intensity of malware attacks is on the rise, computer security companies, researchers, and users are hard-pressed to find new services to help thwart or defend against such assaults.

Theoretical studies on virus detection have revealed that there is no algorithm that can detect all types of viruses [10, 11]; and heuristics based static analysis techniques have been proposed by researchers for virus detection. The rise in number of variants of an 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 [8]. Detection techniques using a program annotator have been proposed; however, the amount of time taken for analysis by these techniques as reported by the authors for even simple malware is too high and thus not suitable for real time detection [12, 9].

Our work is based on the assumption that an original malware *M* contains a sequence of malicious 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, we assume that the difference between *S* and *S'* would not be very large. We implement an antivirus scanner, SAVE (*Static Analyzer for Vicious Executable*), to prove our assumption. Section 2 describes how SAVE works. It includes the description of PE binary parser and the similarity measure algorithms. We present our experimental results in section 3. In section 4 we present our conclusions and point out future works.

2. System Architecture

Our approach is performed directly on Windows Portable Executable (PE) binary code. It is structured into two major steps, which are illustrated in figure 1.

Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS'04)
0-7695-2291-2/04 $ 20.00 IEEE

Firstly, a suspicious PE file is optionally decompressed if it is compressed by using a third party binary compress tools, for example USP Shell [3], and

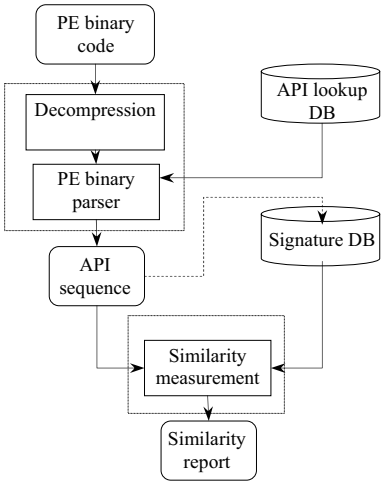

Figure 1: The architecture of our anti-virus system

then passed via a PE binary parser. The output of PE binary parser is a Windows API calling sequence. An API calling sequence consists of a group of 32-bit global ids which represent the static calling sequence of corresponding API functions. We also use the PE parser to generate the API sequence for a particular known Win32 virus. This API sequence will be utilized as the virus' signature and 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 the signature sequence of *v* from the signature database through a similarity measure module which gives 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*. In order to report a non virus, we should repeat the above procedure for all the virus signatures in the signature database.

2.1. PE binary parser

We developed a PE binary parser instead of using a third party disassembler. The reason of doing that is to improve the system performance. Most third party disassemblers output a text file of assembler code, which should be parsed to extract the necessary features for further analysis. The text processes greatly degrade the overall system performance. The second reason is that we are only interested in extracting the Win32 API calling sequence from a PE file, that is to say the disassembling for the instructions other than 'CALL' is not helpful in current version's scanner. Since virus scanner is a speed sensitive application, we decided to use a built-in special purpose PE binary parser to reduce the intermediate text processes and to improve the overall performance.

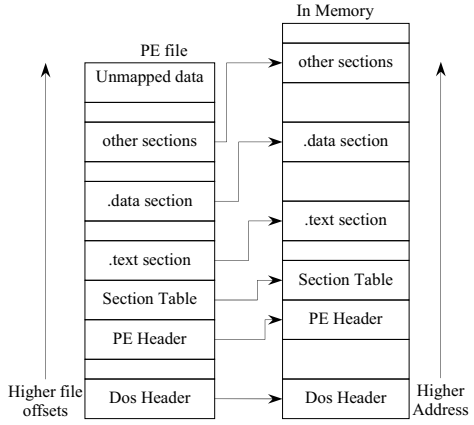

Figure 2: Overview of PE file structure

PE is designed as a common file format for all flavors of windows, on all supported CPUs. The major structure of a PE file is illustrated in figure 2. 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 follows the section table. A PE section represents 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 table, resources and relocations.

Our PE parser is built to extract static API calling sequence. Three major steps should be performed to achieve that goal. The first step is locating 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. The second step is scanning 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. The final step is mapping the API name with its module name to a 32 bits unique global API id by a lookup table.

