J Comput Virol
DOI 10.1007/s11416-007-0074-9

ORIGINAL PAPER

Malware behaviour analysis

Gérard Wagener · Radu State · Alexandre Dulaunoy

Received: 1 July 2007 / Revised: 11 November 2007 / Accepted: 21 November 2007

Abstract Several malware analysis techniques suppose that the disassembled code of a piece of malware is available, which is however not always possible. This paper proposes a flexible and automated approach to extract malware behaviour by observing all the system function calls performed in a virtualized execution environment. Similarities and distances between malware behaviours are computed which allows to classify malware behaviours. The main features of our approach reside in coupling a sequence alignment method to compute similarities and leverage the Hellinger distance to compute associated distances. We also show how the accuracy of the classification process can be improved using a phylogenetic tree. Such a tree shows common functionalities and evolution of malware. This is relevant when dealing with obfuscated malware variants that have often similar behaviour. The phylogenetic trees were assessed using known antivirus results and only a few malware behaviours were wrongly classified.

1 Introduction

This paper proposes an approach to address known anti-reverse engineering techniques in order to determine behaviours of an unknown piece of malware. Our work was motivated by a complex task of analysing a huge amount of malware captured with a medium interaction type of honeypot. All our captured data is known to be malicious and therefore we did not address issues like false positives, or to differentiate between legitimate software and presumed malware. Our main objective is to automatically classify malware based on its behavior. Malware is software that has various malicious goals and often uses anti-reverse engineering techniques to escape from security checks or antivirus programs. In most cases, malware uses anti-reverse engineering techniques in order to make analysis difficult. A huge amount of effort is currently spent to detour anti-reverse engineering techniques [19,21]. The idea behind our approach is not to use traditional reverse engineering techniques like disassembling or debugging but execute a piece of malware in a sandboxed environment and control various parameters, like for instance the execution time, the file-system content, the network, or the windows registry. The execution is done in a virtual operating system that allows to modify execution parameters in an efficient way. During the execution of a piece of malware, the interaction of the latter with the virtual operating system is observed. We will cover these issues in depth in our paper, which is structured as follows: Sect. 2 describes the execution of malware in a secured environment. In Sect. 3 we propose a model for malware behaviour to be a sequence of virtual operating system function calls. These sequences are used to determine similarities and distances between malware behaviours. Section 4 reports experimental results done with malware captured in the wild. Section 5 summarises the related work. Section 6 concludes and describes the future work.

2 Virtual execution of malware

A piece of malware that runs in a Microsoft Windows (*W32*) environment can be examined and its behaviour can be

G. Wagener
LORIA-INRIA, Vandoeuvre, France
e-mail: gerard.wagener@gmail.com

R. State (✉)
INRIA, Le Chesnay Cedex, France
e-mail: state@loria.fr

A. Dulaunoy
CSRRT-LU, Luxembourg, Luxembourg
e-mail: a@foo.be

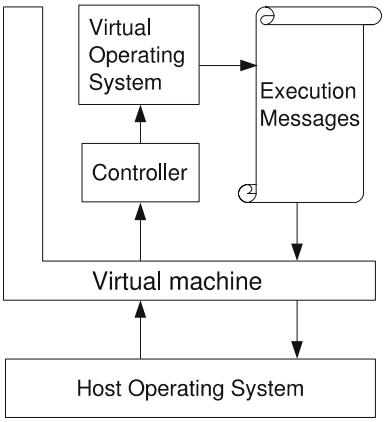

Fig. 1 Virtual execution of malware

extracted by executing it on a plain isolated *W32* machine and comparing the initial state of the machine with the final one. This approach leads however to missing useful intermediate pieces of information. Another secondary requirement is to quickly recover from malware infection. Therefore, a virtual operating system is better suited.

The framework for executing malware is described in Fig. 1. A *U*ser *M*ode *L*inux *UML* [10] is used with restricted network and file system capabilities as a virtual machine. We assume that privileged instructions and direct hardware access are correctly handled by the virtual machine. The virtual machine has a network address and is accessed via *SSH* [28]. Inside the virtual machine *wine* [11] is used as a virtual operating system to execute *W32* malware. A plain debugger is not used because such a tool can be easily detected by a piece of malware. We store the execution messages of *wine* that were generated during execution. The controller uses a heuristic to stop the execution, because a lot of malware does not terminate. In a second stage the execution messages are automatically analyzed and the functions that were called by a piece of malware are extracted. A piece of malware is executed as follows:

1. A new execution environment is created by reusing an execution profile, consisting in a directory and in an emulated network. An execution environment includes file-system with common windows system files, a given windows registry and an emulated network infrastructure. An emulated network infrastructure is a set of commonly used network servers, like for instance DNS[1] servers, web servers or mail servers that can interact with the piece of malware.
2. A piece of malware is copied inside the execution environment via *SSH*.
3. An execution controller is started which includes a heuristics to stop the execution. In fact it is stopped after 10 seconds of execution.
4. This piece of malware is executed and monitored.
5. Raw execution messages are retrieved via *SSH*.
6. The environment is cleaned up.
7. The raw messages are processed in order to find the function calls executed by the piece of malware. The memory layout used during the execution is reconstructed in order to decide which function calls are related to the piece of malware and which ones to the virtual operating system.

