Malware Behaviour Analysis

Gérard Wagener[1], Radu State[2], Alexandre Dulaunoy[3]
LORIA-INRIA[1] INRIA[2] CSRRT[3]

March 15, 2007

Abstract

A malware is a malicious software that often uses anti-reverse engineering techniques to escape from security officers or anti-virus programs. A lot of malware analysis techniques suppose that the assembler code of a malware is available, which is often not the case. We propose in this a paper, an easy and automated mean to extract malware behaviour by observing the system function calls. We apply a sequence alignment method to extract similarities of malware behaviours. Using these similarities we are able to classify malware and identify both malware with known and unknown behaviour. Experimental results show that obfuscated malware variants often have a similar behaviour. We are able to build a phylogenetic tree of malwares in an automated way, where common functionalities identify a potential shared history.

1 Introduction

A malware is a piece of software that has various malicious goals. A lot of malware use anti-reverse engineering techniques in order to make its analysis difficult. A huge amount of effort is currently done to detour anti-reverse engineering techniques [12] [14]. In this paper we propose an easy approach to detour known anti-reverse engineering techniques in order to determine behaviours of a malware. The idea is not to use traditional reverse engineering techniques like disassembling or debugging but execute a malware in a sand-boxed environment and control various parameters, like the execution time, file-system content, network, or the windows registry. The execution is done in a virtual operating system that easily allows to modify the execution parameters. During the execution of malware we observe the interaction of the malware with the virtual operating system.

In section 2 is described the malware execution. In section 3 we define malware behaviour as a sequence of virtual operating system function calls done by a malware. We use these sequences to determine similarities between malware behaviours. In section 4 we analysed 104 malwares and we noticed that our approach can be used to determine common and unknown malware behaviours. Section 6 contains a conclusion and describes the future work.

2 Virtual execution of malware

Our goal is to examine malware that run in a Microsoft Windows (W32) environment and extract its behaviour. A straightforward strategy is to execute the malware on a plain isolated W32 machine and compare the initial state of the machine with the final one. Using this approach a lot of useful intermediate information is lost. We also need a mean to quickly recover from malware infection. Therefore a virtual operating system is better suited.

The malware execution is described in figure 1. We use a *U*ser *M*ode *L*inux [20] with restricted network and file system capabilities as virtual machine. We assume that privileged instructions and direct hardware access are correctly handled by the *UML*. The *UML* has a network address and is accessed via *SSH* [16]. Inside that *UML* we use *wine* [6] as a virtual operating system to execute the windows malware.

We do not use a classical debugger to execute the malware because a debugger can be easily detected by a malware. We observe wine's execution messages which are printed during a malware's execution in an automated way. The controller uses heuristics to stop the execution, because a lot of malware do not terminate. In a second stage we automatically analyse the execution messages deeply and extract the functions that had been started by the malware. A malware is executed as follows.

1. A new execution environment is created by simply copying a directory and establishing an emulated network. An execution environment includes file-system with common windows system files, with a given windows registry and with a emulated network infrastructure. An emulated network infrastructure is a set of common used servers in the Internet, like DNS[1] servers, web servers or mail servers that can interact with the malware.

2. A malware is copied inside the execution environment via *SSH*.

3. An execution controller is started which includes a heuristic to stop the execution. The malware execution is stopped after 10 seconds.

4. A malware is executed and monitored.

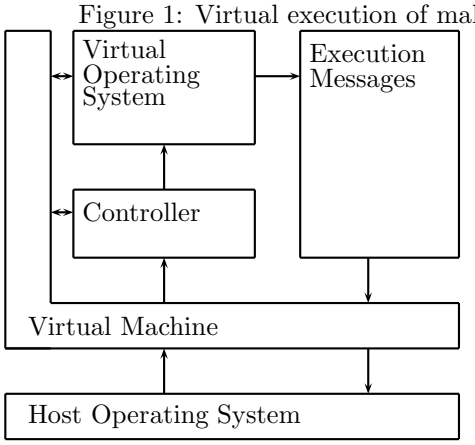
5. Raw execution messages are retrieved via *SSH*.

