J Comput Virol (2011) 7:107–119
DOI 10.1007/s11416-010-0141-5

ORIGINAL PAPER

Malware detection using assembly and API call sequences

Madhu K. Shankarapani · Subbu Ramamoorthy · Ram S. Movva · Srinivas Mukkamala

Received: 3 January 2010 / Accepted: 10 March 2010 / Published online: 3 April 2010

Abstract One of the major problems concerning information assurance is malicious code. To evade detection, malware has also been encrypted or obfuscated to produce variants that continue to plague properly defended and patched networks with zero day exploits. With malware and malware authors using obfuscation techniques to generate automated polymorphic and metamorphic versions, anti-virus software must always keep up with their samples and create a signature that can recognize the new variants. Creating a signature for each variant in a timely fashion is a problem that anti-virus companies face all the time. In this paper we present detection algorithms that can help the anti-virus community to ensure a variant of a known malware can still be detected without the need of creating a signature; a similarity analysis (based on specific quantitative measures) is performed to produce a matrix of similarity scores that can be utilized to determine the likelihood that a piece of code under inspection contains a particular malware. Two general malware detection methods presented in this paper are: Static Analyzer for Vicious Executables (SAVE) and Malware Examiner using Disassembled Code (MEDiC). MEDiC uses assembly calls for analysis and SAVE uses API calls (Static API call sequence and Static API call set) for analysis. We show where Assembly can be superior to API calls in that it allows a more detailed comparison of executables. API calls, on the other hand, can be superior to Assembly for its speed and its smaller signature. Our two proposed techniques are implemented in SAVE) and MEDiC. We present experimental results that indicate that both of our proposed techniques can provide a better detection performance against obfuscated malware. We also found a few false positives, such as those programs that use network functions (e.g. PuTTY) and encrypted programs (no API calls or assembly functions are found in the source code) when the thresholds are set 50% similarity measure. However, these false positives can be minimized, for example by changing the threshold value to 70% that determines whether a program falls in the malicious category or not.

M. K. Shankarapani (✉) · S. Mukkamala
Department of Computer Science, Institute for Complex Additive Systems Analysis, Computational Analysis and Network Enterprise Solutions, New Mexico Tech, Socorro, NM 87801, USA
e-mail: madhuk@cs.nmt.edu

S. Mukkamala
e-mail: srinivas@cs.nmt.edu

S. Ramamoorthy · R. S. Movva
Cyber Security Works, New Mexico Tech, Socorro, NM 87801, USA
e-mail: sramamoorthy@cybersecurityworks.com

R. S. Movva
e-mail: ram@cybersecurityworks.com

1 Introduction

The circle of attack and defense in the world of malicious software is one that never ends. Anti-virus companies are competing to devise their best scanning technology, while the malware writers are devising every possible way to defeat the scanners.

Internet worms, Trojans, and backdoors are now a significant growing threat, alongside EXE infectors and macro viruses. Increasingly, the term malware is used to encompass all threats. A malicious code is a piece of code that can affect the secrecy, integrity, and data and control flow, and functionality of a system. Therefore, detection is a major concern within the research community as well as within the user community. As malicious code can affect the data and control flow of a program, static analysis may naturally be

helpful as part of the detection process 0. We discussed the techniques in this work as a comprehensive scanning technology for today's threats and tomorrow's. Together with the dramatic increase of malware per month, the advanced nature of malware attacks has had a marked effect on the nature of scanning technology.

Malicious software is classified broadly based on the payload and propagation mechanism. In this work, we are classifying malware based on their behavioral pattern. By doing this, we can then use our techniques, which are based on similarities to the known malware signature. The main goal for our techniques is to be able to find similarities in payload patterns that can be used to identify variants of a particular malware. Section 2 provides an insight of what has been done in static analyzer area.

Considering that we want to be able to detect future malware, especially malware variants, we also present obfuscation techniques that can be used to generate variants. These techniques can be seen on a lot of malware variants on the field. We also use the same techniques to produce our own brand of variants for our purpose. Such techniques are presented in Sect. 3.

We use two main techniques to detect malware: API call sequence in SAVE and Assembly in MEDiC. These techniques will be discussed in Sects. 4 and 5, respectively, along with the analysis and the results. We also provide our conclusions and future work in Sect. 6.

2 Related Work

Wisconsin Safety Analyzer (WiSA/SAFE) checks for obfuscations by analyzing the control flow of executables [1]. To our knowledge, SAFE was one of the first techniques that focuses on obfuscations and de-obfuscations. Dullien proposed a graph-based approach to compare between two executables [2].

When exposed to a malware, its source code would not be readily available to analyze. SAFE works directly on executables. However, since disassemblers are available, analyzing the source code can be done just as easily. One of our tools, MEDiC, is using the malicious source code for comparison. The other tool, SAVE, is using Windows API.

The striking similarity between SAFE and SAVE, aside from the names, is that they are both analyzing the executables. While SAFE is focusing on the control flow, SAVE is focusing on the API sequence. Comparing SAVE to Dullien's graph based approaches one of the key differences is that SAVE uses a complete API sequence. MEDiC is a completely different beast as it looks into the assembly source code like Dulline's work to determine the similarity between an unknown executable with a known malware in the signature file.

Analyzing the executables directly can mean that these tools (SAFE and SAVE) can only be used in one particular operating system. By focusing on the assembly source code, however, we can bypass the need to reprogram the tool to understand the architecture of the operating system. One vital requirement of creating a source file is a disassembler that will work on the operating system in question.

System call sequences are extensively studied by intrusion detection researchers to detect normal Vs misuse or anomalous behavior [3–6]. One of the primary differences between SAVE, MEDiC and already published work is that the researchers in the IDS community considered very well known attacks vanilla attacks and did not extend them to complex malware. Today's malware has multiple attack vectors and trying to classifying them based on the taxonomies used will be overwhelming. Most of the results reported are based on heuristic approaches (using machine learning or rule based techniques); SAVE and MEDiC are based on distance measures and provide an indicative of malware traces based on similarity measures. The premise of the research is that malware with similar functions share a common signature. MEDiC analyzes and extracts snippets of API sequences that appear frequently in a number of malicious samples. API snippets are parts of an API sequence that appear in an executable. SAVE uses the full API sequence, whereas MEDiC uses just the snippets. To obtain the API snippets of one category, we find shared characteristics of all the malicious codes in that category; these characteristics must not appear in normal programs. Results from a large set of recent spyware, adware, malware, and normal programs are presented below.

