Journal of Information Processing Systems, Vol.7, No.1, March 2011 DOI : 10.3745/JIPS.2011.7.1.173

Virus Detection Method based on Behavior Resource Tree

Mengsong Zou*, Lansheng Han*, Ming Liu* and Qiwen Liu*

Abstract—Due to the disadvantages of signature-based computer virus detection techniques, behavior-based detection methods have developed rapidly in recent years. However, current popular behavior-based detection methods only take API call sequences as program behavior features and the difference between API calls in the detection is not taken into consideration. This paper divides virus behaviors into separate function modules by introducing DLLs into detection. APIs in different modules have different importance. DLLs and APIs are both considered program calling resources. Based on the calling relationships between DLLs and APIs, program calling resources can be pictured as a tree named program behavior resource tree. Important block structures are selected from the tree as program behavior features. Finally, a virus detection model based on behavior the resource tree is proposed and verified by experiment which provides a helpful reference to virus detection.

Keywords—Computer Virus, Behavior-Based Detection, Dynamic Link Library, Behavior Resource Tree

1. INTRODUCTION

Most current anti-virus software employs a signature-based detection technique which is implemented under the premise that virus signatures are already known [1]. However, as the outbreak of polymorphic viruses and other new kinds of viruses, it is difficult to collect all the viruses' signatures. Because of the disadvantages of signature-based detection, behavior-based virus detection techniques have made rapid progress in recent years [2].

Researchers have proposed several behavior-based virus detection methods, such as Bayes, Support Vector Machine, Decision Tree and so on [3]. D. Wagner and D. Dean drew a control flow graph by analyzing program source code which is hard to obtain [4]. J-Y. Xu [5] and J. Bergeron [6] monitored program behavior by taking API calling sequences as behavior features. However, this method treated every API equally. In other words, different APIs were considered to have equivalent importance. Yet, because of the deviation in design purpose, viruses and benign programs must show differences in their API calls [7]. That is to say, some APIs have distinguishing call frequencies between viruses and benign programs. Thus it is unreasonable to treat all APIs completely equally. E. Stinson and J. Mitchell noticed this and classified API calls by function modules [8]. But they did not mention the different importance among modules. Programs call API functions by loading relevant DLLs while APIs in certain DLLs have certain functions. The special behavior of viruses such as propagation, hide, self-copy, etc. are imple-

Manuscript received August 11, 2010; accepted September 3, 2010.
Corresponding Author: Lansheng Han
* Department of Computer Science and Technology, Huazhong University of Science & Technology, CHINA ({stonezms, hanlansheng}@hotmail.com, {liuming2008, fanko24}@gmail.com)

mented by calling APIs from specific DLLs, which demonstrates that these APIs are relatively more important than APIs in other DLLs. Therefore, we can distinguish different APIs' impact on detection by finding the DLL to which it belongs. This is the reason that we introduce DLLs into virus detection. Furthermore, earlier detection methods only took API calls as virus features. In this paper, we consider both DLLs and APIs as resources called by programs instead and draw behavior resource trees based on the resources called by the program and their calling relationship. By building connections between isolated resources, special block structures with several resources can be constructed and defined as program behavior features to increase detection rate accuracy.

This paper is organized as follows: Section 1 is the introduction and reviews the previous work in the field and presents the paper's main ideas. Section 2 illustrates the importance of introducing DLLs into virus detection with virus key DLLs listed and then introduces how to construct behavior resource trees. Section 3 presents the concept of key block structure and the method of obtaining block structure; also included is the reason for considering it as a behavior feature. Section 4 provides elaborations on the virus detection model based on the behavior resource tree, including the methods for calculating the Detection Value of block structures and of entire trees, the detection rule is also listed. In section 5 our theory is verified by an experiment. Section 6 concludes the paper.

2. PROGRAM BEHAVIOR RESOURCE TREE

2.1 Key Dynamic Link Library

In order to achieve specific aims such as propagation, destruction and hide, a virus must perform some behaviors that cause them to differ from benign programs [9]. Those behaviors can be considered as virus features and the majority of them are implemented by calling APIs in an operation system, which illustrates that the basic element of virus behavior is API calls. As APIs are called from Dynamic Link Library (DLL) while different DLLs have relatively independent function definition, APIs from different DLLs show significant difference in their functions. In other words, program behaviors are divided into several function modules, each modules' function is fulfilled by a group of DLLs. Virus, as a special kind of program, must has its own function modules which leads to a calling inclination to some group of DLLs compared to benign program. Those virus-preferred DLLs tend to include more important API functions in virus detection called key APIs. On the contrary, benign-program-preferred DLLs usually have fewer key APIs. Therefore, a DLL can be considered as a kind of behavior feature and hence it is not enough to monitor only API calls but also necessary to include DLLs into virus detection.