[1] Domain Name System – RFC 1035.

3 Analysing malware behaviour

In order to extract a behaviour from a piece of malware and to overcome anti-reverse engineering techniques, a piece of malware is executed in a virtual operating system and some execution parameters are controlled. In a next step, quantitative measures are computed and used to analyse the malware.

3.1 Malware behaviour

Let A be the set of actions that a piece of malware M can perform. An action $a \in \mathcal{A}$ is considered as a virtual operating system function call that is done by M. Each function call $a \in \mathcal{A}$ is mapped to a code $c \in \mathcal{C}$ such that $\mathcal{C} \subset \mathbb{N}$. A piece of malware can have multiple behaviours. One can imagine a piece of malware that runs on Saturdays a different sequence of actions than on Mondays, such that two different behaviours for the same piece of malware are observed. One such behaviour corresponds to a word $a_1 a_2 a_3 \ldots a_n \in \mathcal{A}^*$.

Table 1 shows two sequences of actions executed by a piece of malware M_1 and a piece of malware M_2. The actions done by M_1 can be seen as the sequence of actions $B_{M_1} = \mathit{LoadLibraryA\ GetProcAddress\ GetProcAddress\ GetProcAddress\ WSAStartup\ CopyFileA\ CreateProcessA} \in \mathcal{A}^*$. The sequence of action codes of M_1 is in this example $S_{M_1} = 1\ 2\ 2\ 2\ 10\ 30\ 40 \in \mathcal{C}^*$.

When we observe the first four rows, we see that the two pieces of malware M_1 and M_2 acquire information about the functions of the operating system. In the fifth row however, the piece of malware M_1 intents to do some networking and the piece of malware M_2 reads a value from the registry. The sixth action done by the two pieces of malware is to copy themselves somewhere in the system. The last row shows that both pieces of malware create a new process.

3.2 Determination of malware actions

Execution messages include virtual operating system functions that are started during execution. One important issue that we must address is to differentiate between functions

Table 1 Malware behaviour example

#	M_1		M_2	
	Function call	Code	Function call	Code
1	LoadLibraryA	1	LoadLibraryA	1
2	GetProcAddress	2	GetProcAddress	2
3	GetProcAddress	2	GetProcAddress	2
4	GetProcAddress	2	GetProcAddress	2
5	WSAStartup	10	RegQueryValueA	20
6	CopyFileA	30	CopyFileA	30
7	CreateProcessA	40	CreateProcessA	40

that are called by the operating system and the ones caused by the piece of malware.

Let F be the set of executed functions which includes the functions called by a piece of malware and those called by the virtual operating system itself. $\mathcal{A} \subset F$. A function normally has attached a return address. Let D be the set of memory addresses used during an execution $D \subset \mathbb{N}$. We have a relation $(F, \mathcal{R}, D)$ $\mathcal{R} \subset F \times D$ that characterizes correctly executed functions. The functions that do not participate in $\mathcal{R}$ indicate anomalies, like for instance program abortion. The execution messages also provide information how the functions are loaded into memory during execution. A function is contained somewhere in memory which induces the relation $(F, \mathcal{I}, D)$ $\mathcal{I} \subset F \times D$. We assume that every function, that has a return address and that was not started by another virtual operating system function, is initiated by a piece malware as it is defined in definition 1.

$$\forall (f, m) \in \mathcal{R},\ f \in \mathcal{A} \ \Leftrightarrow\ \forall f' \in \mathcal{F}, (f', m) \notin \mathcal{I} \tag{1}$$

3.3 Malware behaviour similarities

To illustrate the goal of this section, we will take an example illustrated in Table 1. We note that both pieces of malware M_1 and M_2 are doing the same actions, with one notable exception – the fifth action. M_1 is doing some networking and M_2 is manipulating the windows registry. In order to deal with such an issue, two malware behaviours are compared with each other and a similarity function σ is evaluated. This function compares pair-wise all the action codes and attaches scores for matching, respectively for non-matching sequences. Let $S_{M_1} = a_1 a_2 a_3 \ldots a_m \in \mathcal{C}^*$, $m \in \mathbb{N}$ be the behaviour of the piece of malware M_1 and $S_{M_2} = b_1 b_2 b_3 \ldots b_n \in \mathcal{C}^*$, $n \in \mathbb{N}$ be the behaviour of the piece of malware M_2. The two malware behaviours S_{M_1} and S_{M_2} are mapped on a matrix R like it is shown in Fig. 2. The matrix R is conceptually an edit distance matrix [27]. In case that two action codes are equal, a score of one is affected in a first step. In the other case, where the two actions are different the score is set to zero (Eq. 2).