6. The environment is cleaned up.

7. The raw messages are processed in order to find the function calls which are done by a malware. The memory layout used during the execution is reconstructed in order to decide which function calls are related to the malware and which ones to the virtual operating system.

[1]Domain Name System - RFC 1035

Figure 1: Virtual execution of malware



3 Analysing malware behaviour

In order to extract a behaviour from a malware and to overcome anti-reverse engineering techniques, we execute a malware in a virtual operating system and control some execution parameters. In a next step we propose a mean to observe similarities between various malware behaviours.

3.1 Malware behaviour

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

Table 1 shows two sequences of actions executed by a malware M_1 and a malware M_2. The actions done by M_1 can be seen as the sequence of actions B_{M_1} = *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* $\in \mathcal{C}^*$.

Table 1: Malware behaviour example

M_1		M_2	
Function Call	Code	Function Call	Code
LoadLibraryA	1	LoadLibraryA	1
GetProcAddress	2	GetProcAddress	2
GetProcAddress	2	GetProcAddress	2
GetProcAddress	2	GetProcAddress	2
WSAStartup	10	RegQueryValueA	20
CopyFileA	30	CopyFileA	30
CreateProcessA	40	CreateProcessA	40

When we observe the first four rows we see that the two malwares M_1 and M_2 acquire information about operating system functions. In row five we see that the malware M_1 intents to do some networking and the malware M_2 reads a value from the registry. The sixth action done by the two malwares is to copy themselves somewhere in the system. Finally the seventh row indicates us that both malwares create a new process.

3.2 Determination of malware actions

Execution messages include virtual operating system functions that are started during execution. We have to decide which functions are called by the operating system and which functions are called by the malware.

Let F be the set of executed functions which includes the functions done by a malware and those done 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 execution $D \subset \mathbb{N}$. We have a relation $(F, \mathcal{R}, D)$ $\mathcal{R} \subset F \times D$ that characterises correctly executed functions. The functions that do not participate in $\mathcal{R}$ indicate anomalies, like program abortion. In the execution messages we also observe that libraries are loaded during execution. Let L be the set of loaded libraries during execution. A library is contained somewhere in memory and induces the relation $(L, \mathcal{I}, D)$ $\mathcal{I} \subset L \times D$. We assume that every function, that has a return address that does not belong to a library, is initiated by a malware and not by the virtual operating system as it is defined in property 1. This property ignores the fact that a malware can patch a library's code located in memory which is sometimes done by malwares.

$$\text{Let } m \in D, \text{let } f \in F \ (f, m) \notin \mathcal{I} \Leftrightarrow f \in \mathcal{A} \qquad (1)$$

3.3 Malware behaviour similarities

By looking at table 1 we note that both malwares M_1 and M_2 are doing the same actions, except the fifth action. M_1 is doing some networking and M_2 is manipulating the registry. In this section we propose a mean to determine such similarities.

At first we compare two behaviours with each other and we determine a similarity function σ. For finding this function we compare pair-wise every action code and attach scores for matching, respectively for non-matching. Let $S_{M_1} = a_1 a_2 a_3 \dots a_m \in \mathcal{C}^*, m \in \mathbb{N}^*$ the behaviour of a malware and $S_{M_2} = b_1 b_2 b_3 \dots b_n \in \mathcal{C}^*$, $n \in \mathbb{N}^*$ the behaviour of a malware M_2. The two malware behaviours S_{M_1} and S_{M_2} can be mapped on a matrix R like it is shown in figure 2. The matrix R is conceptually an edit distance matrix [22]. In case two action codes are equal we affect in a first step a score of one. In the other case where the two actions are different the score is set to zero like it is shown in equation 2. In a second step we add the best previous alignment. In case no previous alignment exists we simply use as score 1 for matches and 0 for mismatches $R_{1j} = M_{1j}$, $R_{i1} = M_{i1}$. In the other case we use the equation 3. The resulting table is shown in figure 3.

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

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

We can now define a similarity function σ as the highest score in the matrix divided by the average of the sequence length of $S_{M_1}$and S_{M_2} as it is shown in equation 4.