Virus behavior features consist of and can perform the following: infection, propagation, self-protection, destruction of system function or user files, stealing sensitive information, unauthorized system control and malice net resource occupation, etc. [10]. According to these behaviors, we can search for DLLs that are included in corresponding function modules and utilize them as key DLLs in virus detection. In order to increase detection efficiency, we only select those DLLs with high frequency of being called and eliminate those of lower call frequency. Several common key DLLs with their functions and importance levels are listed in Table 1.

Table 1. Key DLLs with their functions and importance levels

DLL name	Function	Importance Level
kernel32.dll	Memory management、I/O and interrupt	1
ntdll.dll	API user mode terminal	1
version.dll	NT system version check and file installation	2
advapi32.dll	Object safety, registry operation and event log	3
shell32.dll	Win 32 shell file, default setting of open web page and file	3
wsock32.dll, ws2_32.dll	Windows Sockets API, internet application	2
wininet.dll	HTTP browse、download	2
netapi32.dll	Windows net API	2
netman.dll	Network connect management	2
netrap.dll	Remote network management protocol	3
rpcrt4.dll	Remote procedure call protocol	4
iphlpapi.dll	IP helper module	3
secur32.dll	Microsoft security support provider interface	4
samlib.dll	Microsoft security authority manager API library	4
wintrust.dll	Microsoft trust verification APIs	4
ole32.dll, oleaut32.dll	Object linking and embedding module	2
crypt32.dll	Encryption API32	3
shlwapi.dll	Shell light-weight utility library	2
powrprof.dll	Power profile helper	4

2.2 Construction of Behavior Resource Tree

In current operation systems, programs need to request operation system to execute critical operations. System provides many APIs in all kinds of DLLs for user application to interact with it. Thus, by monitoring the program running process, we could get all the DLLs and APIs called by the program as well as the calling relationships between DLLs and between DLLs and APIs. Based on this information, program resources and their calling relations can be described as a tree structure. In this paper, we call this a Behavior Resource Tree (BRT) as is shown in Fig. 1 below:

From this figure, we realize that all the resources called by a program, including DLLs and APIs, altogether construct a tree structure. High layer DLLs can call DLLs in the lower layer, but call relations are limited to adjacent layers, which means a DLL node's direct child DLL

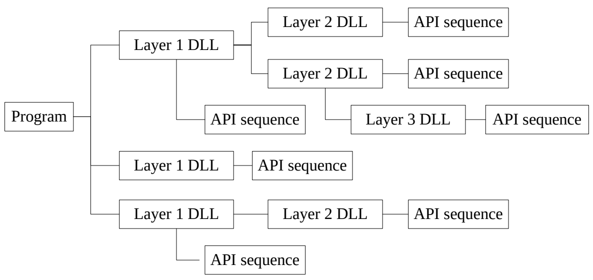


Fig. 1. Diagram of Behavior Resource Tree

nodes are located in the same layer. All the DLLs called by the program form the non-leaf nodes and every DLL calls API sequences that compose the leaf nodes in the tree. In fact, the BRT divides all of a program's behavior into several structures that represent different function modules. Through the calling relationships between DLLs and APIs from higher layers to lower layers, these modules ultimately connect with a basic behavior realization element -API call sequence; thus the entire tree structure of a program is constructed.

3. VIRUS BEHAVIOR FEATURE SET

Most current virus behavior detection methods take API calls as program behavior features, which, as is mentioned above, have many shortcomings. Due to this limitation, we extract special parts from the BRT that are relatively more important in virus detection to serve as key block structures. In this paper, such key block structures are considered program behavior features. Compared to traditional behavior detection methods, there are several reasons to expand the definition of program behavior features:

- Connect isolated resources to monitor programs' behavior more precisely. Because not only APIs are considered as behavior features but also DLLs are taken into consideration, therefore all the resources a program calls construct a tree structure. Thus the monitor range is expanded and the interrelation of resources is defined which means whether a resource exists is not the only factor to consider. All the isolated resources are connected and located in the BRT so that we can better understand the status of every resource in the whole structure and a program's behavior can be divided into different function modules and monitored separately.
- Differential treatment against APIs in different function modules. APIs are treated as equal importance factors in current detection techniques. However, in this paper, taking block structure as a behavior feature actually results in the differential treatment of APIs in different function modules. Every module has its own function and shows a significant difference in virus detection while all modules' functions are implemented by APIs ultimately. As a result, APIs in important modules and less important modules should not be treated equally. Differential treatment is definitely more practical and more accurate.
- Increase detection efficiency. Compared to the whole BRT, one single block structure is only a small part of it. Taking block structure as a behavior feature vector allows us to avoid calculating the whole tree. Through eliminating redundant parts and extracting only key parts, more pointed virus detection could be realized to increase detection efficiency.

In order to extract those relatively more important parts, that is, key block structures and eliminate redundant parts, we first search in the BRT for key DLLs listed in the last section. While a match is found, consider that DLL as a root node and the root node's sub structure in the BRT is a key block structure that we want. During the extraction procedure, a key DLL might be found in several different places of one program and there are more or less some differences among its sub structures. Therefore, the principles of block structure extraction are proposed as follows:

Principle 1: While a DLL has two or more sub structures that have significant difference in their structure, preference in key block structure selection is given to those sub structures whose module function is most probably related with virus behavior or whose hierarchical structure is more complex.

Principle 2: If there are several key DLLs being called in one selected block structure which leads to a huge block size and a much-too complex hierarchical structure, extract non-root DLL key nodes and their sub structure to construct other new key block structures and eliminate these parts from the original selected block structure.

After key block structures are extracted from the BRT, we need to make some secondary treatment against them based on statistics. Due to function difference, a key block structure may have several different versions in different programs. These versions have very familiar structures and unapparent resource differences. In the secondary treatment, they are considered as one key block structure because they perform the same function. Thus we need to select high repetitive rate parts from them, that is, parts which appear in most versions and eliminate those uncommon DLLs and APIs based on statistics through all block structures. At last, the final key block structures are called Feature Block Structures and are stored in a Virus Behavior Feature Database.

4. VIRUS DETECTION MODEL BASED ON THE BEHAVIOR RESOURCE TREE

In this paper, we introduce the concept of Detection Value (DV) into our virus detection model. By assigning values to resources in the BRT, the DV of both block structure and the whole tree can be evaluated. Then we set threshold value through calculation and statistics of all kinds of programs. Compare DV of the testing program to the threshold values, we could get the Whole Detection Rate (WDR). The detection includes three main procedures: calculating the DV of a block structure, calculating the WDV of a BRT and detection rules.

4.1 Calculating DV of Block Structure

In order to calculate the DV of a block structure, we divide the whole block structure into several sub structures from leaf layer to higher layers. Define the sub structure set as $\{T_i\}$.

$$T_i = \{\mathrm{D};\mathrm{V};\mathrm{DD};\mathrm{DA}\}$$

$\mathrm{D} = \{d_1, d_2, \ldots d_n\}$ represents DLL set, where d_i is a DLL in the structure and n is the number of all key DLLs. As is mentioned above, key DLLs show different importance in virus detection because of their function and call frequency differences, which is why DLLs should not be treated equally. In this paper, this differential treatment is realized by assigning different calculating coefficients to DLLs. By building connections among resources of the program, the DLL node actually brings additional information into detection. Therefore, for $\forall d_k \in D$ assign calculating coefficients $P_k, P_k > 1$.

$\mathrm{V} = \{v_{11}, v_{12}, \ldots v_{ij}\}$ is the set of all API nodes included in block structure T_p, where v_{ij} represents the j th child API node of DLL node d_i. In order to assign value to API nodes, we make a statistics through all Feature Block Structures in the Virus Behavior Feature Database, count each of their API amounts n and for $\forall v_{ij} \in \mathrm{V}$, assign value $W(v_{ij}) = 1/n$.

$\mathrm{DD} = \{(d_1 : \{d_p, \ldots d_q\}), \ldots, (d_n : \{d_g, \ldots d_f\})\}$ indicates the call relations among DLLs in the block structure, for example, $(d_1 : \{d_p, \ldots, d_q\})$ means d_1 is the parent node of $d_p, \ldots, d_q$. In other words, this relation set includes all the edges whose two vertices are both DLL nodes.

