J Comput Virol (2008) 4:323–334
DOI 10.1007/s11416-008-0082-4

ORIGINAL PAPER

An intelligent PE-malware detection system based on association mining

Yanfang Ye · Dingding Wang · Tao Li · Dongyi Ye ·
Qingshan Jiang

Received: 24 September 2007 / Revised: 8 January 2008 / Accepted: 13 January 2008 / Published online: 5 February 2008

Abstract The proliferation of malware has presented a serious threat to the security of computer systems. Traditional signature-based anti-virus systems fail to detect polymorphic/metamorphic and new, previously unseen malicious executables. Data mining methods such as Naive Bayes and Decision Tree have been studied on small collections of executables. In this paper, resting on the analysis of Windows APIs called by PE files, we develop the Intelligent Malware Detection System (IMDS) using Objective-Oriented Association (OOA) mining based classification. IMDS is an integrated system consisting of three major modules: PE parser, OOA rule generator, and rule based classifier. An OOA_Fast_FP-Growth algorithm is adapted to efficiently generate OOA rules for classification. A comprehensive experimental study on a large collection of PE files obtained from the anti-virus laboratory of KingSoft Corporation is performed to compare various malware detection approaches. Promising experimental results demonstrate that the accuracy and efficiency of our IMDS system outperform popular anti-virus software such as Norton AntiVirus and McAfee VirusScan, as well as previous data mining based detection systems which employed Naive Bayes, Support Vector Machine (SVM) and Decision Tree techniques. Our system has already been incorporated into the scanning tool of KingSoft's Anti-Virus software.

A short version of the paper is appeared in [33]. The work is partially supported by NSF IIS-0546280 and an IBM Faculty Research Award. The authors would also like to thank the members in the anti-virus laboratory at KingSoft Corporation for their helpful discussions and suggestions.

Y. Ye
Department of Computer Science, Xiamen University,
Xiamen, People's Republic of China

D. Wang · T. Li (✉)
School of Computer Science, Florida International University,
Miami, FL, USA
e-mail: taoli@cs.fiu.edu

D. Ye
College of Maths and Computer Science, Fuzhou University,
Fuzhou, People's Republic of China

Q. Jiang
Software School, Xiamen University, Xiamen,
People's Republic of China

1 Introduction

Malicious executables are programs designed to infiltrate or damage a computer system without the owner's consent, which have become a serious threat to the security of computer systems. New, previously unseen malicious executables, polymorphic malicious executables using encryption and metamorphic malicious executables adopting obfuscation techniques are more complex and difficult to detect. According to its propagation methods, malicious code is usually classified into the following categories [1,7,21]: viruses, worms, trojan horses, backdoors and spyware. Malicious executables do not always exactly fit into these categories and the malicious code combining two or more categories can lead to powerful attacks. For instance, a worm containing a payload can install a back door to allow remote access. Due to the significant loss and damages induced by malicious executables, the malware detection becomes one of the most critical issues in the field of computer security.

Currently, most widely-used malware detection software uses signature-based method to recognize threats [8,9]. Signatures are short strings of bytes which are unique to the programs. They can be used to identify particular viruses in executable files, boot records, or memory with small error rate [14]. However, this signature based method is not effective against modified and unknown malicious executables. The problem lies in the signature extraction and generation process, and in fact that these signatures can easily be bypassed. Heuristic-based recognition, relating to more complex signature based detection techniques, tends to provide protection against new and unknown threats, but this kind of virus detection is usually time consuming and still fail to detect new malicious executables.

Our efforts for detecting polymorphic, metamorphic and new, previously unseen malicious executables lead to the Intelligent Malware Detection System (IMDS), which applies Objective-Oriented Association (OOA) mining based classification [19,25,34]. The IMDS rests on the analysis of Windows Application Programming Interface (API) execution calls which reflect the behavior of program code pieces. The associations among the APIs capture the underlying semantics for the data which are essential to malware detection. Our malware detection is carried out directly on Windows Portable Executable (PE) code with three major steps: (1) first constructing the API execution calls by developing a PE parser; (2) followed by extracting OOA rules using OOA_Fast_FP-Growth algorithm; (3) and finally conducting classification based on the association rules generated in the second step. So far, we have gathered 29,580 executables, of which 12,214 are referred to as benign executables and 17,366 are malicious ones. These executables called 12,964 APIs in total. All of these samples are obtained from the antivirus laboratory of KingSoft Corporation. The collected data in our work is significantly larger than those used in previous studies on data mining for malware detection [15,24,30]. The experimental results illustrate that the accuracy and efficiency of our IMDS outperform popular anti-virus software such as Norton AntiVirus, Dr. Web, McAfee VirusScan and KAV, as well as the systems using data mining approaches such as Naive Bayes, support vector machine (SVM) and Decision Tree.

Our approach for malware detection has been incorporated into the KingSoft's AntiVirus software. In summary, our main contributions are: (1) We develop an integrated IMDS system based on analysis of Windows API execution calls. The system consists of three components: PE parser, rule generator and classifier; (2) We adapt existing association based classification techniques to improve the system effectiveness and efficiency; (3) We evaluate our system on a large collection of executables including 12,214 benign samples and 17,366 malicious ones; (4) We provide a comprehensive experimental study on various anti-virus software as well as various data mining techniques for malware detection using our data collection; (5) Our system has been already incorporated into the KingSoft's AntiVirus software.

The rest of this paper is organized as follows. Section 2 discusses the related work. The IMDS system architecture is described in Sect. 3. We, respectively, present our data collection and OOA mining based classification methodology in Sects. 4 and 5. In Sect. 6, we describe the feature selection and classification methodologies that have been used in previous studies. These techniques are used in our experiments to compare with our IMDS system. Section 7 presents and discusses the experimental results. Finally, Sect. 8 concludes.

2 Related work

Besides the traditional signature-based malware detection methods [14,20] as mentioned in the previous section, there is some work to improve the signature-based detection [5,18,23,26] and also a few attempts to apply data mining and machine learning techniques, such as Naive Bayes method, support vector machine (SVM) and Decision Tree classifiers, to detect new malicious executables. Theoretical studies on computer viruses can be found in [35,36].