3 Obfuscations

The word obfuscation according to Dictionary.com [7] means:

- To make so confused or opaque as to be difficult to perceive or understand.
- To render indistinct or dim; darken.

Taking this approach to code obfuscation, the idea is to make the code difficult for reverse engineering. The concept of obfuscation as explained by Collberg and Thomborson [8] is as follows:

Given a program P and a set of obfuscation transformations T we want to generate program P' such that:

- P' retains functionality and logic of P.
- P' is difficult to reverse engineer.
- Performance of P' is comparable to that of P, i.e. the cost of obfuscation is minimal.

Why does code obfuscation exist if it is that bad? As with everything else in the world, there are always the good side and the bad side. From a legitimate user's point of view, we can expect this scenario: he or she wants to protect his or her work. A software company may not want their adversaries to get a copy of the software. Their adversaries may use the same idea incorporated in this software for their own purpose. It is also wise to make it hard for criminals to understand how the software works as to prevent them from getting valuable information (e.g. passwords, credit card numbers, etc.).

The bad side of obfuscation comes when the techniques are used by malware writers. Malware writers nowadays tend to create many versions of their malware. The code may look a little different, but it runs the same functionality. Perhaps, they differ only in the targets to attack or port numbers to use when propagating. By obfuscating the worms or viruses, they can escape anti-virus detection. Not to mention the malware writers also save the trouble of creating a new malware. At the very least, they can prolong their time to wreak havoc before anti-virus companies provide a detection and removal mechanism. Once they finish milking a particular malware, they can start creating a new malware.

Obfuscation techniques used may be any one or combination of the following [8]:

- *Dead code insertion* There are many dead codes to choose from. The possibilities are virtually endless. With this method, we are able to shift a known signature used by anti-virus software by a few bytes to escape detection.
- *Control flow obfuscation* As the name implies, we shuffle the instructions so that the order in the binary image is different from the order of instructions assumed in the signature used by anti-virus software.
- *Register reassignment* Obfuscation by register reassignment is the process of replacing the usage of one register A with register B if B is not found to be used anywhere in the live range of A. This method has no effect in the program's behavior. Furthermore, it has no time delay of processing extra instructions and jumps.
- *Data obfuscation* Data obfuscation can be as easy as replacing instructions with its equivalence. Other samples of data obfuscation are string splitting or variable type replacement. For example, a Boolean variable may be replaced by two integers [9].
- *Pointer aliasing* This technique is done by replacing variables with global pointers. Functions are referred to by arrays of function pointers. An implementation of this obfuscation is relatively easy using high level languages that allow pointer references. Pointer aliasing can be as simple as changing a = b into *a = **b or as complex as converting all variables and functions into an array of pointers to be referenced by pointers to pointers [10].

Tables 1, 2, 3 and 4 show the first four obfuscation techniques mentioned above, using an excerpt from Beagle's assembly code. All obfuscations are noted in bold.

Table 1 Example of dead code insertion

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh

        or      ecx, 2

        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx

        shr     eax, 8
        push    eax


        invoke
ZipUpdateKeys, a
        pop     eax

        xor     al, a
        ret
ZipEncode endp
```

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        nop
        or      ecx, 2
        inc     eax
        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        push    ecx
        shr     eax, 8
        push    eax
        inc     ecx
        dec     ecx
        invoke
ZipUpdateKeys, a
        pop     eax
        pop     ecx
        xor     al, a
        ret
ZipEncode endp
```

Table 2 Example of control flow obfuscation

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2

        mov     eax, ecx
        xor     ecx, 1

        xor     edx, edx
        mul     ecx

        shr     eax, 8
        push    eax

        invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a

        ret
ZipEncode endp
```

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        jmp     L1
L3:     shr     eax, 8
        push    eax
        jmp     L4
L1:     mov     eax, ecx
        xor     ecx, 1
        jmp     L2
L4:                     invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a
        jmp     L5
L2:     xor     edx, edx
        mul     ecx
        jmp     L3
L5:     ret
ZipEncode endp
```

Table 3 Example of register reassignment

```
; Encode a single byte
ZipEncode proc a: BYTE

        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        shr     eax, 8
        push    eax
        invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a

        ret
ZipEncode endp
```

```
; Encode a single byte
ZipEncode proc a: BYTE
        push    ebx
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        mov     ebx, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        shr     ebx, 8
        push    ebx
        invoke
ZipUpdateKeys, a
        pop     ebx
        xor     al, a
        pop     ebx
        ret
ZipEncode endp
```

Table 4 Example of data obfuscation

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        shr     eax, 8
        push    eax
        invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a
        ret
ZipEncode endp
```

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        mov     eax, ecx
        xor     ecx, 1
        mov     edx, 0
        mul     ecx
        shr     eax, 8
        push    eax
        invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a
        ret
ZipEncode endp
```

In Table 1, we can clearly see that NOP is a no operation instruction, hence a dead code. Pushing the register ECX and popping it later for no reason at all is another example. The same goes for increasing the value of register ECX and decreasing it in a sequence. Meanwhile, increasing the value of register EAX without decreasing it can be dangerous when it is done carelessly. In this example, the value of register EAX is replaced by the value of register ECX in the next line, so changing the value of register EAX would not do anything to the final result.

Table 2 shows chunks of the code in different places connected by the jump instructions and labels. As a result, even though the binary code looks very different from the original, the jump instructions will run the code exactly like the original. Hence, there is no difference whatsoever in the final result.

In Table 3, we simply replace the use of register EAX with the register EBX. In this chunk, the register EBX is originally not used at all. However, for security sake, we push the value of register EBX in the beginning and pop it back at the end, in case the register EBX is actually in use before and after calling this procedure.