$\mathrm{DA} = \{(d_1 : \{v_{11}, \ldots v_{1j}\}), \ldots, (d_n : \{v_{n1}, \ldots, v_{nm}\})\}$ is a set indicating the call relations between DLLs and APIs, that is, pointing out which certain sets of APIs are called from which DLL, such as $v_{n1}, \ldots, v_{nm}$ are child nodes of d_n. Unlike DD, DA includes the rest of the edges in the structure which has one DLL vertex and one API vertex. Thus this set actually indicates all the edges that connect leaf nodes.

After the definition of block structures and value assignment of nodes, we search in the testing program for Feature Block Structure. The search procedure first compares the testing program with the Virus Behavior Feature Database to identify ‘similar’ block structures. In order to fulfill this comparison, a Similarity Judgment rule is needed:

Similarity Judgment: The prerequisite of ‘similar’ is two block structure have the same root DLL node. While the testing block structure has the same DLLs as the feature block structure and the whole relationship among those DLLs is also the same, which means they have exactly the same structure, we determine the two block structures are ‘Completely Similar’. If the structure is not completely the same, count the amount of its DLLs which appear in both the testing structure and feature structure. If this amount accounts for more than C percent of its total DLL number, we determine the two block structures are ‘Partially Similar’. The value of C is decided by experiment result. Both of the two circumstances above are considered ‘similar’ in detection.

After getting the ‘similar’ testing block structure by Similarity Judgment rule, we still cannot calculate the DV of original testing block structure. A match between testing block structure and feature block structure in the database is needed to make a secondary treatment against testing block structure. First, we search in the Feature Block Structure Database for a feature structure which has the same root DLL node with testing block structure and call this feature block structure a Compare Block Structure. Then a traversal of the testing block structure is performed. During the traversal, those nodes of the same name located in the same position of the testing block structure and the corresponding compare block structure are allowed to remain in the structure without a location change, while the rest of the nodes which fail to pass the matching and are excluded from the testing structure. When the whole matching process is completed, those parts contribute less to the detection are eliminated and the remaining parts show more significant importance in virus detection. So the matching procedure increases detection accuracy and reduces computation complexity. After the matching, we call the altered testing block structure a Calculable Block Structure.

For every calculable block structure, we first connect its leaf node and their direct parent node as sub structures. Any sub structure can then be considered as a whole and becomes a new leaf of higher layer. Based on this assumption, we could construct sub structures from lower layer to higher layer. Lower layer sub structures are considered as special ‘nodes’ of higher layer sub structures. Eventually, all these sub structures form one whole Calculable Block Structure. According to the assigned value of APIs and calculating coefficients of DLLs, we begin the calculations from leaf layer. The DV of a leaf sub structure constructed by APIs and their parent DLL node can be calculated as:

$$W(T_i) = P_i \sum_{j=1}^{n} W(v_{ij}) \tag{1}$$

$W(v_{ij})$ is the value of leaf API node v_{ij}, P_i is the calculating coefficient of the parent DLL node of those APIs. From the formula above, we can calculate the product of the sum of all leaf API values and their parent DLL calculating coefficients as the DV of the leaf sub block structure. Because all nodes are connected directly or indirectly with each other by edges in the structure, it is insufficient to consider every node as an isolated element and accumulate all the nodes' values to be the DV of the block structure. Meanwhile, every edge connects a parent node and its child node, that is to say, there are no two nodes in the same layer that can become an edge's two vertices. Therefore, by multiplying the calculating coefficient P_i, it can be revealed that the relationships among nodes in the structure actually produce extra information for detection and we could also take P_i as an extra contribution of introducing the concept of edge in virus detection.

The leaf sub block structure can be considered an integrated leaf 'node' of the higher node with a known DV. Then we can reuse the algorithm above to calculate the DV of the higher layer sub block structure. Through calculating the DV of nested sub block structures from lower layers to higher layers, we get the total DV of the calculable block structure:

$$W(T_n) = P_n\left(\sum_{i=1}^{k} W(T_i) + \sum_{j=1}^{m} W(v_{nj})\right) \tag{2}$$

It is to be noted that nearly all DLLs have two kinds of child nodes: API nodes and DLL nodes except for leaf layer DLLs. For those leaf sub block structures whose root nodes are leaf layer DLLs, we can calculate their DV with formula (1). Formula (2) calculates other block structures' DVs where both the value of child APIs and DV of its sub structures should be added up then multiply that root DLL's calculating coefficient to get this sub block structure's result. Using that result in the calculation of higher layer sub block structures, finally we could get the result of the highest layer-that is the DV of that whole calculable block structure.