Sung et al. [26] developed a signature based malware detection system called SAVE (Static Analyzer of Vicious Executables) which emphasized on detecting polymorphic malware by calculating a similarity measure between the known virus and the suspicious code. The basic idea of this approach is to extract the signatures from the original malware with the hypothesis that all versions of the same malware share a common core signature. Although this work improves the traditional signature-based detection in polymorphic malware detection, it fails against the unknown malware.

Schultz et al. [24] applied Naive Bayes method to detect previously unknown malicious code. The authors downloaded 1,001 benign executables and 3,265 malicious executables from several FTP sites and labeled them by a commercial virus scanner. Furthermore, they examined a small data set of 38 malicious programs and 206 benign programs in Windows PE format. The authors compared their results with traditional signature-based methods and claimed that the voting Naive Bayes classifier outperformed other methods.

Decision Tree was studied in [15,30]. In [30], the authors used the same data set described in [24], and worked on a subset of the data which consists of 125 benign programs and 875 malicious code. They conducted experiments with both Naive Bayes and Decision Tree classifiers, and then concluded that the Decision Tree outperformed the Naive Bayes method in detection rate, false positive rate, and accuracy. Kolter et al. [15] gathered 1,971 benign executables and 1,651 malicious executables in Windows PE format, and examined the performance of different classifiers such as Naive Bayes, support vector machine (SVM) and Decision Tree using tenfold cross validation and plotting Receiver Operating Characteristics (ROC) curves [27]. Their results also showed that the ROC curve of the Decision Tree method dominated all others.

The detail classification methodologies of Naive Bayes, SVM and Decision Tree are described in Sect. 6. Although results were generally good, it would be interesting to know how these classifiers performed on larger collections of executables. And also, it seems that no attempt has been made on other learning methods such as association mining.

Different from earlier studies, our work is based on a large collection of malicious executables collected at KingSoft Anti-Virus Laboratory. In the field of data mining, frequent patterns found by association mining carry the underlying semantics of the data, therefore, we applied OOA mining technique to extract the characterizing frequent patterns to achieve accurate malware detection.

3 The system architecture

Our IMDS system is performed directly on Windows PE code. The system consists of three major components: PE parser, OOA rule generator, and malware detection module, as illustrated in Fig. 1.

The functionality of the PE parser is to generate the Windows API execution calls for each benign/malicious executable. If a PE file is previously compressed by a third party binary compress tool such as UPX and ASPack Shell or embedded a homemade packer, it needs to be decompressed before being passed to the PE parser and we use the

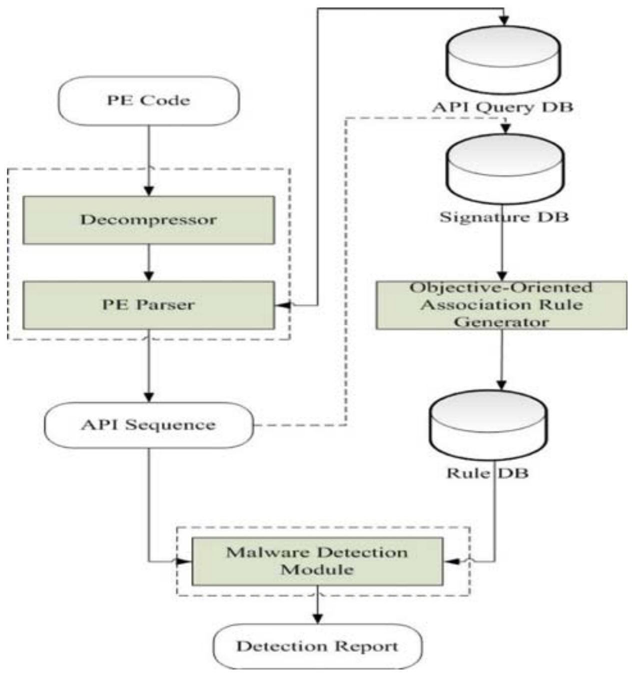


Fig. 1 IMDS system architecture

dissembler W32Dasm developed by KingSoft Anti-Virus Laboratory to dissemble the PE code and output the assembly instructions as the input of our PE parser. Through the API query database, the API execution calls generated by the PE parser can be converted to a group of 32-bit global IDs which represents the static execution calls of the corresponding API functions. For example, the API "KERNEL32.DLL, OpenProcess" executes the function that returns a handle to an existing process object and it can be encoded as 0x00500E16. Then we use the API calls as the signatures of the PE files and store them in the signature database. After that, an OOA mining algorithm is applied to generate class association rules which are recorded in the rule database. To finally determine whether a PE file is malicious or not, we pass the selected API calls together with the rules generated to the malware detection module to perform the association rule based classification. The detail procedures of data collection and OOA mining based classification are described in the following two sections, respectively.

4 Data collection and transformation

PE is designed as a common file format for all flavors of Windows operating system, and PE viruses are in the majority of the viruses rising in recent years. Some famous viruses such as CIH, CodeRed, CodeBlue, Nimda, Sircam, Killonce, Sobig, and LoveGate all aim at PE files.

As stated previously, we obtained 29,580 Windows PE files of which 12,214 were recognized as benign executables while 17,366 were malicious executables. The malicious executables mainly consisted of backdoors, worms, and trojan horses and all of them were provided by the Anti-virus laboratory of KingSoft Corporation. The benign executables were gathered from the system files of Windows 2000/NT and XP operating system.

Since a virus scanner is usually a speed sensitive application, in order to improve the system performance, we developed a PE parser as described in the previous section to construct the API execution calls of PE files instead of using a third party disassembler. However, if a PE file adopts Entry Point Obscuring (EPO) techniques or is previously compressed by a third party binary compress tool such as UPX and ASPack Shell, we firstly use the dissembler W32Dasm developed by KingSoft Anti-Virus Laboratory to dissemble the PE code and output the assembly instructions as the input of our PE parser.