2.1.1. Locating the imported address table

Within a PE file, there's 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 [4]. 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_DIRECTORY_ENTRY_IMPORT in the DataDirectory array. The imports entry points to an array of IMAGE_IMPORT_DESCRIPTOR structure. There's one such structure for each imported module. Figure 3 shows a PE file importing some APIs from KERNEL.DLL.

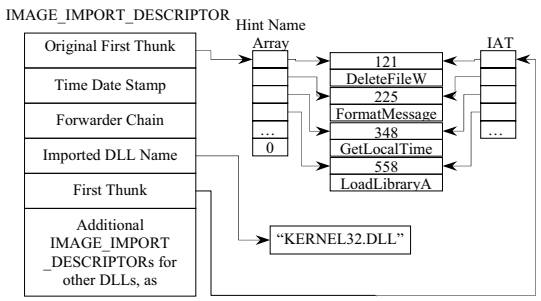

Figure 3: Structure of IMAGE_IMPORT_DESCRIPTOR

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 [5]. 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.

2.1.2. Scanning code sections

By referring to the section table immediately following the PE header we can locate all the code sections in 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 4. The CALL instruction we are interested in is FF1504104A00. The corresponding assembly instruction is 'CALL dword ptr [004A1004]' [6], 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.

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

Figure 4: A sample of machine code from the code section of MyDoom

2.1.3. 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.

2.2. Similarity measures

By using the mechanism described above, we can construct the API sequence of a suspicious PE binary file. Let's denote it *Vu* (vector of unknown). We can also generate a static API calling sequence for a known Win32 virus and store it in a signature database. 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.

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

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 [1] to deal with the 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 5.

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.

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

Figure 5: Sequence alignment represented by path through grid

Figure 6 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 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.

Next, we apply the traditional similarity functions on *Vs'* and *Vu'*. Cosine measure, extended Jaccard measure and Pearson correlation measure are the popular measures of similarity for sequences. The cosine measure is given by

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

and it captures a scale-invariant understanding of similarity. The extended Jaccard measure [2] is computed as

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

which measures the ratio of the number of shared attributes of *Vs'* and *Vu'* to the number of attributes possessed by *Vs'* or *Vu'*. The Pearson's correlation measure is defined as

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

which measures the strength and direction of the linear relationship between *Vs'* and *Vu'*.

The reason to utilize three different measures is that none of them can output the effective results for all sequences measure. The following table 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.

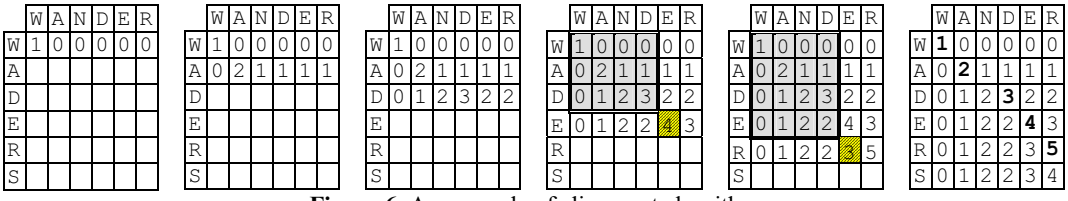

Figure 6: An example of alignment algorithm

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

Our similarity report is generated by calculating the $S^{(m)}(Vs_i', Vu')$ value for each virus 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 virus (or perhaps a new virus that doesn't yet have a signature in our database). In our experiment, using the value of .90 as the threshold works quite well.

Table 1: Mutual correction between measures

$[Vs'],[Vu']$	$S^{(C)}(Vs', Vu')$	$S^{(J)}(Vs', Vu')$	$S^{(P)}(Vs', Vu')$
[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

3. Experimental Results

Several recent viruses (Win32 PE) were used for analysis: Mydoom, Blaster, Beagle, Bika [7]. For each virus we created a set of polymorphic versions by the obfuscation techniques described in [8]. 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 detect these polymorphic versions by using eight different virus scanners and our new scanner. Table 2 shows the experimental results. As can be seen from the last column, our scanner named SAVE which is implemented by the techniques described in section 2, performs 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 SAVE on each benign program; our scanner reported "negative" in each case.

We compare our scanner's performance to that of another static executable scanner, SAFE (Static Analyzer for Executables) [9], which was claimed to be able to detect obfuscated version 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 Mihai and Somesh in their paper. We also ran our scanner against the same executable codes in their experiments.