4.2 Calculating WDV of the BRT

In this paper, a program's behavior is represented by several feature block structures. Thus, in order to calculate the WDV of the BRT, all the program's testing block structures should be calculated altogether with an algorithm. In Section 4.1, we can get a single block structure's DV. Since the DV of different block structures have no comparability with each other, we need to compare the DV of the testing block structure to the DV of its corresponding feature block structure to get a proportion before calculating the total DV of the BRT. This proportion indicates a testing block structure's probability of exhibiting virus behavior and is called Detection Rate (DR).

After calculating the testing block structure's DV $W(T_i)$ and its corresponding feature block structure's DV $W_0(T_i)$, $W(T_i)$ divided by $W_0(T_i)$ is the testing block structure's DR L_i:

$$L_i = \frac{W(T_i)}{W_0(T_i)} \tag{3}$$

As a behavior feature, different block structures apparently show significant different importance in virus detection. So we need to assign a Block Structure Coefficient (BSC) $q_i\,(0 < q_i < 1)$ to every feature block structure based on statistics of their criticality and frequency of occurrence in viruses. The BSC of a block structure actually reflects its percentage in the calculation of total DV.

Adding up all the product of DR L_i and BSC q_i, we get the total detection rate. Then we calculate the total Threshold Detection Rate (TDR) by multiplying TDR $L_{i\text{-}max}$ and BSC q_i. The quotient obtained by dividing the total detection rate by the total threshold detection rate is the Whole Detection Rate R_k of the testing BRT:

$$R_k = \frac{\sum_{i=1}^{n} q_i L_i}{\sum_{i=1}^{n} q_i L_{i\text{-}max}} \tag{4}$$

4.3 Virus Detection Rules

The virus detection method in this paper is based on a program's BRT. Detection rules include searching for possible virus-specific structures in the BRT, judging by the DV of all block structures and judging by the WDR of the BRT. According their order of application, the detection rules are defined as follows:

Detection Rule 1: Search in the program's BRT to find whether there is one or more special structures which can be called virus-specific structure that only exist in virus program while benign program does not have. The form of expression of this kind of structure is certain resources with special connection among themselves.

The method of deciding whether there are virus-specific structures in a BRT is by comparing the testing structure to a virus-specific structure. Because this structure is still a part of the block structure, the Similarity Judgment process can also be used on it. If the two structures satisfy the 'Completely Similar' condition and more than 90 percent of their APIs are the same, we define the testing structure as a virus-specific structure. Any program that has one or more virus-specific structures is judged to be a virus and there is no need to calculate the DV of its block structure or WDR of the BRT. If there is no virus-specific structure in the program, we turn to Detection Rule 2.

Detection Rule 2: Based on the algorithm mentioned above, calculate DV of every testing block structure of the program and obtain its DR L_i. It is necessary to set a Threshold Detection Rate (TDR) $L_{i\text{-}max}$ for every block structure in the Virus Behavior Feature Database which is determined through statistics based upon a large amount of benign program and virus samples. Compare L_i with the feature block structure's Threshold Detection Rate $L_{i\text{-}max}$, if there is one or more structures' DRs which are bigger than their corresponding TDR, the program is judged to be a virus. If all the DRs of testing block structures are smaller than their TDR, we turn to Detection Rule 3. It can also be expressed as:

$\exists L_i \in \{L_1, L_2, \ldots, L_n\}, L_i > L_{i\text{-}max}$, the testing program is judged to be a virus

$\forall L_i \in \{L_1, L_1, \ldots, L_n\}, L_i < L_{i-max}$, turn to Detection Rule 3

Detection Rule 3: If Detection Rules 1and 2 cannot find out whether the testing program is a virus or not, we need to approach the detection from the perspective of the whole BRT. For this reason, all the DRs of testing block structures should be calculated altogether to get the WDR and the last detection rule is based on it. After testing a large number of sample programs' WDRs, we will obtain optimal Threshold Whole Detection Rate (TWDR) R_{max} that serves to separate benign programs and viruses through statistics. If the WDR of a testing program is greater or equal to TWDR, the program is judged to be a virus, otherwise it is judged to be a benign program. That is to say:

If $R_k \geq R_{max}$, testing program is judged to be a virus

If $R_k < R_{max}$, testing program is judged to be a benign program

5. Experiment

Based on the idea proposed above, an experiment was performed for verification. First, we used Dependency Walker to analyze all the virus samples collected and to get their BRTs. From these BRTs, all of their DLL resources are stored and sorted by frequency of occurrence. Regarding those most commonly occurring DLLs as root nodes, we start to find corresponding feature block structures from virus samples. After searching among large amounts of virus BRTs and making comparisons among obtained structures, virus behavior feature block structures with high frequency of occurrence were selected and dominated by their root DLL nodes' names.