Before proposing our design of the PE parser, let us firstly look at the general outline of PE file format. Figure 2 is the general layout of a PE file. All PE files must start with a simple DOS MZ header, thus DOS can verify whether a file is a valid executable or not when the program is running under DOS system. Next to the DOS header, there is a PE

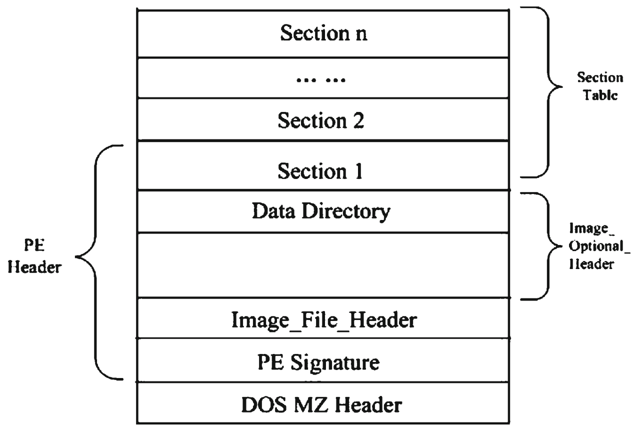


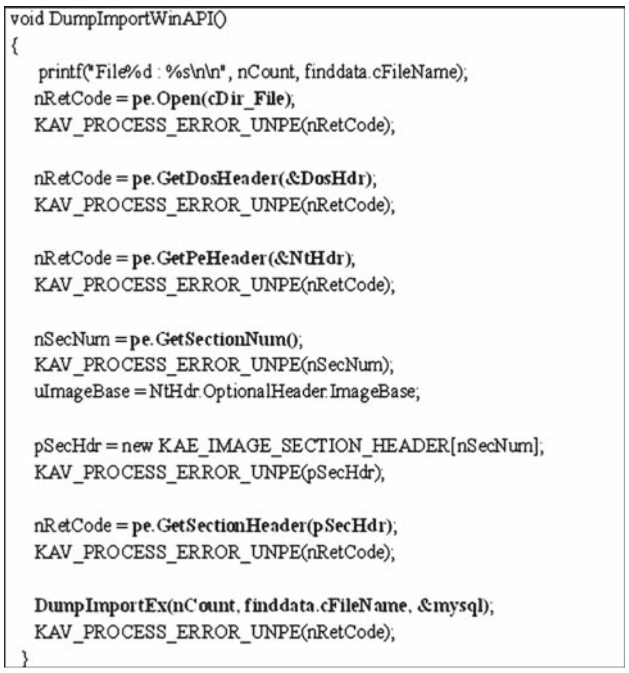
Fig. 2 General layout of a PE file

header which contains essential information used to load a PE file. The content of a PE file is divided into sections and each section stores data with common attributes.

Then we look at the import table from which our PE parser extracts the APIs called by a PE file. If the PE file calls another piece of executable code, we call it an import function. For example, the code of a Windows API is always in the correlative Dynamic Linked Library (DLL) file of the "system32" directory. The pointers to the names of the import functions and their resident DLL are recorded in the Imported Address Table (IAT). In order to extract statically the API execution calls of a PE file, our PE parser performs the following three major steps. First of all, it locates the IAT which contains the pointers to the hints and names of the imported API, and then builds a binary lookup tree from all imported APIs. The second step is to scan the code section(s) to extract the CALL instructions and their target addresses, and then the PE parser searches the target addresses in the binary lookup tree to find the corresponding API. In the final step, we map the API name along with its module name to a 32-bit unique global API ID. For example, the API "MAPI32.MAPIReadMail" is encoded as 0x00600F12. By using integer representation, the costly string comparisons are avoided in later operations.

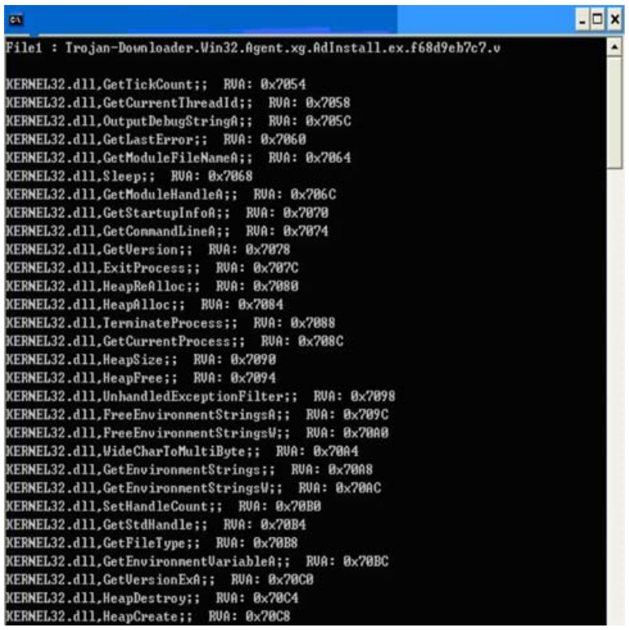
The implementation of our PE parser involves the following procedures:

1. Verifying if the file is a valid PE file;
2. Locating the PE header by examining the DOS header;
3. Obtaining the address of the data directory in Image_ Optional_Header;
4. Extracting the value of VirtualAddress in the data directory;
5. Finding Image_Import_Descriptor structures using the value of VirtualAddress;
6. Checking the RVA value in each Image_Import_ Descriptor structure found in step 5 to locate the arrays which contain the API function names;
7. Extracting API function names.



```
void DumpImportWinAPI()
{
    printf("File%d : %s\n\n", nCount, finddata.cFileName);
   nRetCode = pe.Open(cDir_File);
   KAV_PROCESS_ERROR_UNPE(nRetCode);

   nRetCode = pe.GetDosHeader(&DosHdr);
   KAV_PROCESS_ERROR_UNPE(nRetCode);

   nRetCode = pe.GetPeHeader(&NtHdr);
   KAV_PROCESS_ERROR_UNPE(nRetCode);

   nSecNum = pe.GetSectionNum();
   KAV_PROCESS_ERROR_UNPE(nSecNum);
   uImageBase = NtHdr.OptionalHeader.ImageBase;

   pSecHdr = new KAE_IMAGE_SECTION_HEADER[nSecNum];
   KAV_PROCESS_ERROR_UNPE(pSecHdr);

   nRetCode = pe.GetSectionHeader(pSecHdr);
   KAV_PROCESS_ERROR_UNPE(nRetCode);

   DumpImportEx(nCount, finddata.cFileName, &mysql);
   KAV_PROCESS_ERROR_UNPE(nRetCode);
 }
```