In Table 4, an instance of data obfuscation is by replacing the XOR instruction. The operation XOR on the register EDX with itself yields the value of 0 in the register EDX. We can also assign the value of 0 directly to the register EDX. Another method would be to use the operation AND on the register EDX and the value of 0.

Obfuscation methods were used extensively to create our own modified worms and viruses, our own variants. The reasons for creating variants are as follows:

- Most anti-virus software recognizes variants that we collected from the wild.
- We get into the working mind of script-kiddies as we try to fool the anti-virus software into believing that these variants are harmless.
- We can use our variants to devise a new detection method that recognizes variants of a known malware.

4 SAVE

4.1 SAVE overview

Detection using API was the first method started by the New Mexico Tech's malware group. This method is called SAVE (Static Analyzer for Vicious Executables). It was developed in conjunction with the MEDiC method. In SAVE, a signature of a malware is determined by its sequence of API calls. Each sequence is denoted as a vector [10].

SAVE's algorithm of calculating similarity measure is performed directly on Microsoft Windows Portable Executable (PE) binary code. It is structured into two steps which are illustrated in Fig. 1.

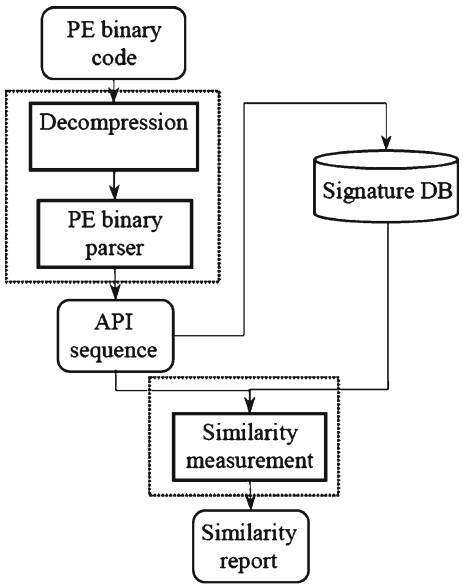

Fig. 1 SAVE's two-step algorithm

First, the PE code is (optionally) decompressed and passed through a PE file parser, producing the intermediate representation which consists of a Windows API calling sequence. We map each Windows API to a global 32-bit integer id number. The 16 most significant bits of the integer represent a particular Win32 module (dynamically linked library), and the last 16 bits represent a particular API in this module. The API calling sequence consists of a sequence of these global id numbers that represent the static calling sequence of the corresponding API functions. This sequence is compared to a known malware sequence or signature (from the signature database) and is passed through the similarity measure module to generate the similarity report. The detection decision is made based on this similarity report.

The PE binary parser transforms the PE binary file into an API calling sequence. It uses two components, W32Dasm version 8.9 and a text parser for disassembled code. W32Dasm by URSoftware Co. is a commercial disassembler, which disassembled the PE code and outputs assembly instructions, imported modules, imported API's, and recourse information. The text parser parses the output from W32Dasm to a static API calling sequence, which becomes our signature.

4.2 Similarity measures

A signature is an API sequence of a known virus that has been previously identified. Let's denote it *Vs* (vector of signature). The API sequence of a suspicious PE binary file is denoted *Vu* (vector of unknown). To identify whether the new

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

Fig. 2 Optimal alignment algorithm

executable with signature *Vu* is an obfuscated version of the virus represented by *Vs*, we measure the similarity between *Vs* and *Vu*.

Modified SAVE performs sequence alignment and uses three different vector calculations as its similarity measures:

- Sequence alignment.
- Similarity functions: Cosine, Jaccard, and Pearson.

4.2.1 Sequence alignment

The optimal alignment algorithm Fig. 2 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 grid [11].

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.

After applying the above algorithm, the two original sequences become "WANDER-" and "WA-DERS". In our case, API sequences *Vs* and *Vu* are inserted with some zeros to generate *Vs*' and *Vu*', which have optimal alignment. The algorithm has a complexity of $O(l_s x l_u)$, where l_s, l_u are the length of sequence *Vs* and *Vu*.

4.2.2 Similarity functions

We apply the traditional similarity functions on *Vs*' and *Vu*'. Cosine measure, extended Jaccard measure, and the Pearson correlation measure are the popular measures of similarity for sequences. The cosine measure is given below and captures a scale-invariant understanding of similarity.

Cosine similarity: Cosine similarity is a measure of similarity between two vectors of n dimensions by finding the angle between them.

$$\text{Cosine similarity} = \cos^{-1}\left[\frac{Vs' \cdot Vu'}{\|Vs'\| * \|Vu'\|}\right] \tag{1}$$

Extended Jaccard measure: The extended Jaccard coefficient measures the degree of overlap between two sets and is computed as the ratio of the number of shared attributes of *Vs*' AND *Vu*' to the number possessed by *Vs*' OR *Vu*'.

$$\text{Extended Jaccard measure} = \frac{Vs' \cdot Vu'}{\|Vs'\|_2^2 + \|Vu'\|_2^2 - Vs'Vu'} \tag{2}$$

Pearson correlation: Correlation gives the linear relationship between two variables. For a series of n measurements of variables *Vs*' and *Vu*', Pearson correlation is given by the formula below.

$$\text{Pearsons correlation} = \frac{\sum_{i=1}^{n}\left(\mathrm{Vs}_i' * \mathrm{Vu}_i'\right) - \left(\mathrm{n} * \overline{\mathrm{Vs}'} * \overline{\mathrm{Vu}'}\right)}{(\mathrm{n}-1) * \mathrm{S}_{\mathrm{Vs}'} * \mathrm{S}_{\mathrm{Vu}'}} \tag{3}$$

where Vs_i' and Vu_i' are values of variable *Vs*' and *Vu*', respectively, at position i, n is the number of measurements, $\mathrm{S}_{\mathrm{Vs}'}$ and $\mathrm{S}_{\mathrm{Vu}'}$ are standard deviations of *Vs*' and *Vu*', respectively, and $\overline{\mathrm{Vs}'}$ and $\overline{\mathrm{Vu}'}$ are means of *Vs*' and *Vu*', respectively.