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

Figure 2: Matrix-headings

	b_1	b_2	b_3	b_j	b_n
a_1					
a_2					
a_3					
ai					
a_m					

Figure 3: S_{M_1} and S_{M_2} scores

	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

In case where two sequences S_{M_1} and S_{M_2} have no common characters $\sigma = 0$ and in case where S_1 and S_2 are identical we obtain $\sigma = 1$. In figure 3, $\sigma = 0{,}85$ that means that the behaviour of malware M_1 and the behaviour of malware M_2 are 85% similar.

A malware behaviour similarity can be seen in a different way like it is shown in equation 5. Intuitively it shows how many function calls are different. In the example 3 $\sigma' = \frac{1}{7}$. One of the seven function calls is different.

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

We want to see similarities between malware behaviours. Equation 4 can be used for one malware behaviour pair. In a set of malware behaviours P we create pairs and we compute the average similarity of a given malware regarding all the other pairs. A malware with a low average similarity regarding all the other malware behaviours can be seen as unknown behaviour of a malware, 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 $N = card(P)$. Looking at equation 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 behaviour and the number of needed couples is thus $\frac{N(N-1)}{2}$. An average similarity $\bar{\sigma_i}$ of a malware compared with other malware behaviours is defined in equation 7.

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

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

3.4 Malware behaviour phylogenetic tree

In section 3.3 we have introduced malware behaviours which can be used for classification of malware. We define a similarity matrix Z that contains the similarities between the various malware behaviours like it is shown in figure 5, which is a matrix where a cell corresponds to a similarity σ' between two malwares. Using such a similarity matrix we can construct a phylogenetic tree [7] of malware behaviours in which we can observe common malware behaviour groups.

A phylogenetic tree shows the common history of species. It clarifies 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 malwares and the parents represent the similarity σ' between these behaviours. A sub tree describes a malware family.

In the similarity matrix Z we seek the greatest similarity, we look to which malware behaviours or which intermediary nodes that similarity belongs and regroup the two malware behaviours or intermediate nodes. We add this group in the similarity matrix and remove the previous selected nodes or malware behaviours. Furthermore we link the selected nodes which each other and put this sub tree in the phylogenetic tree. We continue this process until the similarity matrix contains no elements. The pseudo code can be found in figure 4.

An example is shown in the figures 6, 7, 8, 9. The matrix in figure 6 is the initial similarity matrix. The smallest value in that matrix is 1 which fulfils the condition $src \neq dst$. The nodes B and C are grouped form the sub-tree in figure 6. In figure 7 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 due to the fact that $3 < 5$. The merging process is continued like it shown in figure 7 and 8. Finally the resulting sub-trees of figures 6, 7 and 8 are interconnected and a tree emerges, see figure 9.

Figure 4: Pseudo code

```
while(size(Z) ≠ 0){
        (src, dst, sim) = minZ_ij such that src ≠ dst
        g = group(src,dst)
        r = addrow(g);
        c = addcolumn(g);
         for (j = 0; j < size(Z);j++){
                m_1 = get_sim(src,j)
                m_2 = get_sim(dst,j)
                if (m_1 > m_2)
                        s' = m_2
                else
                        s' = m_1
                set_row(r,j,s')
                set_column(c,j,s')
        }
        remove_row(src)
        remove_row(dst)
        remove_col(src)
        remove_col(dst)
        add_tree(g)
}
```

Figure 5: Similarity matrix Z

	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

Figure 6: Group B and C

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

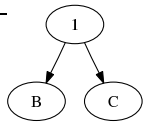


Figure 7: Group A and D

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

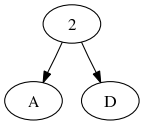


Figure 8: Group AD and BC

	AD	BC
AD	0	3
BC	3	0

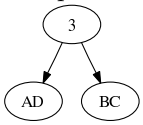
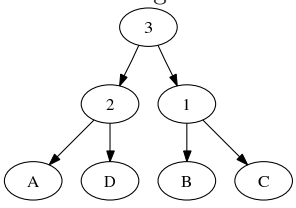


Figure 9: Joining the sub-trees



4 Experimental Validation

In order to examine similarities between behaviours of malware we first need to extract behaviours of malware like it is described in section 2. We check if our virtual operating system is resistant to common anti-reverse engineering techniques. Then we execute a set of 104 random chosen, different malwares. Each malware is identified by its *md5 hash* [15]. We observe the execution messages of the virtual operating system and build sequences of a malware's called functions. Using this sequences we generate malware pairs and computed a similarity for each pair. We have a look at the pairs and create a top 10 list of common malwares behaviours. We also established a list of the top 10 most exotic malware behaviours.