	b_1	b_2	b_3	b_j	b_n
a_1					
a_2					
a_3					
ai					
a_m					

Fig. 2 Matrix-headings

	1	2	2	2	20	30	40
1	1	0	0	0	0	0	0
2	0	2	2	2	1	1	1
2	0	2	3	3	2	2	2
2	0	2	3	4	3	3	3
10	0	1	2	3	4	4	4
30	0	1	2	3	4	5	4
40	0	1	2	3	4	4	6

Fig. 3 S_{M_1} and S_{M_2} scores

In a second step the best previous alignment is added. In case no previous alignment exists, the score 1 is used for matches and 0 for mismatches $R_{1j} = M_{1j}$, $R_{i1} = M_{i1}$. In the other case, Eq. 3 is used. The resulting table is shown in Fig. 3.

$$M_{ij} = \begin{cases} 1 & \text{if } a_i = b_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$R_{ij} = M_{ij} + \max\left(\max_{1 \le k \le i-1} R_{k,j-1}, \max_{1 \le k \le j-1} R_{i-1,k}\right) \tag{3}$$

The similarity function σ (Eq. 4) uses the highest score in the matrix R divided by the mean sequence length.

$$\sigma(S_{M_1}, S_{M_2}) = \frac{2 \cdot \max R_{ij}}{m + n} \tag{4}$$

We note that if the two sequences S_{M_1} and S_{M_2} have no common characters then $\sigma = 0$. If S_1 and S_2 are identical then we obtain $\sigma = 1$. In Fig. 3, $\sigma = 0.85$ and this means that the behaviour of the piece of malware M_1 and that of the piece of malware M_2 are 85% similar.

The formula 5 shows how many function calls of two malware behaviours are different. In the example 3 $\sigma' = \frac{1}{7}$. One of the seven function calls is different.

$$\sigma'(S_{M_1}, S_{M_2}) = 1 - \sigma(S_{M_1}, S_{M_2}) \tag{5}$$

Equation 4 can be used for computing the pairwise similarities of pieces of malware. In a set of malware behaviours P, pairs are created and the average similarity of a given piece of malware is computed with respect to all the other pairs. A malware behaviour with a low average similarity to all the other malware behaviours can be seen as unknown behaviour, that was never seen before. On the other hand, a malware behaviour with a high average similarity can be seen as known malware behaviour.

Let N be the number of malware behaviours $N = card(P)$. Analyzing Eq. 4 we notice that $\sigma(S_{M_i}, S_{M_j}) = \sigma(S_{M_j}, S_{M_i})$. We do not have to compute every possible couple of malware behaviours and the number of needed couples is thus $\frac{N(N-1)}{2}$.

An average similarity $\bar{\sigma}_i$ of a malware behaviour compared with the other ones is defined in Eq. 7. Equation 6 avoids to compute the distance of a malware behaviour with itself.

$$\delta(i, j) = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

$$\bar{\sigma}_i = \frac{\sum_{j=1}^{N} \sigma_j \cdot \delta(i, j)}{N - 1} \tag{7}$$

3.4 Distance between malware behaviours

The similarity approach presented in the previous section has various drawbacks. Notably, the function call order influences the similarity (Eqs. 4 and 5). This is not suitable in cases like the following. Given two pieces of malware M_1 and M_2, where M_1 listens on a backdoor for commands and then observes a user's processes, while M_2 does it the other way round. In such a case the similarity function (Eq. 4) returns a low similarity. An even more realistic case is that a piece of malware does the two activities concurrently. In that case the similarity function (Eq. 4) is influenced by the decisions of the operating system scheduler. The main idea to improve the model is to rely also on the frequencies of function calls. We considered for this purpose the use of the Hellinger distance [1].

As it was previously defined, P is the set of observed malware behaviours from different pieces of malware. Let $\hat{\mathcal{A}}$ be set of all observed function calls done by the analyzed malware (defined in Eq. 8). Let A_c be the set of functions executed by a malware behaviour c.

$$\forall c \in P, \quad \hat{\mathcal{A}} = \bigcup \mathcal{A}_c \tag{8}$$

Using the set of called functions $\hat{\mathcal{A}}$ and a set of malware behaviours, a contingency matrix H' (see Table 2) is built. A row contains the frequencies of function calls for a given malware behaviour. Let x be a malware behaviour, let a be a function call done by a piece of malware. The notation $|x|_a$ is used for designating the number of occurrences of a function a in a malware behaviour x. The function call frequencies contained the matrix H' are $H'_{ij} = |c_i|_{a_j}$. In a next step the relative frequencies of function calls are computed and a new matrix H'', that has the same structure than the matrix H', is created like it is shown in Eq. 9, where $N_{\hat{\mathcal{A}}} = card(\hat{\mathcal{A}})$.