In the current version of SAVE, we calculate the mean value of the three measures. For a particular measure between a virus signature and a suspicious binary file, $S^{(m)}(Vs_i', Vu')$, which stands for the similarity between virus signature i and a 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 is (a variant of). Let us denote the index as i_{max}. By comparing this largest value with a threshold, we make a decision whether the binary file is a piece of malware and identify which malware it is. In our experiments, the threshold 0.8–0.9 seemed to work quite well.

Table 5 shows the preliminary results of our recent investigation of the MyDoom worm and several other recent worms and viruses, using eight different commercial scanners and proxy services (✓ indicates detection, × indicates failure to detect, and **?** indicates only an alert; all scanners used are the most current and updated version).

The obfuscation techniques used to produce the polymorphic versions of different malware tested in the experiments include control flow modification (e.g. MyDoom V2, Beagle V2), data segment modification (e.g. MyDoom V1, Beagle V1), and insertion of dead code (e.g. Bika V1). Our ongoing experiments also include investigation of metamorphic versions. As can be seen from the last column of Table 5,

Table 5 Polymorphic malware detection using different scanners

	N	M[a]	M[b]	D	P	K	F	A	SAVE
W32.Mydoom.A	✓	✓	✓	✓	✓	✓	✓	✓	✓
W32.Mydoom.A V1	×	✓	✓	×	×	✓	✓	×	✓
W32.Mydoom.A V2	✓	×	×	×	×	×	×	×	✓
W32.Mydoom.A V3	×	×	×	×	×	×	×	×	✓
W32.Mydoom.A V4	×	×	×	×	×	×	×	×	✓
W32.Mydoom.A V5	×	?	×	×	×	×	×	×	✓
W32.Mydoom.A V6	×	×	×	×	×	×	×	×	✓
W32.Mydoom.A V7	×	×	×	×	×	×	×	×	✓
W32.Bika	✓	✓	✓	✓	✓	✓	✓	✓	✓
W32.Bika V1	×	×	×	✓	×	✓	✓	✓	✓
W32.Bika V2	×	×	×	✓	×	✓	✓	✓	✓
W32.Bika V3	×	×	×	✓	×	✓	✓	✓	✓
W32.Beagle.B	✓	✓	✓	✓	✓	✓	✓	✓	✓
W32.Beagle.B V1	✓	✓	✓	×	×	✓	✓	×	✓
W32.Beagle.B V2	✓	×	×	×	×	×	×	×	✓
W32.Blaster. Worm	✓	✓	✓	✓	✓	✓	✓	✓	✓
W32.Blaster. Worm V1	×	✓	✓	✓	✓	✓	✓	×	✓
W32.Blaster. Worm V2	✓	✓	✓	×	×	✓	✓	×	✓
W32.Blaster. Worm V3	✓	✓	✓	✓	✓	×	×	×	✓
W32.Blaster. Worm V4	×	×	×	×	×	✓	✓	×	✓

N Norton, M^a McAfee UNIX, M^b McAfee, *D* Dr. Web, *P* Panda, *K* Kaspersky, *F* F-Secure, *A* Anti Ghostbusters, *SAVE* NMT developed Static Analyzer for Vicious Executables

NMT's SAVE, a signature based detection algorithm, performs the most accurate detection.

Suppose we have a collection of four malware signatures with a signature file of M1, M2, M3, and M4. An unknown executable enters a computer. Using SAVE, we create a sequence of API calls sequence for this executable called U1. This sequence U1 is then compared against M1, M2, M3, and M4. The result of the comparison using similarity measures will be shown on screen as percentages of how similar U1 to M1, to M2, to M3, and to M4.

Table 6 shows the actual similarity values of malware compared to malware and normal programs. The results indicate that normal programs tested have a very less similarity value when compared to the actual malware samples.

By utilizing the above algorithm we implemented our polymorphic virus scanner SAVE (Static Analyzer for Vicious Executable). We created two versions of the scanner SAVE1 and 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.

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. Since SAVE2 is much faster than SAVE1. The performance comparison is shown in Fig. 3. We used SAVE2 as a pre-scanner to perform extremely efficient batch scanning. For the files have high suspicious output, we scan they by SAVE1 again to generate the highly confident output.

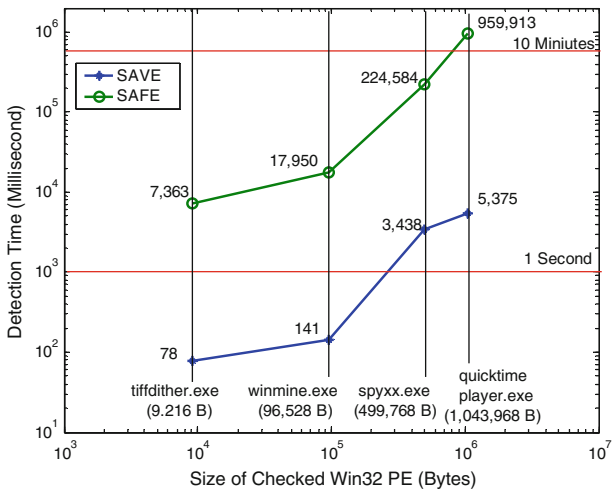

Fig. 3 Performance comparison on four executables. (The *values* near curves are detection time, in millisecond)

Table 6 Similarity values of real malware versus normal programs using API Sequence (SAVE)