Figures 7 and 8 show the performance comparison between SAVE and SAFE. The higher curve shows the total time of SAFE for checking four benign Win32 PE files, which are listed at the bottom of the diagram. The corresponding performance of our detector, SAVE, is given in the lower curve. As can be seen, SAVE is about one hundred times faster than SAFE for checking middle size PE file. As shown in Figure 8, with respect to Win32 PE size, the detection time taken by SAFE increases in a much higher rate than our detector.

Table 2: Polymorphic Malware Detection using Different Scanners

	N	M^1	M^2	D	P	K	F	A	S
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, **S** – NMT developed **S**tatic **A**nalyzer for **V**icious **E**xecutable, which uses the methods described in this paper.

4. Conclusions and future directions

In this paper, a methodology for generating signature of Win32 PE malicious is presented, aimed at

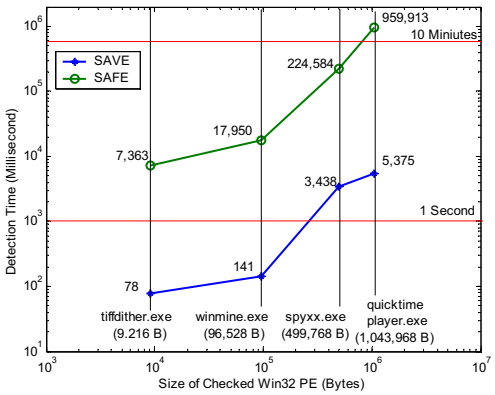

Figure 7: Performance comparison on four executables. (The values near curves are detection time, in millisecond)

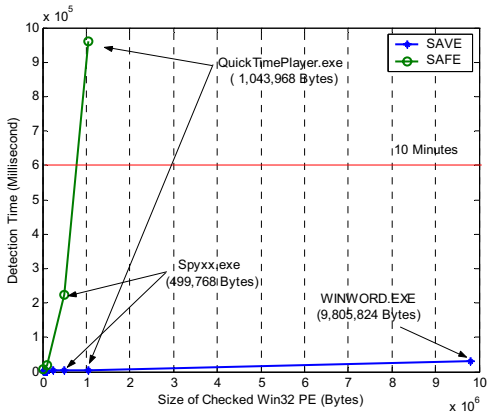

Figure 8: Detection time taken by SAFE increases in a much higher rate than that of SAVE with respect to size

supporting polymorphic malicious code detection. The key idea is that to preserve the functionality, a polymorphic malware should contain a similar API calling sequence. We described in detail the implementation of PE scanner for creating API calling sequence as signature and the algorithm of similarity measure for signature matching. Our experiments showed that our scanner, SAVE, was accurate and efficient in detecting polymorphic malware.

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

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.

References

[1] W.C. Wilson, "Activity Pattern Analysis by Means of Sequence-Alignment Methods", *Environment and Planning*, A (30):1017-1038, 1998.

[2] A. Strehl, and J. Ghosh, "Value-based Customer Grouping from Large Retail Datasets", *Proc. SPIE Conference on Data Mining and Knowledge Discovery*, Orlando, volume 4057, pages 33-42, April 2000.

[3] M. Oberhumer, and L. Molnar, http://upx.sourceforge.net/.

[4] M. Pietrek, "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format", *MSDN Magazine*, March 2002.

[5] Microsoft Corporation, "Portable Executable Formats", *Formats specification for Windows*.

[6] Intel Corporation, "IA-32 Intel® Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference".

[7] Symantec Cooperation, http://securityresponse.symantec.com.

[8] A. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static Analyzer for Vicious Executables (SAVE)", *20th Annual Computer Security Applications Conference*, December 6-10, 2004.

[9] M. Christodorescu, and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns", *Proc. The Twelfth USENIX Security Symposium*, Berkeley, CA, 2003.

[10] F. Cohen, "Computer Viruses: Theory and Experiments", *Computers and Security*, Vol. 6: pp. 22 – 35, 1987.

[11] D. Chess, and S. White, "An Undetectable Computer Virus", *Virus Bulletin Conference*, September 2000.

[12] J. Bergeron, M. Debbabi, M. Erhioui, and B. Ktari, "Static Analysis of Binary Code to Isolate Malicious Behaviors", *Proc. The IEEE 4th International Workshops on Enterprise Security (WETICE'99),* Stanford University, California, USA, June 16-18, 1999.