Since a feature block structure can be seen in many different programs with similar overall

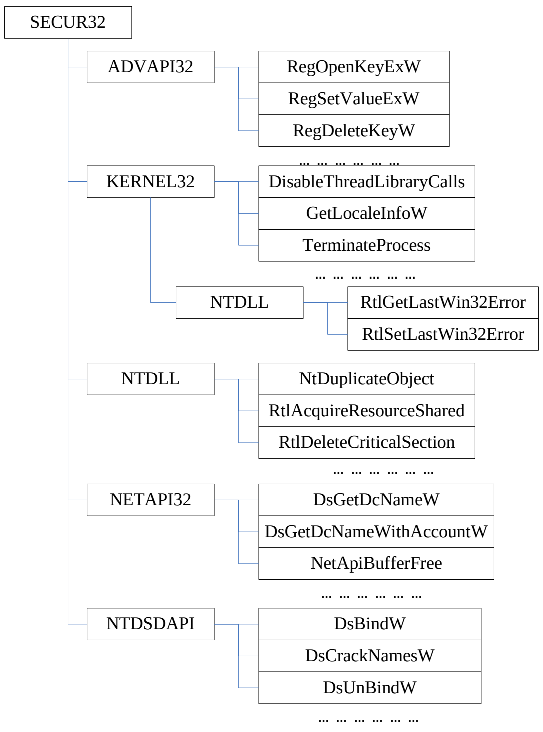


Fig. 2. SECUR32 Block Structure

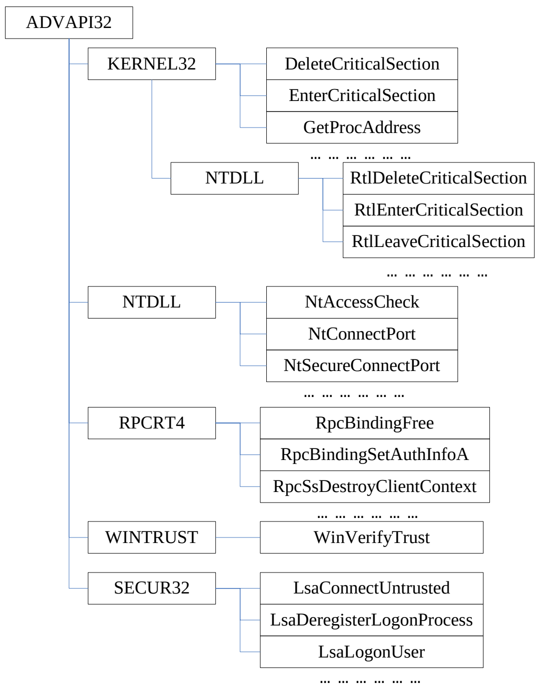


Fig. 3. ADVAPI32 Block Structure

structure but with differences in detail, we call them different versions of one feature block structure. As a result, there are overlapping parts in distinct versions. Feature block structure is determined by selecting the most representative overlapping part from numerous versions. Moreover, with an increase in layers and calling resources, it is much harder to find overlapping parts from different programs. Therefore in this paper we tried to control the feature block structure to be less than five layers to increase detection efficiency and accuracy. Two feature block structures constructed by experiment are shown below:

Subject to space limitations, the block structures above are only a small part of the whole feature block structures and API leaf nodes cannot be completely displayed. In the experiment, we find out that the relation among DLLs in block structures of different programs is similar. That is, the overlapping rate of DLLs is relatively high while the major difference between programs lies in API calls. In order to get more accurate results, the API calls we chose to be in the block structure all existed in more than 50% of the sample programs.