	Backdoor. HackDefender. c1.virus	Backdoor. HackDefender. c3.virus	Backdoor. Trojan. c12.virus	Backdoor. Trojan. c16.virus	Construction. Kit. c1.virus	Construction. Kit. c3.virus	Hacktool. Rxhide. c1.virus	Hacktool. Rxhide. c2.virus	Hacktool. c5.virus	Hacktool. c6.virus
Malware										
Backdoor.HackDefender.c1.virus	100	73.74	36.62	31.49	15.6	40.71	6.98	14.23	25.67	23.4
Backdoor.HackDefender.c3.virus	52.02	100	29.46	24.36	33.54	33.54	67.6	54.16	33.54	33.54
Backdoor.Trojan.c12.virus	25.39	32.51	100	38.67	94.59	72.72	57.38	54.12	72.7	69.6
Backdoor.Trojan.c16.virus	40.72	36.61	86.32	100	89.45	89.46	74.08	63.58	88.39	84.22
Construction.Kit.c1.virus	15.2	11.15	12.16	9.11	100	13.18	41.79	35.66	13.18	13.18
Construction.Kit.c3.virus	27.43	30.47	84.14	45.83	93.57	100	66.77	68.55	85.17	75.79
Hacktool.Rxhide.c1.virus	8.09	9.11	9.11	5.05	10.12	10.12	100	23.36	10.12	10.12
Hacktool.Rxhide.c2.virus	17.24	18.24	28.43	20.28	30.48	30.47	39.65	100	30.47	29.46
Hacktool.c5.virus	23.35	28.45	84.13	45.84	93.56	94.6	72.79	65.48	100	83.11
Hacktool.c6.virus	26.41	27.43	83.11	44.82	92.54	93.57	74.85	63.42	84.14	100
Normal										
REGSVR32.EXE	7.09	11.13	8.11	7.08	11.14	7.1	7.1	7.09	9.12	13.18
actmovie.exe	18.25	18.25	19.28	17.23	29.44	23.33	23.33	23.33	21.32	19.28
appletviewer.exe	9.11	0	2.03	7.09	9.12	10.12	10.12	10.12	8.1	0.01
arp.exe	4.05	9.11	9.11	7.08	0.01	0.01	0.01	0.01	4.05	0.01
attrib.exe	0	0	0.01	0	0.01	0.01	0.01	0.01	0	0.01
extcheck.exe	9.11	0	2.03	7.09	9.12	10.12	10.12	10.12	8.1	0.01
iisreset.exe	8.1	9.11	7.08	5.06	0.01	7.09	7.08	7.08	8.1	0.01
ipsecmon.exe	9.11	9.11	14.18	11.14	20.29	15.19	15.19	14.18	12.14	12.16
java.exe	9.11	0	2.03	7.09	9.12	10.12	10.12	10.12	8.1	0.01
ping.exe	6.07	7.09	7.1	10.13	16.21	12.15	12.15	12.15	6.07	11.14
rsh.exe	7.1	13.17	15.19	16.21	16.21	17.23	17.23	14.18	10.12	11.14
rundll32.exe	10.12	10.12	1.02	7.08	10.13	7.08	7.08	7.08	9.11	13.17

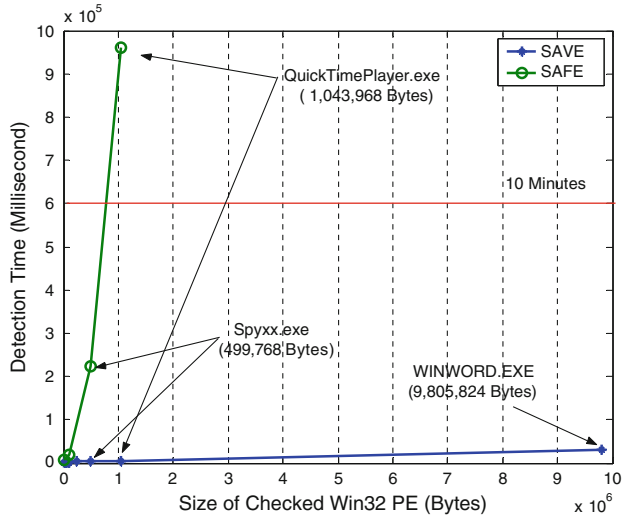

Fig. 4 Detection time taken by SAFE increases in at a much higher rate than that of SAVE with respect to size

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 malware. We ran our scanner against the same executable codes in their experiments for fair comparison. Most of the comparison experiments were performed by Dr. Dennis Xu.

Figures 3 and 4 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 file. As shown in Fig. 4, with respect to Win32 PE size, the detection time taken by SAFE increases in a much higher rate than our detector.

5 MEDiC

5.1 MEDiC Algorithm

MEDiC (Malware Examiner using Disassembled Code) is our answer to a more accurate malware detection method. While MEDiC is still using static detection method, it checks an unknown executable more thoroughly. In MEDiC, a signature of a malware is determined by its disassembled code. While we do not use the whole disassembled code, we still use at least the pseudo-code part of the algorithm.

When an executable is run through a de-assembler, the resulting code includes variable and constant declarations, a list of API calls it makes, and the main algorithm itself in the form of a procedure with a label and a sequence of instructions. In MEDiC, our only concern is the procedures. We treat each procedure as a label and instructions pair that we call *a code island*. A code island is a set of key/value pair with label being the key and the instructions it contains being the value. A label followed by a sequence of instructions is similar to a function or a procedure in a high level language. Each code island in the disassembled code is treated as a checkpoint.

In order to simplify the detection algorithm, we do not include all label/instructions pairs as code islands. In our analysis, we use a *dictionary threshold*. This threshold represents how many instructions there are in a code island in order to be considered important. For example, if we set the threshold to 10, there has to be ten instructions associated with a code island in order for the label to be considered as a code island and be included in the signature set.

For detection algorithm, we use the same method of creating a list of checkpoints from a potential malicious code. First, we disassemble the code use the PE Explorer to get an ASM file. Then, we create a list of checkpoints from the disassembled code. The list of checkpoints contains key/value pairs for the code islands in the code, similar to the ones we get for the signature set Fig. 5.

Since signature matching returns a number of matches and a number of checks, our program sets a *virus threshold*. This is the lowest ratio of matches over the number of checks performed that will conclude the code as malicious.

Based on the checkpoints, our program performs a series of scanning. The first scanning is to match the key/value pairs with the ones in the signature set. If the number of matches exceeds the set virus threshold, then we stop scanning and conclude that this code is malicious. If not we continue to pass 2 in order to perform the second scanning step Fig. 5.