Fig. 3 Core algorithm of PE parser

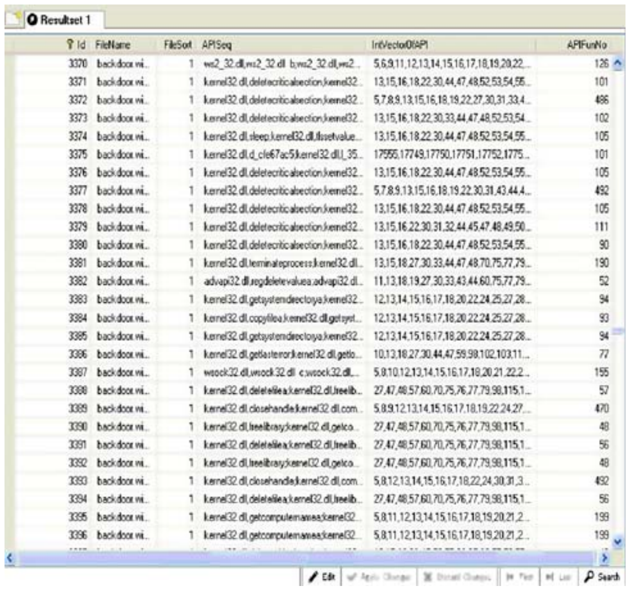


```
File1 : Trojan-Downloader.Win32.Agent.xg.AdInstall.ex.f68d9eb7c7.v

KERNEL32.dll,GetTickCount;;  RVA: 0x7054
KERNEL32.dll,GetCurrentThreadId;;  RVA: 0x7058
KERNEL32.dll,OutputDebugStringA;;  RVA: 0x705C
KERNEL32.dll,GetLastError;;  RVA: 0x7060
KERNEL32.dll,GetModuleFileNameA;;  RVA: 0x7064
KERNEL32.dll,Sleep;;  RVA: 0x7068
KERNEL32.dll,GetModuleHandleA;;  RVA: 0x706C
KERNEL32.dll,GetStartupInfoA;;  RVA: 0x7070
KERNEL32.dll,GetCommandLineA;;  RVA: 0x7074
KERNEL32.dll,GetVersion;;  RVA: 0x7078
KERNEL32.dll,ExitProcess;;  RVA: 0x707C
KERNEL32.dll,HeapReAlloc;;  RVA: 0x7080
KERNEL32.dll,HeapAlloc;;  RVA: 0x7084
KERNEL32.dll,TerminateProcess;;  RVA: 0x7088
KERNEL32.dll,GetCurrentProcess;;  RVA: 0x708C
KERNEL32.dll,HeapSize;;  RVA: 0x7090
KERNEL32.dll,HeapFree;;  RVA: 0x7094
KERNEL32.dll,UnhandledExceptionFilter;;  RVA: 0x7098
KERNEL32.dll,FreeEnvironmentStringsA;;  RVA: 0x709C
KERNEL32.dll,FreeEnvironmentStringsW;;  RVA: 0x70A0
KERNEL32.dll,WideCharToMultiByte;;  RVA: 0x70A4
KERNEL32.dll,GetEnvironmentStrings;;  RVA: 0x70A8
KERNEL32.dll,GetEnvironmentStringsW;;  RVA: 0x70AC
KERNEL32.dll,SetHandleCount;;  RVA: 0x70B0
KERNEL32.dll,GetStdHandle;;  RVA: 0x70B4
KERNEL32.dll,GetFileType;;  RVA: 0x70B8
KERNEL32.dll,GetEnvironmentVariableA;;  RVA: 0x70BC
KERNEL32.dll,GetVersionExA;;  RVA: 0x70C0
KERNEL32.dll,HeapDestroy;;  RVA: 0x70C4
KERNEL32.dll,HeapCreate;;  RVA: 0x70C8
```

Fig. 4 API calls samples with integer representation generated by PE parser

The six functions of the core algorithm in Fig. 3 illustrate the above procedures implemented in the PE parser. And Fig. 4 shows a sample API calls generated by the PE parser. These APIs are called by a malicious software named Trojan-Downloader.exe.

Once we get the API calls, we can use them as the signatures of the PE files and stored them in the signature database.


Resultset 1

Id	FileName	FileSort	APISeq	IntVectorOfAPI	APIFunNo
3370	backdoor.wi...	1	ws2_32.dll,ws2_32.dll b;ws2_32.dll,ws2...	5,6,9,11,12,13,14,15,16,17,18,19,20,22...	126
3371	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	13,15,16,18,22,30,44,47,48,52,53,54,55...	101
3372	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	5,7,8,9,13,15,16,18,19,22,27,30,31,33,4...	486
3373	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	13,15,16,18,22,30,33,44,47,48,52,53,54...	102
3374	backdoor.wi...	1	kernel32.dll,sleep;kernel32.dll,tlssetvalue...	13,15,16,18,22,30,44,47,48,52,53,54,55...	105
3375	backdoor.wi...	1	kernel32.dll,d_cle67ac5;kernel32.dll,l_35...	17555,17749,17750,17751,17752,1775...	101
3376	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	13,15,16,18,22,30,44,47,48,52,53,54,55...	105
3377	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	5,7,8,9,13,15,16,18,19,22,30,31,43,44,4...	492
3378	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	13,15,16,18,22,30,44,47,48,52,53,54,55...	105
3379	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	13,15,16,22,30,31,32,44,45,47,48,49,50...	111
3380	backdoor.wi...	1	kernel32.dll,deletecriticalsection;kernel32...	13,15,16,18,22,30,44,47,48,52,53,54,55...	90
3381	backdoor.wi...	1	kernel32.dll,terminateprocess;kernel32.dll...	13,15,18,27,30,33,44,47,48,70,75,77,79...	190
3382	backdoor.wi...	1	advapi32.dll,regdeletevaluea;advapi32.dll...	11,13,18,19,27,30,33,43,44,60,75,77,79...	52
3383	backdoor.wi...	1	kernel32.dll,getsystemdirectorya;kernel32...	12,13,14,15,16,17,18,20,22,24,25,27,28...	94
3384	backdoor.wi...	1	kernel32.dll,copyfilea;kernel32.dll,getsyst...	12,13,14,15,16,17,18,20,22,24,25,27,28...	93
3385	backdoor.wi...	1	kernel32.dll,getsystemdirectorya;kernel32...	12,13,14,15,16,17,18,20,22,24,25,27,28...	94
3386	backdoor.wi...	1	kernel32.dll,getlasterror;kernel32.dll,getlo...	10,13,18,27,30,44,47,59,98,102,103,11...	77
3387	backdoor.wi...	1	wsock32.dll,wsock32.dll c;wsock32.dll...	5,8,10,12,13,14,15,16,17,18,20,21,22,2...	155
3388	backdoor.wi...	1	kernel32.dll,deletefilea;kernel32.dll,freelib...	27,47,48,57,60,70,75,76,77,79,98,115,1...	57
3389	backdoor.wi...	1	kernel32.dll,closehandle;kernel32.dll,com...	5,8,9,12,13,14,15,16,17,18,19,22,24,27...	470
3390	backdoor.wi...	1	kernel32.dll,freelibrary;kernel32.dll,getco...	27,47,48,57,60,70,75,76,77,79,98,115,1...	48
3391	backdoor.wi...	1	kernel32.dll,deletefilea;kernel32.dll,freelib...	27,47,48,57,60,70,75,76,77,79,98,115,1...	56
3392	backdoor.wi...	1	kernel32.dll,freelibrary;kernel32.dll,getco...	27,47,48,57,60,70,75,76,77,79,98,115,1...	48
3393	backdoor.wi...	1	kernel32.dll,closehandle;kernel32.dll,com...	5,8,12,13,14,15,16,17,18,22,24,30,31,3...	492
3394	backdoor.wi...	1	kernel32.dll,deletefilea;kernel32.dll,freelib...	27,47,48,57,60,70,75,76,77,79,98,115,1...	56
3395	backdoor.wi...	1	kernel32.dll,getcomputernamea;kernel32...	5,8,11,12,13,14,15,16,17,18,19,20,21,2...	199
3396	backdoor.wi...	1	kernel32.dll,getcomputernamea;kernel32...	5,8,11,12,13,14,15,16,17,18,19,20,21,2...	199