4.1 Anti-reverse engineering techniques resistance

On our malware samples we noticed that static analysis was not possible on 18% of the malware. We checked this by observing the exit code of a disassembler *objdump* [13]. Furthermore according to the *Norman sandbox* [12] 15% of the malware use anti-emulation code. Anti-emulation code can be used to detect or confuse debuggers and monitoring tools. Results are represented in table 2. We have created some binaries that contain anti-reverse engineering code and we have included a function call that should be observed. Then we have used analysis tools and tried to find the defined function call. At first we detect a debugger using the processor flags and the code changes its behaviour in case a debugger is running. Next we generated machine instructions during execution and executed them. Then we created obfuscated assembler code which cannot be read by *objdump*. We have also tested the code integrity check, this technique detects debuggers. 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 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*.

Table 2: Anti-reverse engineering techniques used with various reverse engineering tools

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

4.2 Malware execution

After having checked the virtual operating system for common anti-reverse engineering techniques, we executed our malwares in the virtual operating system. Table 3 shows general information about our examined malware set. We have used quite recent malware. The malware was caught by nepenthes [11] a software which emulates known exploitable services and catch malwares that try to exploit such a service. Furthermore the malware is scanned by anti-viruses[2] and 22% of the malware is not detected. Finally 34% of the malware are worms.

4.3 Malware behaviour similarity

For the sake of clarity, table 4 is an incomplete, list of malware behaviours that have a similarity σ of 1 or in other words they are completely similar. In the first row we see that the malware was transformed to escape from signatures and a new signature was the anti-virus reply. Next we can observe that the

[2] http://www.fprot.org
http://www.bitdefender.com
http://www.free-av.de
http://www.clamav.net

Table 3: General information about the malware set

Number of malwares	104
Observation period	2005-2007
Malware from 2005	10
Malware from 2006	91
Malware from 2007	3
Average file size	135KB
Smallest file	8KB
Biggest file	665KB
Worms	34%
not detected by anti-virus	22%

Table 4: Most Similar observed malwares

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/Virut.A virus
Trojan.Gobot-4	Trojan.Downloader.Delf-35
Trojan.Mybot-5011	Trojan.IRCBot-121
Trojan.Mybot-5079	Trojan.EggDrop-5

malware *Sdbot1234944.1* and the malware *Backdoor-Server/agent.aew* start the same sequence of functions during execution.

Table 5 gives an overview about the average similarity $\bar{\sigma}$. The left part of the table shows the malware behaviours with a low average similarity so these behaviours are not similar regarding the other behaviours of the analysed malware set. Some of those malwares did only a few function calls. We can analyse these function calls and learn new malware techniques and can evaluate our virtual operating system.

The right side indicates the malware behaviours that have been often observed, these behaviours have a high similarity σ. From anti-viruses results we observed that 34% of the malware are worms which explains the high similarity for the worms. A worm often exploits a service and needs to acquire information about the operating system functions using the functions *LoadLibrary* and *GetProcAddress*.

Table 5: 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

4.4 Malware behaviour phylogenetic tree

From our malware set we have build a similarity matrix. From that matrix we have established a phylogenetic tree. In this paper, the complete tree is not represented due to space constraints but we are presenting some sub-trees. The complete tree is on-line[3]. The tree is very large and a user has to scroll through the tree.

From the phylogenetic root's tree's point of view, see figure 10, we observe that we have three families. One family can be called the malware kernel family and the other one the malware W32 API[4] family. The malware of the first family use direct kernel functions in order to escape from API hooking, a technique for monitoring function calls, and the members of the W32 API family use W32 API functions. The third family clusters the malwares that confused the virtual operating system. The three families are not very similar, σ' is close to 1, which can be explained that each family uses a different set of system functions. Malware labelled *unknown* were not detected by anti-viruses.