All the selected feature block structures are then used to construct a Virus Behavior Feature Database. Every feature block structure needs to be assigned a calculating coefficient which is determined based on its function and occurrence frequency. After the procedure of search and selection, we obtain 57 feature block structures altogether. Table 2. lists the top 10 most commonly seen block structures along with their calculating coefficients.

In order to test our theory, this paper selects virus: Trojan-Downloader.Win32.EDog for verification. First we analyzed this executable program to get its BRT. According to Detection Rule 1, we search the tree to examine if there are any virus-specific structures. Due to the rarity of virus-specific structures, there were no matches in this virus. Therefore, we turned to Detection

Table 2. Behavior Feature Block Structure Database

Block Structure	Calculating Coefficient	Block Structure	Calculating Coefficient
SECUR32	0.8	POWRPROF	0.6
WINTRUST	0.8	DNSAPI	0.3
RPCRT4	0.8	NTDSAPI	0.4
SAMLIB	0.7	WS2_32	0.4
ADVAPI32	0.7	NETAPI32	0.3

Rule 2 and search the BRT to find block structures which were similar to feature block structures.

Based on Similarity Judgment, those ‘Completely Similar’ block structures and ‘Partially Similar’ ones whose parameters $C > 80\%$ were selected for further matching against feature block structure in the database. After eliminating the redundant parts, we obtained 12 calculable block structures $T_1 \sim T_{12}$: ADVAPI32, RPCRT4, SAMLIB, WINTRUST, SECUR32, SHELL32, POWRPROF, WS2_32, DNSAPI, IPHLPAPI, NETRAP, NETAPI32. The algorithm proposed in Section 5.1 was used to calculate every block structure’s DV $W(T_i)$ and its corresponding feature block structure’s DV $W_0(T_i)$. Then we determined the detection rates L_i of all calculable block structures.

In this table, $L_{i\text{-}max}$ is Threshold Detection Rate of a block structure. Its value is determined by calculating a block structure’s DR of a virus and benign program and choosing the optimal one to distinguish between benign program and virus. In the experiment, we found that as there was an increase in layer number and the complication of structure, the calculated DV was bigger and the detection rate was lower. The reason is that as the structure became more complex, there were more DLL, API calls and function modules. So there were more differences between the testing block structure and feature block structure which lead to the decrease in detection rate. Furthermore, the detection rate of virus block structures which included more key function modules such as RPCRT4, SAMLIB were relatively higher than the other ones which included more non-key function modules. This phenomenon verifies the rationality of dividing program behavior into function modules.

As is shown in Table 3, there was no block structure’s DR exceeding the threshold value. Thus we turned to Detection Rule 3 to calculate the WDR R_k:

$$R_k = \frac{\sum_{i=1}^{12} d_i L_i}{\sum_{i=1}^{12} d_i L_{i\text{-}max}} = \frac{4.087}{5.721} \approx 0.71$$

By calculating the WDR of large amounts of benign programs and viruses, we found that the WDR of benign programs are typically lower than viruses. Statistics demonstrate that the WDR s of benign programs are often lower than 0.7 while the WDR of viruses are usually higher than 0.7. As a result, we set the threshold whole detection rate at $R_{max} = 0.7$. Because $R_k \geq R_{max}$, that testing program is judged to be a virus which proves the validity of the virus detection method based on the BRT as we proposed.

Table 3. DV and DR of Block Structure

T_i	Name	d_i	$W(T_i)$	$W_0(T_i)$	L_i	L_{i-max}
T_1	ADVAPI32	0.7	4.11	5.78	0.71	0.82
T_2	RPCRT4	0.8	3.44	4.98	0.69	0.89
T_3	SAMLIB	0.7	1.77	2.26	0.78	0.93
T_4	WINTRUST	0.8	5.51	8.05	0.68	0.81
T_5	SECUR32	0.8	8.92	16.71	0.53	0.78
T_6	SHELL32	0.6	9.11	22.34	0.41	0.68
T_7	POWRPROF	0.6	5.06	6.86	0.73	0.85
T_8	WS2_32	0.4	2.14	4.77	0.45	0.84
T_9	DNSAPI	0.3	5.26	8.53	0.61	0.80
T_{10}	IPHLPAPI,	0.3	9.67	19.13	0.51	0.69
T_{11}	NETRAP	0.2	0.91	1.17	0.78	0.86
T_{12}	NETAPI32	0.3	6.24	11.21	0.56	0.77

6. Summary and Further Research

Due to the disadvantages of current virus detection methods, this paper proposed a new virus behavior detection method based on a program behavior resource tree. Current detection methods usually take APIs as a program behavior feature and all APIs are considered to have equal importance. However, this paper introduced DLLs into virus detection and divides program behavior into several function modules. Moreover, the paper constructed behavior resource trees reflecting the connections among DLL and API resources. Then critical parts were selected from the tree to form block structures which were considered to be new behavior feature vectors. Based on this, the paper advanced an algorithm to calculate the detection value of single block structures and of whole programs. Three detection rules were defined to accomplish the detection process from the parts to the whole. Ultimately, an experiment was performed to verify that the paper's ideas provide a reasonable solution for virus detection.