Fig. 5 Sample data in the signature database

As shown in Fig. 5, there are six fields in the signature database, which are record ID, PE file name, file type ("0" represents benign file while "1" is for malicious file), called APIs name, called API ID and the total number of called API functions. The transaction data can also be easily converted to relational data if necessary. Now the data is ready for the next step, i.e., OOA mining based classification to finally achieve the goal of detecting the malware.

5 Classification based on OOA mining

Both classification and association mining play important roles in data mining techniques. Classification is "the task of learning a target function that maps each feature set to one of the predefined class labels" [28]. For association rule mining, there is no predetermined target. Given a set of transactions in the database, all the rules that satisfied the support and confidence thresholds will be discovered [3]. As a matter of fact, classification and association rule mining can be integrated to association rule based classification [4,19]. This technique utilizes the properties of frequent patterns to solve the scalability and overfitting issues in classification and achieves excellent accuracy [4]. In our IMDS system, we adapted OOA mining techniques [25] to generate the rules.

5.1 OOA definitions

Other than traditional itemset correlation oriented association mining, OOA mining models association patterns that are explicitly relating to a user's objective. In our IMDS system, the goal is to find out how a set of API calls supports the specific objectives: $Obj_1 = (Group = Malicious)$, and $Obj_2 = (Group = Benign)$.

- **Definition 1 (support and confidence)** Let $I = \{I_1, \ldots, I_m\}$ be an itemset and $I \rightarrow Obj(os\%, oc\%)$ be an association rule in OOA mining. The support and confidence of the rule are defined as:

$$os\% = supp(I, Obj) = \frac{count(I \cup \{Obj\}, DB)}{|DB|} \times 100\%$$

$$oc\% = conf(I, Obj) = \frac{count(I \cup \{Obj\}, DB)}{count(I, DB)} \times 100\%$$

  where the function $count(I \cup \{Obj\}, DB)$ returns the number of records in the dataset DB where $I \cup \{Obj\}$ holds.
- **Definition 2 (OOA frequent itemset)** Given $mos\%$ as a user-specified minimum support. I is an OOA frequent itemset/pattern in DB if $os\% \geq mos\%$.
- **Definition 3 (OOA rule)** Given $moc\%$ as a user-specified confidence. Let $I = \{I_1, \ldots, I_m\}$ be an OOA frequent itemset. $I \rightarrow Obj(os\%, oc\%)$ is an OOA rule if $oc\% \geq moc\%$.

In order to better explain different OOA mining algorithms, we use a sample file in Table 1 as a simple example in the following subsections.

5.2 OOA_Apriori algorithm

Apriori algorithm [2] can be extended to OOA mining. It can be achieved in two major steps: (1) generate OOA frequent patterns by Apriori algorithm; (2) estimate all of such OOA frequent pattern; if the confidence of the itemset satisfies $oc\% \geq moc\%$, then insert it into the OOA rule set. Given the example data as shown in Table 1, $mos\%$=25%, $moc\%$=65%, and $Obj_1 = (Group = Malicious)$ as the objective, the frequent itemsets by using OOA_Apriori algorithm are: $\{API_1\}, \{API_2\}, \{API_3\}, \{API_4\}, \{API_5\}, \{API_1, APT_3\},$

Table 1 Sample dataset

ID	Called API	File type
1	API_1, API_5	Benign
2	API_1, API_3	Malicious
3	$API_1, API_2, API_4, API_5$	Benign
4	$API_1, API_2, API_3, API_4, API_5$	Malicious
5	API_3, API_5	Malicious
6	API_2, API_4	Benign
7	API_2, API_4, API_5	Malicious
8	API_2, API_5	Benign

$\{API_2, APT_4\}$, $\{API_2, APT_5\}$, $\{API_3, APT_5\}$, $\{API_4, APT_5\}$, $\{API_2, API_4, APT_5\}$. The association rules generated are:

1. $\{API_3\} \rightarrow Obj_1$ (37.5%,100%);
2. $\{API_1, API_3\} \rightarrow Obj_1$ (25%,100%);
3. $\{API_3, API_5\} \rightarrow Obj_1$ (25%,100%);
4. $\{API_4, API_5\} \rightarrow Obj_1$ (25%,66.7%);
5. $\{API_2, API_4, API_5\} \rightarrow Obj_1$ (25%,66.7%).

5.3 OOA_FP-Growth algorithm

Although OOA_Apriori algorithm can be implemented easily, it requires many iterations to generate all of the frequent itemsets before generating the association rules. An alternative OOA mining algorithm called OOA_FP-Growth is designed based on FP-Growth algorithm [10,11]. This algorithm encodes the dataset using an FP-tree structure and extracts frequent itemsets from it. There are two steps in the OOA_FP-Growth algorithm.