In figure 11 we have identified a malware family called *pinfi* by the Norman sandbox. This family is a sub-tree of the API family. This malware has had various behaviours but is classified in a same family due to the fact that these malware start similar functions. The *pinfi* malware creates various files on the harddisk, creates different registry keys, monitors which functions are started by the user and does some

[3] http://nepenthes.csrrt.org:10080/malware_behaviour/cache
[4] Application Programming Interface

IRC[5], observes the users clipboard and checks some processor features in order to know whether it runs in an emulator or on a real machine. We observe different similarities σ' due to the fact that the malware behaves differently in the virtual operating system.

Figure 10: Phylogenetic tree root

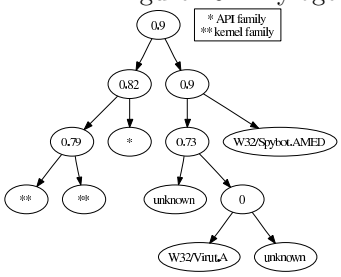
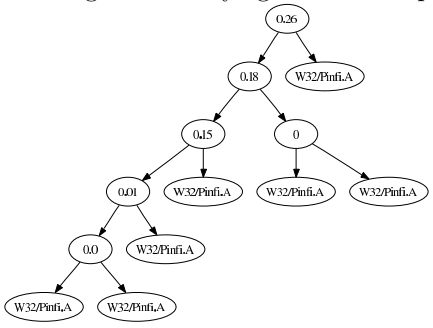


Figure 11: Phylogenetic tree - pinfi familiy



5 Related work

Anti-virus software frequently use signature matching techniques for detecting malware which can be easily detoured [17, 2]. The authors of [2] propose a mean to undo obfuscated malware code in order to improve the detection rate of anti-virus scanners. The authors of [1] propose a method of detecting polymorphic malware based on sub graph isomorphism problem. Behavioural malware analysis often extract API function calls or system function calls [17, 19, 18, 9, 8]. One technique is to detect anomalies or deviations from learnt system / API function call sequences [9, 8, 18, 19]. The authors of [19] represent a malware behaviour as sequence of events and apply a clustering approach. The authors of [18] construct a probabilistic suffix tree that contains the probability of a function call in a sequence of function calls. For accessing the function calls of a malware static binary analysis can be used [17] or malware can be emulated [9, 12, 21]. The authors of [8] use a hybrid technique. Static binary analysis can be easily fooled [17]. On the other hand malware can be emulated [9, 21]. Emulation has also its drawbacks [5, 3]. There is often no guarantee whether the initial conditions are fulfilled that the complete binary is executed. Wilson [22] proposed sequence alignment methods on daily activities patterns. [17] applied this technique for generating malware signatures and has observed good results for detecting obfuscated malware variants. According to PEiD [14], a tool that is able to detect the compression technique, 47% of the malwares in our set that contains 1966 malwares during the experiment, use unknown compression techniques. Furthermore for a common packer like UPX [10] there are 41 different versions without compatibility. Most packers are open-source and the malware author has the possibility to write his custom packer. Decompression needed for [17] becomes thus impossible for our collected malware. Therefore we used the emulation technique. The price we pay is the completeness of the function call sequence which depends on the execution heuristics. Another difference of this paper regarding [17] is that we use the sequence alignment method for malware classification and not for generating signatures. A final difference is that we deduce the similarity directly from the alignment matrix. Further information about theoretic and practical viruses are explained in [4].

[5]Internet Relay Char RFC 2810

6 Conclusion and future works

In this paper we propose an approach to execute malware in a virtual operating system based on freely available tools. We define a malware behaviour as sequence of virtual operating system function calls. Such a behaviour is extracted from a malware and

we introduce a mean to compute similarities between malware behaviours based on sequence alignment. Using these similarities we build a similarity matrix. We propose a mean for detecting new malware techniques based on the low average similarity σ of malware behaviours. From the similarity matrix we create a phylogenetic tree in which we can identify malware families based on common behaviours. Finally we have tested our approach with collected malware.