From the misses in the first scanning, we now perform the second scanning. The second scanning excludes the key in the key/value matching and determines how many value matches in case the key names are different. Key names may be different because of padding or moving functions around. Again, we conclude the scan if the number of matches exceeds the set virus threshold Fig. 6.

The third scanning uses a more thorough process. This part is used when even the value does not match. This third scanning is used against a more difficult obfuscation than padding or moving functions. We scan and determine how many instructions in this set match our signature Fig. 6.

Based on the *search threshold* value, we loosen the number of instructions we have to match with the signature as we scan more lines. For example, the search threshold value is set to 5:

- If there are five or less instructions that we have to match in the signature, then we have to match all five instructions in order to include this code island as a match.

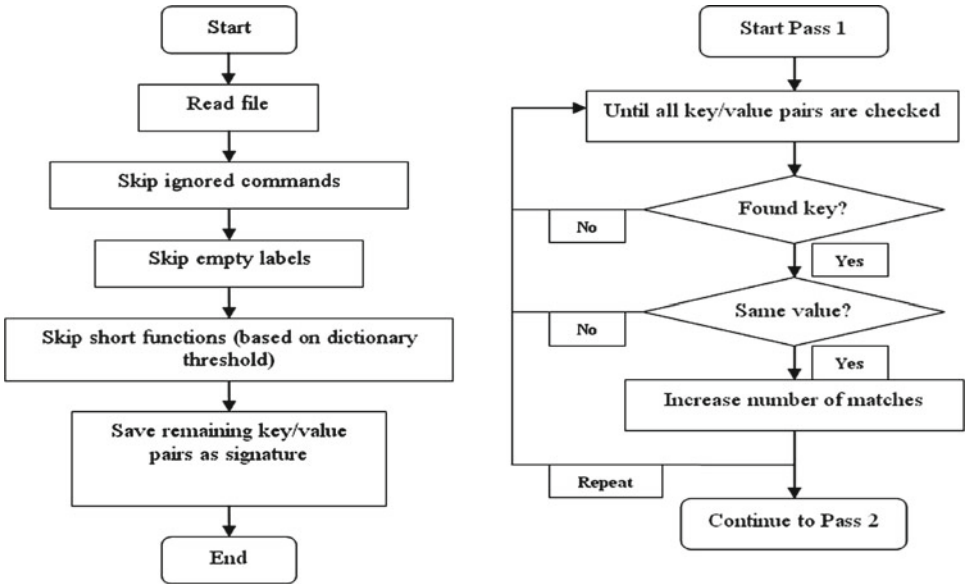

Fig. 5 Flowchart to describe the process of identifying a signature (*left*) and flowchart to describe the first scanning process (*right*)

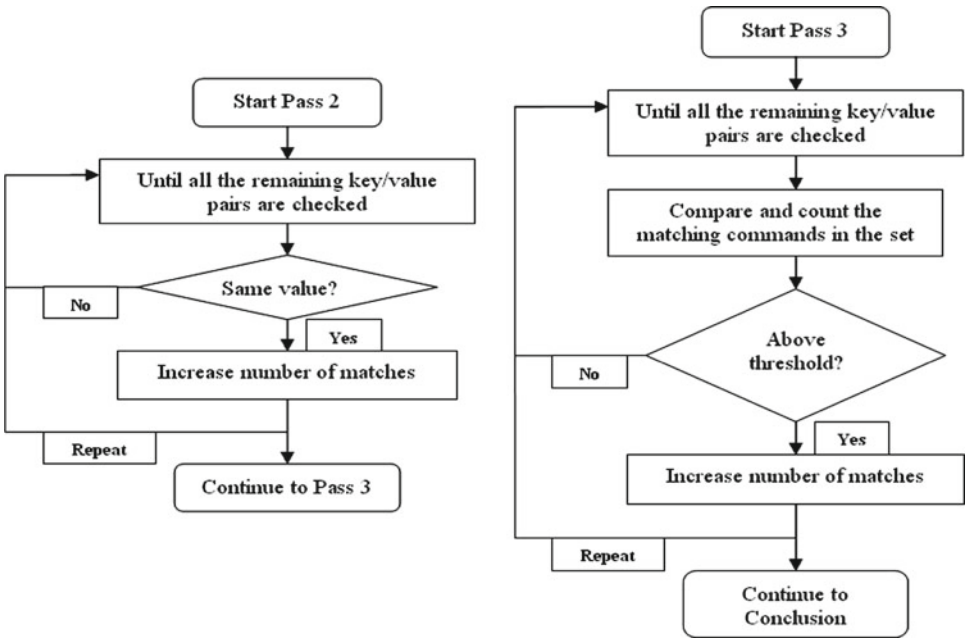

Fig. 6 Flowchart to describe the second (*left*) and third (*right*) scanning steps

- If there are six to ten instructions, we have to match five to nine instructions, respectively.
- If there are 11–15 instructions, we have to match 9–13 instructions, respectively.

5.2 MEDiC experiment on real variants

The next experiment focuses on the variants available in the wild instead of our very own obfuscated versions of the malware. From the malware samples in our collection, we used commercial anti-virus scanners to identify each variant. Let us take Wozer.C as an example. We tested three variants of Wozer: C, E, and F. The anti-virus scanners treat the three samples as different variants. Therefore, the scanners require different signatures to recognize them.

We provided our tool with only one sample for each known malware. We set our tool to recognize a piece of code as malware when *virus threshold* rises above 65%. According to our experiments, this percentage is the value that minimizes the number of false positives down to zero while still maintaining its ability to detect variants of malware in its signature.