$$H''_{ij} = \frac{H'_{ij}}{\sum_{k=1}^{N_{\hat{\mathcal{A}}}} H_{ik}} \tag{9}$$

Table 2 Contingency matrix of malware behaviours

	a_1	a_2	a_3	...	a_j	...	$a_{N_{\hat{\mathcal{A}}}}$
c_1							
c_2							
c_3							
⋮							
c_i							
⋮							
c_N							

$\forall c_i \in P, \forall a_j \in \hat{\mathcal{A}}$

	GetProcAddress	LoadLibrary	Connect	CreateFile
c_1	$\frac{10(1-\epsilon)}{17}$	$\frac{5(1-\epsilon)}{17}$	$\frac{2(1-\epsilon)}{17}$	$\frac{\epsilon}{1}$
c_2	$\frac{2(1-\epsilon)}{17}$	$\frac{\epsilon}{2}$	$\frac{1(1-\epsilon)}{2}$	$\frac{\epsilon}{2}$
c_3	$\frac{5(1-\epsilon)}{17}$	$\frac{5(1-\epsilon)}{17}$	$\frac{1(1-\epsilon)}{11}$	$\frac{\epsilon}{1}$

c_1 is a malware behaviour.

$c_1 = \underbrace{GetProcAddress \ldots GetProcAddress}_{10times} LoadLibrary \ldots$

Fig. 4 Smooth contingency table

Since it is quite unusual that a piece of malware calls every available virtual operating system function. For every malware behaviour there is a set of functions that were never called (defined in Eq. 10).

$$\forall c_i \in P, \quad V_i = \{a_j \in \hat{\mathcal{A}} \mid |c_i|_{a_j} = 0\} \tag{10}$$

A frequent observation is that a piece of malware did not call some functions. This fact does not mean that this piece of malware never calls these functions. It might be that this behaviour was not observed yet. In order to tackle this problem the technique of statistical smoothing is applied. A matrix H, that has the same structure as the matrix H'' is created. The values are smoothed as given by Eq. 11.

$$H_{ij} = \begin{cases} H''_{ij} & \text{if } V_i = \emptyset \\ \frac{\epsilon}{p} & \text{if } \exists x \in V_i \ p = card(V_i) \\ H''_{ij}(1 - \epsilon) & \text{otherwise} \end{cases} \tag{11}$$

An example of a smoothed contingency table is given in Fig. 4. In that table the malware behaviour c_1 starts the function *GetProcAddress* 10 times. This number is divided by 17 in order to create relative frequencies. Finally that number is smoothed, i.e. multiplied by $(1 - \epsilon)$ because the function *GetProcAddress* was called at least once during execution. For the malware behaviour c_1 the function *CreateFile* was never started and for the malware behaviour c_1 only one function was never started, so p is 1 and ϵ is divided by 1 according the Eq. 11.

Using a smoothed contingency matrix H, the Hellinger distance (defined in Eq. 12), can be applied for the two malware behaviours. The Hellinger distance shows how many information is contained in a malware behaviour and this

measure is invariant to the order of the execution of functions or to the concurrency of the function calls.

$$h : P \times P \rightarrow \mathbb{R}_+ \\ (c_u, c_v) \mapsto \sqrt{\sum_{j=1}^{N_{\hat{A}}} (\sqrt{H_u} j - \sqrt{H_{vj}})^2} \tag{12}$$

3.5 Phylogenetic tree for malware

The analysis of large datasets of malware can be improved by tracking the evolutionary changes in the exposed behavioral profiles.

A phylogenetic tree [13] visualizes the common history of species. It illustrates how species evolved into various families that have specific properties. Usually it is a binary tree where leaves are species. In our case the leaves of the tree are pieces of malware and the parents represent the similarity σ' or distance between these behaviours. A sub tree describes a malware family.

At first, every malware behaviour is added to the phylogenetic tree as a leaf. In the similarity matrix Z (shown in Fig. 10), the largest similarity and respectively the smallest distance are retrieved and the corresponding nodes are looked up. Next the two nodes are grouped. This group is injected in the matrix Z and the entries belonging to the children of the new group are removed. Finally, the new group is added to the phylogenetic tree. This process is continued until the matrix Z is empty. The pseudo code is described in Figs. 5, 6, 7, 8 and 9. The keyword *global* is used in case a variable is shared between procedures. The procedures use hash tables, which are sets of couples (key,values), as distance matrix representation. The notation $x \rightarrow y$ shows that from the hash table x the value of the key y is accessed. The function $keys$ returns an array of the keys from a hash table. The notation $x[i]$ is used to access the ith element of a table x. The notation $x.length$ is used for accessing the number of elements of a table x.