1. Construct the FP-tree:
   (a) First of all, select all records satisfying the objective as a sub-dataset.
   (b) Scan the sub-dataset to determine the support count of each item; sort the frequent items in decreasing order of their support counts.
   (c) Create a set of nodes for each record; form a path to encode the record and increase the frequency count of each node along the path by 1.
   (d) Algorithm stops when every record has been mapped onto one of the paths in the FP-tree.
2. Generate frequent itemsets: similar to the FP-Growth algorithm, OOA_FPGrowth generates frequent itemsets by exploring the FP-tree in a bottom-up fashion.

Now, we use the example data file shown in Table 1 to illustrate the OOA_FP-Growth algorithm. Given that the minimum support count is 2 and the objective is $Obj_1 = (Group = Malicious)$, we obtain the sub-dataset as shown in Table 2. Based on the sub-dataset we construct the OOA_FP-tree as illustrated in Fig. 6. Searching the tree in a bottom-up fashion, we generate the following frequent itemsets:
$\{API_2, API_4\}$, $\{API_4, API_5\}$, $\{API_2, API_5\}$, $\{API_2, API_4, API_5\}$, $\{API_1, API_3\}$, $\{API_3, API_5\}$.

5.4 OOA_Fast_FP-Growth algorithm

In general, OOA_FP-Growth algorithm is much faster than OOA_Apriori for mining frequent itemsets. However, when the minimum support is small, OOA_FP-Growth generates a huge number of conditional FP-trees recursively, which is time and space consuming. Our malware detection relies on finding frequent patterns from large collections of data, therefore, the efficiency is an essential issue to our system. In our IMDS system, we extend a modified FP-Growth algorithm proposed in [6] to conduct the OOA mining. This algorithm greatly reduces the costs of processing time and memory space, and we call it OOA_Fast_FP-Growth algorithm.

Similar to OOA_FP-Growth algorithm, there are also two steps in OOA_Fast_FP-Growth algorithm: constructing an OOA_Fast_FP-tree and generating frequent patterns from the tree. But the structure of an OOA_Fast_FP-tree is different from that of an OOA_FP-tree in the following way: (1) The paths of an OOA_Fast_FP-tree are directed, and there is no path from the root to leaves. Thus, fewer pointers are needed and less memory space is required. (2) In an OOA_FP-tree, each node is the name of an item, but in an OOA_Fast_FP-tree, each node is the related order of the item, which is determined by the support count of the item. Based on the sub-dataset shown in Table 2, we construct the OOA_Fast_FP-tree as illustrated in Fig. 7. The support counts from *API1* to *API5*, respectively, are 3, 3, 2, 2, 2 and their related orders are 3, 4, 1, 5, 2.

The rule generation procedure of the algorithm relies on the definition of the constrained subtree. Let $k_i \prec \cdots \prec k_2 \prec k_1$ be a set of the items' orders and P is a subpath from the root to the node N in the OOA_Fast_FP-tree, we say path P is constrained by the itemset $\{k_i, \ldots, k_2, k_1\}$ if the following two conditions are satisfied: (1) There exists a node M, which is a child node of N, and the orders of $k_i, \ldots, k_2, k_1$ appear in the subpath from node N to M; (2) k_i appears in the children of the node N, and k_1 appears in the

Table 2 Sub-dataset from Table 1

ID	Called API	File type
2	API_1, API_3	Malicious
4	$API_1, API_2, API_3, API_4, API_5$	Malicious
5	API_3, API_5	Malicious
7	API_2, API_4, API_5	Malicious

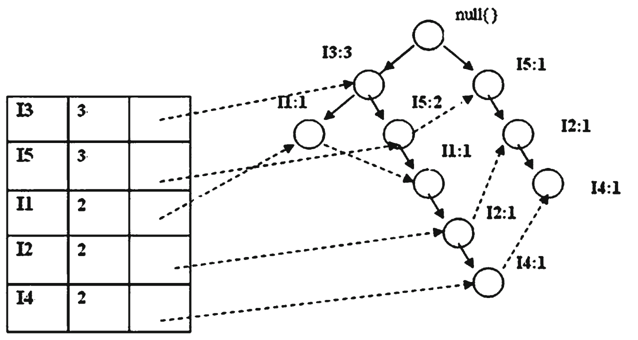


Fig. 6 FP-tree constructed from Table 2

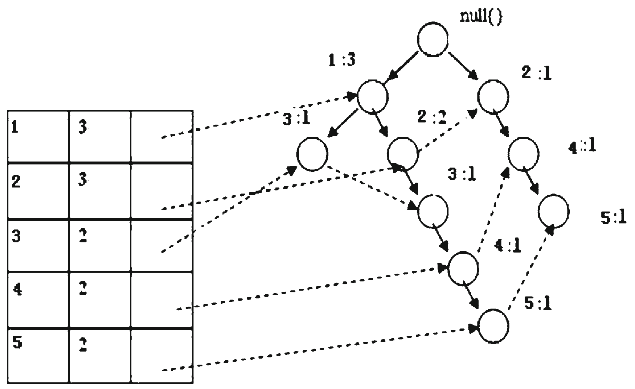


Fig. 7 Fast FP-tree constructed from Table 2

node M. If a subtree consists of all the subpaths which are constrained by $\{k_i, \ldots, k_2, k_1\}$, we call it constrained subtree by $\{k_i, \ldots, k_2, k_1\}$, and denote it as $ST(k_1, k_2, \ldots, k_i)$.

The constrained subtree $ST(k_1, k_2, \ldots, k_i)$ is a special subtree of the OOA_Fast_FP-tree, so we need to record the sum of the frequency counts of the nodes with the same order. Then, the constrained subtree $ST(k_1, k_2, \ldots, k_i)$ can be obtained by searching the OOA_Fast_FP-tree in a bottom-up fashion. The detailed description can be referred in [6].