Table 7 demonstrates that our tool can detect all three samples of Wozer with only Wozer.C signature in its arsenal. There variants could still be detected because the *virus threshold* value is still reached when compared to its original, Wozer.C. The same trend occurs with other malware like

Table 7 Variant detection using one signature set for each strand of malware

AV classifications	Signatures used					
	Perenast.C1	Wozer.C	Zmist.C1	Beagle.A	MyDoom.A	Klez.E
Perenast.C1	✓	×	×	×	×	×
Perenast.C3	✓	×	×	×	×	×
Wozer.C	×	✓	×	×	×	×
Wozer.E	×	✓	×	×	×	×
Wozer.F	×	✓	×	×	×	×
ZMist.C1	×	×	✓	×	×	×
ZMist.C2	×	×	✓	×	×	×
ZMist.C3	×	×	✓	×	×	×
Beagle.A	×	×	×	✓	×	×
Beagle.I	×	×	×	✓	×	×
Beagle.J	×	×	×	✓	×	×
MyDoom.A	×	×	×	×	✓	×
MyDoom.E	×	×	×	×	✓	×
MyDoom.F	×	×	×	×	✓	×
Klez.E	×	×	×	×	×	✓
Klez.H	×	×	×	×	×	✓

Mydoom, Perenast, Beagle, Klez, and ZMist. While anti-virus software classifies them differently, NMTMEDIC recognizes the samples as variants.

In the real world, application of this technique has a great advantage over current detection techniques deployed by commercial anti-virus software. Suppose a new malware XA surfaces to the public. After a careful analysis, we and anti-virus software provide upgrades. Several weeks later, the malware author lets loose XB, which is a variant of XA. It gets even worse when the malware author releases his or her source code. Many more people can change and let loose their own variants that escape anti-virus detection.

Faced with new unknowns, the easiest method to detect them is to create a signature for each unknown. This seems to be the method used by most virus scanners. Unfortunately, anyone with access to the source code can create as many variants as he or she wants. Meanwhile, with just the original malware, our tool has a greater chance of identifying the variant without an upgrade from us.

Table 8 shows the actual similarity values of malware compared to malware and normal programs using MEDiC. The results indicate that normal programs tested have a very less similarity value when compared to the actual malware samples.

5.3 MEDiC experiments on spyware and adware

Regarding spyware and adware, we include several samples of popular spyware in our experiments to see how our tool performs in this category. The main reason we include spyware and adware into our experiments is that while spyware and adware may not disable the system, their method of installation can be devious. Once installed in a system, they can and may be used by others to perform even more malicious behavior.

From the results in Table 9, we can see that NMTMEDIC can be applied to spyware as well. The numbers represent percentages of similarity of a particular spyware when compared to the signatures provided to our tool.

After analyzing the result for matches around and above 65%, there are several interesting points we can make from this experiment:

- Gator e-Wallet (Gator.CMESys and Gator.GMT) and Claria DashBar (Claria.A.DbAu) have high similarity to each other because they come from the same company. Gator Corporation changed its name to Claria in October 2003. The results are highlighted in bold. Hence the claim here is that in the future any similar product or versions made by this company is bound to be detected by NMTMEDIC.
- Bargains, ShopAtHome, and WebRebates have high similarity matches because they are categorized as data mining programs. All these programs watch the user as and when they are surfing the internet and report the findings to the author. The results are highlighted in bold.
- Save and WhenU.Sync are also from the same company. The result is highlighted in italic.
- Since Gator and Claria function libraries are so large, they are recognized using the signatures of other malicious programs. The results are highlighted in italic.

Table 8 Similarity values of real malware versus normal programs using assembly (MEDiC)

	Backdoor.HackDefender.c1.asm	Backdoor.HackDefender.c3.asm	Backdoor.Trojan.c12.asm	Backdoor.Trojan.c16.asm	Construction.Kit.c1.asm	Construction.Kit.c3.asm	Hacktool.Rxhide.c1.asm	Hacktool.Rxhide.c2.asm	Hacktool.c5.asm	Hacktool.c6.asm
Malwares										
Backdoor.HackDefender.c1.asm	100	54.93	38.03	59.15	18.31	4.23	7.04	33.8	40.85	40.85
Backdoor.HackDefender.c3.asm	62.69	100	40.3	49.25	22.39	7.46	10.45	34.33	43.28	43.28
Backdoor.Trojan.c12.asm	41.07	39.29	100	100	12.5	5.36	5.36	55.36	80.36	80.36
Backdoor.Trojan.c16.asm	43.18	31.82	69.32	100	11.36	3.41	4.55	37.5	53.41	53.41
Construction.Kit.c1.asm	84.62	84.62	53.85	76.92	100	0	15.38	53.85	69.23	69.23
Construction.Kit.c3.asm	30.77	38.46	23.08	23.08	0	100	0	23.08	23.08	23.08
Hacktool.Rxhide.c1.asm	66.67	83.33	50	66.67	33.33	0	100	66.67	50	50
Hacktool.Rxhide.c2.asm	33.33	33.33	52.63	56.14	12.28	5.26	8.77	100	52.63	54.39
Hacktool.c5.asm	51.02	48.98	91.84	95.92	18.37	6.12	6.12	63.27	100	95.92
Hacktool.c6.asm	52.08	50	93.75	97.92	18.75	6.25	6.25	66.67	97.92	100
Normal										
REGSVR32.asm	5.15	5.15	4.91	2.61	2.12	0.95	0.38	0.32	4.73	11.11
actmovie.asm	11.34	11.34	9.2	6.52	2.12	1.68	1.21	1.13	7.69	11.11
appletviewer.asm	14.43	14.43	11.04	6.96	8.49	6.09	2.49	1.88	9.47	11.11
arp.asm	9.28	9.28	9.2	5.65	5.41	6.31	1.36	0.86	7.1	19.44
attrib.asm	6.19	6.19	5.52	2.17	2.12	0.59	1.13	0.59	5.33	2.78
extcheck.asm	14.43	14.43	11.04	6.96	8.49	6.13	2.49	1.88	9.47	11.11
iisreset.asm	7.22	7.22	8.59	5.65	2.12	3.91	1.13	0.7	7.1	8.33
ipsecmon.asm	7.22	7.22	9.82	5.22	3.67	2.04	1.36	1.4	6.51	11.11
java.asm	13.4	13.4	11.04	6.96	8.49	6.04	2.19	1.83	10.06	11.11
ping.asm	7.22	7.22	6.75	4.78	3.86	3.81	0.91	0.59	7.1	19.44
rsh.asm	9.28	9.28	6.13	3.91	1.74	1.86	1.21	0.75	6.51	5.56
rundll32.asm	6.19	6.19	5.52	4.35	4.05	2.5	0.6	0.75	5.92	2.78