An example of a phylogenetic tree is shown in Figs. 11, 12, and 13. The characters *A*,*B*,*C* and *D* are malware behaviours represented by the pieces of malware that performed that actions. The matrix in Fig. 11 is the initial similarity matrix Z. The smallest value in that matrix is 1 which fulfills the condition $src \neq dst$. The nodes *B* and *C* are grouped and form the sub-tree in Fig. 11. In Fig. 12, the group *BC* is put in the matrix. The cells of the row or column of the group *BC* correspond to the smallest value of the row *B* and *C* with respect to the current cell. The cell (1,0) has the value 3 because $3 < 5$. The merging process is continued (Figs. 12 and 13).

```
1:  procedure FINDGROUP
2:      global nodeList
3:      global min                          ▷ Distance of n1 n2
4:      global n1, n2                       ▷ Selected nodes
5:      min ← ∞
6:      rows ← keys(nodeList)
7:      i ← 0
8:      while i < rows.length do
9:          cols ← keys(nodeList → rows[i])
10:         j ← 0
11:         while j < cols.length do
12:             d ← (nodeList → rows[i] → cols[j])
13:             if rows[i] ≠ cols[j] then
14:                 min ← d
15:                 n1 = rows[i]
16:                 n2 = cols[j]
17:             end if
18:             j ← j + 1
19:         end while
20:         j ← j + 1
21:     end while
22: end procedure
```

Fig. 5 Find two nodes for mergeing

```
1:  procedure GROUPNODES
2:      global id
3:      global nodeList
4:      id ← id + 1                 ▷ New id for the new node
5:      (nodeList → id) ← min       ▷ Add new node
6:      rows ← keys(nodeList)
7:      i ← 0
8:      while i < rows.length do
9:          d_n1 = nodeList → rows[i] → n1
10:         d_n2 = nodeList → rows[i] → n2
11:         if d_n1 > d_n2 then     ▷ Choose smallest one
12:             d_G = d_n2          ▷ Distance for new node
13:         else
14:             d_G = d_n1          ▷ Distance for new node
15:         end if
16:         (nodeList → id → rows[i]) ← d_G
17:         i ← i + 1
18:     end while
19: end procedure
```

Fig. 6 Create group

4 Experimental validation

The Table 3 shows general information about our examined malware set. We have used quite recent malware. The pieces of malware were captured by Nepenthes [18] – Nepenthes emulates known exploitable services and catches thus malware. Furthermore, collected pieces of malware are scanned by the following antiviruses (Fprot, BitDefender, Free-Av and Clamav). About a quarter of the malware (22%) is not detected. The average antivirus detection rate is 51%. Finally, roughly 34% of the malware are worms.

In order to examine similarities between malware behaviours, all pieces of malware are analyzed and the sequences associated to their behaviour are generated. An additional check is done to verify that the virtual operating system

```
1: procedure REMOVEROWS
2:     global nodeList
3:     global n1, n2
4:     newNodeList ← ∅
5:     rows ← keys(nodeList)
6:     i ← 0
7:     while i < rows.length do
8:         if rows[i] ≠ n1 then
9:             if rows[i] ≠ n2 then
10:                x ← nodeList → row[i]
11:                newNodeList → row[i] ← x
12:            end if
13:        end if
14:        i ← i + 1
15:    end while
16:    localList = newLocalList
17: end procedure
```

Fig. 7 Remove rows

```
1: procedure ADJUSTCOLUMNS
2:     global nodeList
3:     global n1, n2
4:     newNodeList = ∅
5:     rows = keys(nodeList)
6:     i ← 0
7:     while i < rows.length do
8:         cols = keys(nodeList → rows[i])
9:         j ← 0
10:        while j < rows.length do
11:            if cols[j] ≠ n1 then
12:                if cols[j] ≠ n2 then
13:                    y ← newNodeList
14:                    x ← nodeList → rows[i] → cols[j]
15:                    y → rows[i] → cols[j] ← x
16:                end if
17:            end if
18:            j ← j + 1
19:        end while
20:        i ← i + 1
21:    end while
22:    nodeList = newNodeList
23: end procedure
```

Fig. 8 Adjust columns

```
1: while nodeList ≠ ∅ do
2:     findGroup
3:     groupNodes
4:     removeRows
5:     adjustColumns
6:     addToTree(n1, n2, min)
7: end while
```

Fig. 9 Build the tree

is resistant to common anti-reverse engineering techniques Unfortunately, static analysis was not possible on 18% of the malware. We checked this by observing the exit code of a disassembler *objdump* [20]. Furthermore, according to the