5.5 An illustrating example

At the beginning of this section, we state that frequent patterns are essential to accurate classification. To demonstrate the effectiveness of the frequent patterns, we show an example rule generated by OOA_Fast_FP-Growth algorithm. We sample 5,611 records from our signature database, of which 3,394 records are malicious executables and 2,217 records are benign executables. One of the rules we generated is:
$(2230, 398, 145, 138, 115, 77) \rightarrow Obj_1 = (Group = Malicious)(os = 0.296739, oc = 0.993437)$,
where *os* and *oc* represent the support and confidence, representatively. After converting the API IDs to API names via our API query database, this rule becomes:
$(KERNEL32.DLL, OpenProcess; CopyFileA; CloseHandle; GetVersionExA; GetModuleFileNameA; WriteFile;) \rightarrow Obj_1 = (Group = Malicious)\ (os = 0.296739, oc = 0.993437)$.
After analyzing the API calls in this rule, we know that the program actually executes the following functions:

- returns a handle to an existing process object;
- copies an existing file to a new file;
- closes the handle of the open object;
- obtains extended information about the version of the currently running operating system;
- retrieves the complete path of the file that contains the specified module of current process;
- writes data to the file.

with the *os* and *oc* values, we know that this API calls appears in 1,665 malware, while only in 11 benign files. Obviously, it is one of the essential rules for determining whether an executable is malicious or not. In the experiments section, we perform a comprehensive experimental study to evaluate the efficiency of different OOA mining algorithms.

5.6 Associative classifier

For OOA rule generation, we use the OOA_Fast_FP-Growth algorithm to obtain all the association rules with certain support and confidence thresholds, and two objectives: $Obj_1 = (Group = Malicious)$, $Obj_2 = (Group = Benign)$. The number of the rules is also correlated to the number of the samples. We sample 5,611 records from our signature database, of which 3,394 records are malicious executables and 2,217 records are benign executables. Thus we generate 50 rules with $mos = 0.38$ and $moc = 0.90$. Then we apply the technique of classification based on association rules (CBA) to build a CBA classifier [19] as our malware detection module. The CBA classifier is built on rules with high support and confidence and uses the association between frequent patterns and the type of files for prediction. So, our malware detection module takes the input of generated OOA rules and outputs the prediction of whether an executable is malicious or not.

6 Other data mining techniques for malware detection

Several data mining techniques such as Naive Bayes, SVM and Decision Tree have been applied to malware detection. In our work, we have performed a comprehensive experimental study comparing our IMDS with those methods. Here, we briefly describe the methods used in our experimental comparisons.

6.1 Classification methods

Naive Bayes. Naive Bayes is one of the most successful learning algorithms for text categorization which is based on the Bayes rule assuming conditional independence between classes. Based on the rule, using the joint probabilities of sample observations and classes, the algorithm attempts to estimate the conditional probabilities of classes given an observation.

Support Vector machines (SVMs). SVMs [29] have exhibited superb performance in binary classification tasks. SVM aims at searching for a hyperplane that separates the two classes of data with largest margin (the margin is the distance between the hyperplane and the point closest to it).

Decision Tree. Decision Tree builds a binary classification tree. Each node corresponds to a binary predicate on one attribute. One branch corresponds to the positive instances of the predicate and the other to the negative instances. Thus, each node corresponds to a sequence of predicates and their values appearing on the downward path from the root to it. Each leaf is labeled by a class. To predict the class label of an input, a path to a leaf from the root is found depending on the value of the predicate at each node visited. The predicates are chosen from top to bottom by calculating the information gain of each attribute, which is the expected reduction in entropy caused by partitioning of the samples according to the attribute.

6.2 Feature selection

Feature selection is important for many pattern classification systems [13,16,17]. Identifying the most representative features is critical to minimize the classification error. As not all of the API calls are contributing to malware detection [26,32], in the experiments, we applied the Max-Relevance [22] feature selection algorithm to improve the efficiency and accuracy of the classifiers.

The main idea of Max-Relevance algorithm is to select a set of API calls with the highest relevance to the target class, i.e., the file type. Given a_i which represents the API with ID i, and the file type f ("0" represents benign executables and "1" is for malicious executables), their mutual information is defined in terms of their frequencies of appearances $p(a_i)$, $p(f)$, and $p(a_i, f)$ as follows:

$$I(a_i, f) = \int\int p(a_i, f) log \frac{p(a_i, f)}{p(a_i)p(f)} d(a_i)d(f).$$

With this algorithm, we select the top m APIs in the descent order of $I(a_i, f)$, i.e., the best m individual features correlated to the file types of the PE files.

In the experiments, we use Max-Relevance on the classifiers of Naive Bayes, SVM and Decision Tree, and then compare the results of these classifiers with our IMDS system.

7 Experimental results and analysis

We conduct three sets of experiments using our collected data. In the first set of experiments, we evaluate the efficiency of different OOA mining algorithms. The second set of experiments is to compare the abilities to detect polymorphic/metamorphic and unknown malware of our IMDS system with current widely used anti-virus software. The efficiency and false positives by using different scanners have also been examined. Finally, we compare our IMDS system with other classification based methods. All the experiments are conducted under the environment of Windows 2000 operating system plus Intel P4 1 GHz CPU and 1 GB of RAM.

7.1 Evaluation of different OOA mining algorithms

In the first set of experiments, we implement OOA_Apriori, OOA_FP-Growth, and OOA_Fast_FP-Growth algorithms under Microsoft Visual C++ environment. We sample 5,611 records from our signature database, which includes 3,394 records of malicious executables and 2,217 records of benign executables. By using different support and confidence thresholds, we compare the efficiency of the three algorithms. The results are shown in Table 3 and Fig. 8. From Table 3, we observe that the time complexity increases exponentially as the minimum support threshold decreases. However, it shows obviously that the OOA_Fast_FP-Growth algorithm is much more efficient than the other two algorithms, and it even doubles the speed of performing OOA_FP-Growth algorithm. Figure 8 presents a clearer graphical view of the results.