Table 9 Analysis of similarity matches for spyware

	Signatures used									
	180SA.Sauhook	Bargains	Begin2Search	Claria.A.DbAu	Ebates	Gator.CMESys	ShopAtHome	Sidefind	WebRebates	WhenU.Sync
Bargains	6.5	100	5.41	4.3	7.9	5.5	**79.2**	8.3	**63.4**	5.9
Claria.A.DbAu	**63.9**	16.8	46.2	100	51.3	**80.3**	18.9	33.3	16.7	41.7
Gator.CMESys	60	12.5	34.2	36.8	40.9	100	13.6	27.7	13.1	34.6
Gator.GMT	**70.8**	30.6	*67.9*	**99.2**	*63.5*	84.9	26.6	*72.2*	29.3	55.2
Save	**66.9**	16.8	48.4	36.4	42.6	*67.3*	21.8	22.2	22.7	*69.4*
ShopAtHome	6.0	**82.5**	5.4	4.8	7.7	5.0	100	8.3	**86.2**	7.0
WebRebates	6.5	**66.9**	5.7	4.9	8.1	4.8	**82.2**	5.5	100	6.9

- 180SA also recognizes a lot of Gator/Claria functionalities. The results are highlighted in bold.

We ran NMTMEDIC equipped with malware and spyware signatures and scanned normal programs. We set the virus threshold to 65%, which means a particular code must be 65% similar to any of the signatures to be considered a variant of a malware or a spyware. In these experiments, we have found no false positives so far, because the chance of malicious code islands to be present in a normal program is very slim. Although, there is one program called PuTTY (a telnet program) that came very close: 60% similar to 180SA and 58% similar to Klez. This is caused by the amount of network functionality in PuTTY.

6 Conclusions and future work

In this paper, a methodology for composing signatures of malicious codes from PE's is presented that is aimed at detecting polymorphic malware and variants of known malware. The key assumption is that to preserve its functionality, a polymorphic malware should contain a sufficiently similar API calling sequence or assembly code. Experiments performed on a large set of polymorphic malware showed that SAVE and MEDiC are accurate and efficient in detecting polymorphic malware and variants of known malware. SAVE and MEDiC can assist to prevent breakouts of previously identified malware while waiting for a more thorough analysis of the variants.

From the two detection techniques, we found that detection using assembly (MEDiC) provides a more detailed and a more accurate detection algorithm with the expense of speed when compared to detection using API call sequence.

The whole technique of identifying the signature for a particular malware is different from the work done by other research groups. From the results of our experiments, we came to a conclusion that we have achieved our main goal of detecting malicious variants that usually escape commercial anti-virus detection in general. The results clearly reveal the alarming deficiency of current scanning techniques and the tremendous potential of our approach.

We found a few programs to be 60% similar to a few malwares, such as those programs that use network functions (e.g. PuTTY) and encrypted programs (no API calls or assembly functions are found in the source code). As for encrypted programs, we have not found legitimate programs that use encryption in their executables. These encrypted programs usually falls in our gray area (we are uncertain of their legitimacy). So we can almost always raise a flag that these programs are highly probable to be malicious.

In the future, we hope to include assembly code equivalency scan to detect more code modifications. For example, the following codes are equivalent (they produce a zero value in register A):

- XOR A, A
- MOV A, 0h
- SUB A, A

In addition to that, we hope to expand MEDiC algorithm to scan binary codes. This will cut its independency on having to disassemble the code first. On the other hand, this approach will also terminate any cross-platform advantage and become like SAVE. The tool must then be written specifically for the machine code of a particular operating system. We also hope to increase the detection accuracy so that the number of false positives will still be minimized.

Acknowledgments The authors would like to acknowledge New Mexico Tech's ICASA (Institute for Complex Additive Systems Analysis). Most of the Initial work was performed under the supervision of Dr. Andrew H. Sung. We would also like to acknowledge Mr. Kartikeeyan Ramamurthy for his insightful suggestions, Dr. Xie Tao for his initial studies of obfuscation that stimulated our interest on the subject, and Dr. Dennis Xu and Dr. Anthonis Sulliman who performed the initial studies on API sequence similarities and performance comparisons between (API sequence and API set) and Wisconsin Safety Analyzer (WiSA/SAFE).

References

1. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. 12th USENIX Security Symposium. http://www.cs.wisc.edu/wisa/papers/security03/cj03.pdf, August 2003
2. Dullien, T., Rolles, R.: Graph-based comparison of executable objects. IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA-2004), pp. 161–173
3. Hofmeyr, S.A., Somayaji, A., Forrest, S.: Intrusion detection using sequences of system calls. J. Comput. Secur. **6**, 151–180 (1998)
4. Wespi, A., Debar, H.: Building an intrusion-detection system to detect suspicious process behavior. Rec. Adv. Intrusion Detect. (1999)
5. Wespi, A., Dacier, M., Debar, H.: Intrusion Detection Using Variable-Length Audit Trail Patterns. Rec. Adv. Intrusion Detect. 110–129 (2000)
6. Kang, D.-K., Fuller, D., Honavar, V.: Learning classifiers for misuse and anomaly detection using a bag of system calls representation. Proceedings of the 6th IEEE Systems, Man, and Cybernetics Workshop (IAW 05), West Point, NY, IEEE, pp. 118–125 (2005)
7. Dictionary, http://dictionary.reference.com/search?q=obfuscation, November 2005
8. Collberg, C., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation—tools for software protection. IEEE Transactions on Software Engineering, pp. 701–746, August 2002
9. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations Technical Report 148, July 1997
10. Sung, A.H., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer for vicious executables (SAVE). Proceedings of 20th Annual Computer Security Applications Conference (ACSAC), pp. 326–334, IEEE Computer Society Press, ISBN 0-7695-2252-1 (2004)
11. Wilson, W.C.: Activity Pattern Analysis by means of Sequence-Alignment Methods. J. Environ. Plan. **30**, 1017–1038 (1998)