	S_{M_1}	S_{M_2}	S_{M_3}	S_{M_j}	S_{M_N}
S_{M_1}	0				
S_{M_2}		0			
S_{M_3}			0		
S_{M_j}				0	
S_{M_N}					0

Fig. 10 Similarity matrix Z

	A	B	C	D
A	0	3	5	2
B	3	0	1	4
C	5	1	0	8
D	2	4	8	0

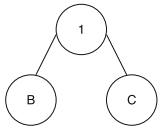

Fig. 11 Groups B and C

	A	BC	D
A	0	3	2
BC	3	0	4
D	2	4	3

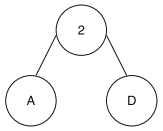

Fig. 12 Groups A and D

	AD	BC
AD	0	3
BC	3	0

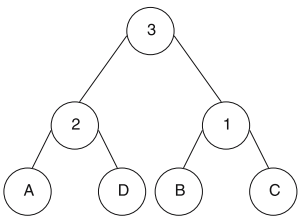

Fig. 13 Groups AD and BC

Table 3 The malware data set

Number of malware	104
Observation period	2005–2007
Malware from 2005	10
Malware from 2006	91
Malware from 2007	3
Average file size	135 KB
Smallest file	8 KB
Biggest file	665 KB
Worms	34%
Not detected by antivirus	22%

Norman sandbox [19], 15% of the existing malware use anti-emulation code. Anti-emulation code can be used to detect or confuse debuggers and monitoring tools. Our results are represented in Table 4. We have created some binaries that contain anti-reverse engineering code and we have included a behaviour that should be observed. Next we have used analysis tools and tried to find the defined behaviour. At first we detected a debugger using the processor flags and the code

Table 4 Anti-reverse engineering techniques used with various reverse engineering tools

Technique	Debugger	Disassembler	Monitor	Virtual OS
anti-debugger	×	√	√	√
OP code generation	√	×	√	√
obfuscated assembler code	√	×	√	√
integrity check	×	√	√	√
sleep	√	√	×	×
exceptions	×	×	√	√
anti monitor	√	√	×	√
anti virtual OS	√	√	√	×

changes its behaviour in case a debugger is running. Next we generated machine instructions during execution and executed them. Next we created obfuscated assembler code which cannot be examined by *objdump*. We have also tested the code integrity check. The sleep action is commonly used to escape from sandboxes due to the fact that an execution cannot run for an infinity of time. As an exception handling technique we used a division by zero. We used the *CreateFile* function to communicate directly with the monitoring tools and thus detected them. Furthermore we observed special environment parameters and identified that we are running in a virtual operating system. Finally we injected *Linux* system calls in a windows binary in order to escape from the virtual operating system and we noticed that the damage is restricted inside the *UML*. Although, the success rates of a monitoring tool and a virtual operating system are the same, a virtual operating system provides more execution information than monitoring tools.

Using the above mentioned sequences compute pairwise distances among all pieces of malware. A top 10 list of the most common malware behaviours was generated. Similarly, a list of the top 10 most exotic/rare malware behaviours was obtained. In order to get a finer classification we built a phylogenetic tree of malware behaviours.

For the sake of clarity, Table 5 lists an incomplete set of malware behaviours that have a high similarity σ – in other words these are completely similar. In the first row we see that this malware was transformed to escape from signatures and a new signature was the antivirus reply. We can also observe that the piece of malware *Sdbot1234944.1* and the the piece of malware *Backdoor-Server/agent.aew* start the same sequence of functions during execution.

Table 6 gives an overview about the average similarity $\bar{\sigma}$. The left part of the table shows the malware behaviours

Table 5 Most similar observed malware

WORM/Rbot.193536.29	WORM/Rbot.177664.5
Worm/Sdbot.1234944.1	Backdoor-Server/Agent.aew
Worm/Sdbot.1234944.1	Unknown
Worm/IRCBot.AZ.393	Worm/Rbot.140288.8
Backdoor-Server/Agent.N.1	Worm/Win32.Doomber
Trojan.Gobot-4	Trojan.Gobot.R
Trojan/Dldr.Agent.CY.3	W32/Virus.A virus
Trojan.Gobot-4	Trojan.Downloader.Delf-35
Trojan.Mybot-5011	Trojan.IRCBot-121
Trojan.Mybot-5079	Trojan.EggDrop-5

Table 6 Similarity classification

Lowest average similarity		Highest average similarity	
Malware name	$\bar{\sigma}$	Malware name	$\bar{\sigma}$
Win32.Virtob.E	0.010	Worm/IRCBot.AZ.393	0.440
Win32.Virtob.C	0.021	Worm/Rbot.140288.8	0.440
Backdoor.EggDrop.V	0.039	Worm/Rbot.94208.37	0.439
Unknown malware	0.064	W32/Ircbot1.gen	0.439
Unknown malware	0.070	W32/Ircbot1.gen	0.438
RBot.D3186764	0.075	W32/Spybot.NOZ	0.437
SDBot.AMA	0.105	Generic.Sdbot.68B7CEC5	0.437
Backdoor.Oscarbot.A	0.126	RBot.668E20D5	0.436
Unknown virus	0.128	RBot.DD0FC8A7	0.436
RBot.227328	0.131	RBot.C64D5E67	0.436

with a low average similarity. The associated behaviours are not similar to other behaviours of the analyzed malware set. Some of those pieces of malware did only a few function calls. The right side indicates the malware behaviours that were often observed. These behaviours have a high similarity σ. From antivirus software, we observed that 34% of the malware are worms which explains this high similarity. A worm often exploits a service and needs to inject shellcode using the functions *LoadLibrary* and *GetProcAddress*.