In future work, we hope to determine feature block structures and virus-specific structures in formal methods. We also plan to develop a dynamic detection system monitoring program which responds to virus threats with greater speed.

References

[1] Jeffrey O. Kephart and William C. Arnold, "Automatic Extraction of Computer Virus Signatures," 4th Virus Bulletin International Conference, Jersey, USA, 1994, pp.178-184.

[2] Matthew G. Schultz, Eleazar Eskin, Erez Zadok and Salvatore J. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," IEEE Symposium on security and privacy, 2001, pp.38-49.

[3] Jeremy Z. Kolter and Marcus A. Maloof, "Learning to Detect Malicious Executables in the Wild," Proceedings of the tenth ACM SIGKDD international conference, 2004, pp.2721-2744.

[4] David Wagner and Drew Dean, "Intrusion Detection via Static Analysis," Proceedings of the IEEE Symposium on Security and Privacy, 2001, pp.156-168.

[5] J-Y. Xu, A. H. Sung, P. Chavez and S. Mukkamala, "Polymorphic Malicious Executable Scanner by API Sequence Analysis," Proceedings of the Fourth International Conference on Hybrid Intelligent Systems, 2004, pp.378-383.

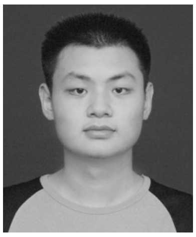
[6] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie and N. TawbiStatic, "Detection of Malicious Code in Executable Programs," Int. J. of Req. Eng., 2001, pp.45-48.

[7] XU Ming, CHEN Chun and YING Jing, "Anomaly Detection Based on System Call Classification," Journal of Software, Vol.15, No.3, 2004, pp.391-403.

[8] Elizabeth Stinson and John C. Mitchell, "Characterizing Bots' Remote Control Behavior. In Detection of Intrusions & Malware, and Vulnerability Assessment," 2007, pp.89-108.

[9] Essam Al Daoud, Iqbal H. Jebril and Belal Zaqaibeh, "Computer Virus Strategies and Detection Methods," Int. J. Open Problems Compt. Math., Vol.1, No.2, 2008, pp.12-20.

[10] F. Cohen, "Computer viruses:: Theory and experiments," Computers & security, Vol.6, 1987, pp.22-33.

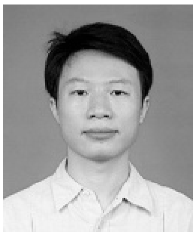


Mengsong Zou

He received his BS degrees in the Dept. of Computer Sci. & Tech. from Huazhong University of Science and Technology in 2008. And now he is undertaking a master's course as a member of the Information Security lab at Huazhong Univ. of Sci. and Tech. His research interests include Virus detection, Net-virus propagation models and information security.

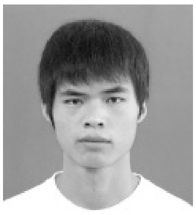
Lansheng Han

He received a Ph.D. degree in Information Security (2003-2006) from the Dept. of Computer Science & Technology, Huazhong University of Science and Technology. Now he is working as an Associate Professor at the School of Computer Science of HUST. His research interests include Information Security, Spreading Models of Viruses, Connection Models and Access Control.



Ming Liu

He received a Ph.D. degree in Information Security from the Dept. of Computer Science & Technology, Huazhong University of Science and Technology. Now he is working at the School of Computer Science of HUST. His research interests include Information Security, Network Security and Computer Viruses.



Qiwen Liu

He received his BS degrees from the Dept. of Traffic Science and Engineering at Huazhong University of Science and Technology in 2008. Now he is undertaking a master's course as a member of the Information Security lab at Huazhong Univ. of Sci. and Tech. His research interests include Computer viruses, Net-virus propagation models and information security.