The analysis results of a malware heavily depend on the capabilities of the virtual operating system. There are various research problems open that we are trying to solve in future.

Execution heuristics. Most malware never terminate so we abort the execution after 10 seconds. An example is a malware which includes a backdoor and which waits for commands. A solution is a token execution system. A malware has initially a credit of n tokens. A token is consumed per called function. When various phenomena are observed the execution controller gives more tokens to the virtual operating system in order to continue the malware execution.

Evaluation of similarity. We get a low similarity 4 in case a malware does the same actions than another malware but in a different order. A solution to that problem is to align the sequences first and then compute the similarities [22]. In future work we plan to evaluate other similarity functions and possible distance functions.

Solving scalability problems. Currently, we have tested our approach on a set of 104 malwares. We have access to a collection of 12000 different malwares[6] and we intend to analyse malware behaviours at a larger scale than in a range of 104 different malwares.

API oriented distance functions. We observe the function calls done by a malware. Currently every function call is treated equally but we think that some function are more important than other ones and intent to attach weights to API functions with respect to similarity computation.

Code mapping. For each function we assign a code. We do not consider the parameters of functions. The reason is that various functions take addresses as parameters and these addresses change from execution to execution. We need to find a more abstract representation of the parameters which is independent of the machine. Another problem is multi-threading. The sequence of functions called by a multi-threaded malware depends on the choices done by the process scheduler which decrease the quality of our sequences. Finally the sequence length becomes a problem in case a malware is executed for a long period. One can imagine to shorten the sequences by detecting loops.

[6] caught by CSRRT's collectors

7 Acknowledgements

We would like to thank Mr Eric FILIOL research professor at ESAT for the pointers and suggestions concerning the related works in the area.

References

[1] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Recognizing self-mutating malware by code normalization and control-flow graph analysis. *IEEE Security & Privacy*, 2007. in press.

[2] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. November 2005.

[3] Peter Ferrie. Attacks on virtual machine emulators. Symantec Advanced Threat Research.

[4] Eric Filiol. *Les virus informatiques : théorie, pratique et applications*. Springer Verlag, 2004.

[5] Richard Ford. The future of virus detection. In *Information Security Technical Report*, pages 19–26. Elsevier, 2004.

[6] Alexandre Julliard. wine. http://www.winehq.com.

[7] J. Kim and T. Warnow. Tutorial on phylogenetic tree estimation, 1999.

[8] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. *Behavior-based Spyware Detection.* Vancouver, BC, Canada, August 2006.

[9] A. Mounji M. Swimmer, B. Le Charlier. Dynamic detection and classification of computer viruses using general behavior patterns. *Proceedings of the 5th International Virus Bulletin Conference*, pages 75–88, 1995.

[10] John F. Reiser Markus F.X.J. Oberhumer, Laszlo Molnar. Upx. http://upx.sourceforge.net/.

[11] Nepenthes. http://nepenthes.mwcollect.org.

[12] Norman. http://sandbox.norman.no.

[13] Objdump. http://www.gnu.org/software/binutils.

[14] Peid. http://peid.tk/.

[15] R. Rivest. Md5 message-digest algorithm. *RFC1321*, 1992.

[16] Secure shell. http://www.openssh.com.

[17] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. *Static Analyzer of Vicious Executables (SAVE)*, pages 326–334. IEEE Computer Society, Washington, DC, USA, 2004.

[18] G. Mazeroff V. De Cerqueira J. Gregor M. G. Thomason. Probabilistic trees and automata for application behavior modeling. *Proceedings of the 43rd ACM Southeast Conference*, 2003.

[19] Jigar J.Mody Tony Lee. Behavioral classification. Eicar, May 2006.

[20] User mode linux. http://user-mode-linux.sourceforge.net.

[21] Matthew Evan Wagner. Behavior oriented detection of malicious code at run-time, 2004. Florida Institute of Technology.

[22] Wilson. Activity pattern analysis by means of sequence-alignment methods. *Environment and Planning*, pages 1017–1038, 1998.