The set of malware was analyzed and a similarity matrix captures the relationships among the pieces of malware. We have constructed a phylogenetic tree using this matrix. We can not include in this paper the complete trees due to space constraints, but the complete trees are on-line.[2] We will illustrate though in the following some sub-trees.

In Fig. 14, we can highlight three malware families. One family can be called the malware kernel family and the other one the malware *W32* API[3] family. Malware of the first family uses direct kernel functions in order to escape from API

[2] http://nepenthes.csrrt.org:10080/malware_behaviour.

[3] Application Programming Interface.

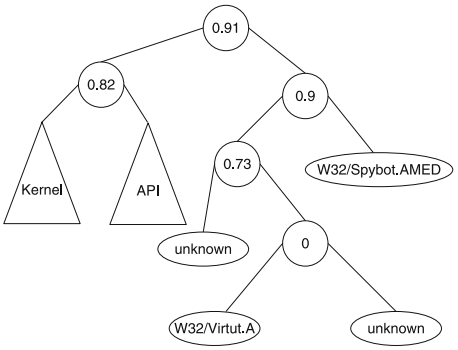

Fig. 14 Phylogenetic tree root

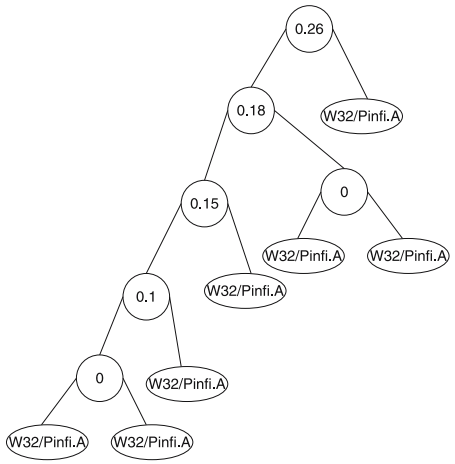

Fig. 15 Phylogenetic tree – pinfi family

hooking, a technique for monitoring function calls, while the members of the *W32* API family use *W32* API functions. The third family clusters malware for which an initial condition was not fulfilled during execution. The three families are not very similar, σ' is close to 1, which can be explained since each family uses a different set of system functions. Malware labelled *unknown* was not detected by antiviruses.

In Fig. 15 we show a malware family called *pinfi* by the Norman sandbox. This family is a sub-tree of the API family. This piece of malware has various behaviours but is classified in a same family due to the fact that that malware starts similar functions. The *pinfi* malware creates various files on the hard disk, creates different registry keys, monitors which functions are started by the user, does some IRC,[4] observes the user's clipboard and checks some processor flags. The latter operations are done in order to know whether it runs in an debugger or on a real machine. We observe different similarities σ' due to the fact that the pieces of malware behaves differently in the virtual operating system.

We have built two trees: for one tree we used the the Hellinger distance matrix, while the other one used a similarity matrix. The names of the *Norman sandbox* were chosen. These names are composed of three parts. The first part indicates the platform on which the piece of malware is running (in our case *W32*). The next part is a specific name for the piece of malware. The last part identifies a variant of the piece of malware. In the two trees we counted the leaves where the first and the second part did not match. We ignored the nodes where one leaf was not detected by an antivirus. Results are presented in Table 7. Seven percent of the tested pieces of malware mismatched with *Norman*'s names. These pieces of malware were analyzed manually and we noticed that some of them executed a similar function sequence but with different parameters. We ignored these parameters in our model (presented in Sect. 3) due to the fact that multiple parameters are execution specific. Another interesting observation is that some of these pieces of malware behave similarly at the beginning. This could be explained by a malware author that has copied parts from existing malware into a newer offspring.

Table 7 Information about phylogenetic trees

Total leaves	104
Unknown leaves	38
Number of incoherence	5

5 Related work

Most antivirus software frequently use signature matching techniques for detecting malware, but as it was revealed in [4,22], this approach can be easily detoured. The authors of [4] propose a mean to undo obfuscated malware code in order to improve the detection rate of antivirus scanners. Another idea is presented in [2], where a method for detecting polymorphic malware based on sub graph isomorphism problem, is introduced. Model checking based solution were also tried out and a first report is given in [3]. The relevant previous works on API/system call level based analysis of malware are described in [8,14,17,22–24]. Some previous papers addressed the accessing of the function calls from a malware, using static binary analysis [15,22], while other authors considered the emulation of malware [17,19,25]. The CWSandbox [26] uses the technique of API hooking for monitoring the activities of a piece of malware. The authors of [14] use a hybrid technique, since static binary analysis can be easily fooled [16] and on the other hand malware can be emulated [17,25]. The authors of [7,5] argue that the emulation technique has also its drawbacks, because no guarantees are given on whether the initial conditions are fulfilled and if the complete binary gets executed. A groundbreaking result [6] shows that it is not possible to build a perfect controller that always knows if a piece of malware terminate or

[4] Internet Relay Chat RFC 2810.

not. This is also the reason why we had to use heuristics to control the malware execution. Two major papers [9,12] have addressed the construction of phylogenies for malware. The first of them [9] addressed the computational complexity of such a task and proposed greedy algorithms for this purpose. The paper [12] addressed the tracking of malware evolution based on opcode/instruction level permutations and reordering. Our work is complementary to these previous works, since we leverage tree constructions approaches for data that is directly related to the behavior of a malware.

6 Conclusion and future works

In this paper we addressed the automated classification of malware based on behavioral analysis. We define a malware behavior to be a sequence of executed functions. We introduced a mean to compute similarities and distances between malware behaviors based on sequence alignment respectively the Hellinger distance. We propose to detect unknown (0 day) malware based on the low average similarity with existing known one. This paper also introduces a phylogenetic tree based approach for malware behavior which can track the evolution of malware features an implementations. We validated our approach with real data coming from a large malware capturing infrastructure. We plan to extend our future research work with better heuristics, an improved sequence alignment procedure and a differentiated (per called function) analysis.

Acknowledgements We thank Mr Eric Filiol research professor at ESAT, for the pointers and suggestions concerning the related works in the area.

References

1. Abdi, H.: Distance. In Salkind, N.J. (eds.) Encyclopedia of Measurement and Statistics, pp. 280–284. Sage, Thousand Oaks (2007)
2. Bruschi, D., Martignoni, L., Monga, M.: Recognizing self-mutating malware by code normalization and control-flow graph analysis. IEEE Secur. Privac. (2007, in press)
3. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 5–14. ACM Press, New York (2007).
4. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005
5. Ferrie, P.: Attacks on virtual machine emulators. In Proceedings AVAR (2006)
6. Filiol, E.: Les virus informatiques: théorie, pratique et applications. Springer, Heidelberg (2004)
7. Ford, R.: The future of virus detection. Information Security Technical Report, pp. 19–26. Elsevier, Amsterdam (2004)
8. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for Unix processes. In: Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy, pp. 120–128. IEEE Computer Society Press (1996)
9. Goldberg, L.A., Goldberg, P.W., Phillips, C.A., Sorkin, G.B.: Constructing computer virus phylogenies. J. Algorith. **26**(1), 188–208 (1998)
10. Hoskins, M.E.: User-mode linux. Linux J. **2006**(145), 2 (2006)
11. Julliard, A.: Wine. http://www.winehq.com
12. Karim, Md.E., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. J. Comput. Virol. **1**(1–2), 13–23 (2005)
13. Kim, J., Warnow, T.: Tutorial on phylogenetic tree estimation (1999)
14. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based spyware detection. In: Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, August 2006. Springer, Heidelberg
15. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: SSYM'05: Proceedings of the 14th Conference on USENIX Security Symposium, Berkeley, CA, USA, p. 11. USENIX Association (2005)
16. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Secur. Privac. **5**(2), 40–45 (2007)
17. Swimmer, A.M.M., Le Charlier, B.: Dynamic detection and classification of computer viruses using general behavior patterns. In: Proceedings of the 5th International Virus Bulletin Conference, pp. 75–88 (1995)
18. Nepenthes: http://nepenthes.mwcollect.org
19. Norman: http://sandbox.norman.no
20. Objdump: http://www.gnu.org/software/binutils
21. Peid: http://peid.tk/
22. Sung, A.H., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04), Washington, DC, USA, pp. 326–334. IEEE Computer Society (2004)
23. Mazeroff, G., De Cerqueira, V., Gregor, J., Thomason, M.G.: Probabilistic trees and automata for application behavior modeling. In: Proceedings of the 43rd ACM Southeast Conference (2003)
24. Mody Tony Lee, J.J.: Behavioral classification. In: Proceedings Eicar'06, May 2006
25. Matthew Evan Wagner. Behavior oriented detection of malicious code at run-time. Master's thesis, Florida Institute of Technology (2004)
26. Willems, Carsten Holz, Thorsten, Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. Secur. Privac. Mag. **5**(April), 32–39 (2007)
27. Wilson: Activity pattern analysis by means of sequence-alignment methods. Environ. Plann. **30**, 1017–1038 (1998)
28. Ylonen, T.: SSH – secure login connections over the internet. In: Proceedings of the 6th Security Symposium, p. 37. USENIX Association, Berkeley (1996)