Table 3 Running time of different OOA mining algorithms (min)

Experiment	1	2	3	4
Mos	0.355	0.35	0.34	0.294
Moc	0.9	0.95	0.9	0.98
OOA_Apriori	25	260.5	∞	∞
OOA_FP-Growth	8	16.14	60.5	280.2
OOA_Fast_FP-Growth	4.1	7.99	28.8	143.5

Mos and Moc represent the minimum support and minimum confidence for each experiment

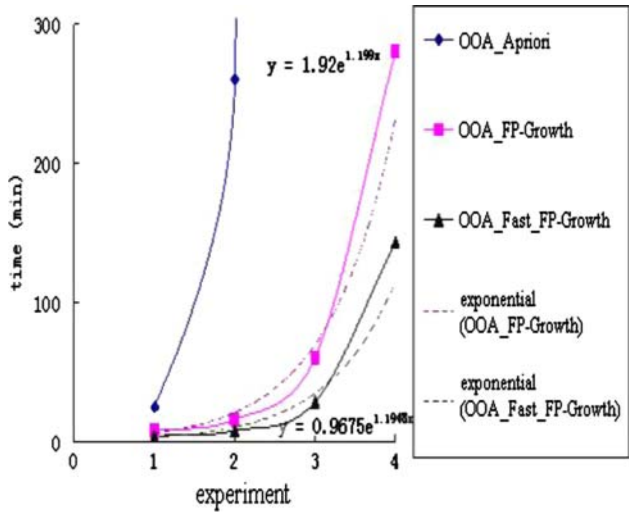


Fig. 8 Comparison on the efficiency of different OOA mining algorithms

7.2 Comparisons of different anti-virus scanners

In this section, we examine the abilities of detecting polymorphic malware and unknown malware of our system in comparison with some of the popular software tools such as Norton AntiVirus 2007, Dr. Web 2007, McAfee VirusScan 2007 and Kaspersky Anti-Virus 2007. We use all of their newest versions of the base of signature on the same day (20 September, 2007) for testing. The efficiency and the number of false positives are also evaluated.

7.2.1 Polymorphic/metamorphic virus detection

In this experiment, we sample 3,000 malicious files and 2,000 benign files in the training data set, then we use 1,500 malware and 500 benign executables as the test data set. Several recent Win32 PE viruses are included in the test data set for analysis such as Lovedoor, My doom, Blaster, and Beagle. For each virus, we apply the encryption and obfuscation techniques described in [26], including flow modification, data segment modification and insertion of dead code, to create a set of polymorphic/metamorphic versions. Then we compare our system with current most widely used anti-virus software. The results shown in Table 4 demonstrate that our IMDS system achieves better accuracy than other software in polymorphic malware detection.

7.2.2 Unknown malware detection

In order to examine the ability of identifying new and previously unknown malware of our IMDS system, we use 1,000 malware for test. These malware are not simple modifications of well known malware and we do not have any information about their nature. They are analyzed by the experts in KingSoft Anti-virus laboratory and their signatures have not been recorded into the virus signature database. Comparing with other anti-virus software, our IMDS system performs most accurate detection. The results are listed in Table 5.

Table 4 Polymorphic/metamorphic malware detection

Software	N	M	D	K	IMDS
Beagle	√	√	√	√	√
Beagle V1	√	√	×	√	√
Beagle V2	√	×	×	√	√
Beagle V3	×	×	×	√	√
Beagle V4	√	√	×	×	√
Blaster	√	√	√	√	√
Blaster V1	√	√	√	√	√
Blaster V2	×	×	×	×	√
Lovedoor	√	√	√	√	√
Lovedoor V1	×	×	√	×	√
Lovedoor V2	×	×	×	×	√
Lovedoor V3	×	√	×	√	√
Mydoom	√	√	√	√	√
Mydoom V1	×	×	×	×	√
Mydoom V1	×	×	×	?	√

N Norton AntiVirus, *M* MacAfee, *D* Dr. Web, *K* Kaspersky, "√" successful detection, "×" failure to detect, "?" only an "alert"; all the scanners used are of most current and updated version

Table 5 Unknown malware detection

Software	N	M	D	K	IMDS
malware 1	√	√	√	×	√
malware 2	×	×	√	√	√
malware 3	√	×	×	×	√
malware 4	×	×	×	×	×
malware 5	×	√	×	×	√
malware 6	×	×	√	×	√
malware 7	√	×	×	×	√
malware 8	×	×	×	×	√
malware 9	×	√	√	×	√
malware 10	×	×	×	×	√
malware 11	×	×	×	√	√
malware 12	×	√	×	√	√
⋮	⋮	⋮	⋮	⋮	⋮
malware 1000	√	×	×	×	√
Stat.	633	753	620	780	920
Ratio. (%)	63.3	75.3	62	78	92

7.2.3 System efficiency and false positives

In malware detection, a false positive occurs when the scanner marks a benign file as a malicious one by error. False positives can be costly nuisances due to the waste of time and resources to deal with those falsely reported files. In this set of experiments, in order to examine the system efficiency and the number of false positives of the IMDS system, we sample 2,000 executables in the test data set, which contains 500 malicious executables and 1,500 benign ones.

First, we compare the efficiency of our system with different scanners including the scanner named "SAVE" [26,32] described in related work and some widely used anti-virus software.

The results in Fig. 9 illustrate that our IMDS system achieves much higher efficiency than other scanners when being executed in the same environment.

The number of false positives by using different scanners are also examined. By scanning 1,500 benign executables whose signatures have not been recorded in the signature database, we obtain the results shown in Fig. 10. Figure 10

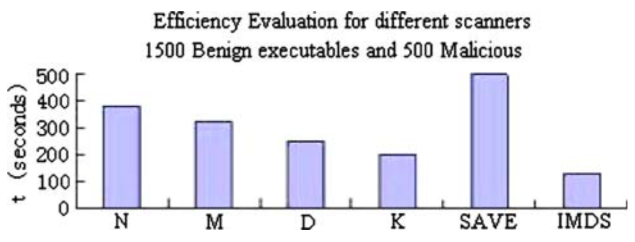


Fig. 9 Efficiency of different scanners

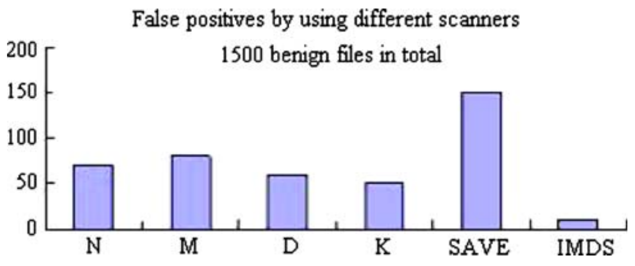


Fig. 10 False positives by using different scanners

clearly shows that the false positives by using our IMDS system are much fewer than other scanners.

7.3 Comparisons of different classification methods

In this set of experiments, we compare our system with other classification methods described in Sect. 6. We randomly select 2,843 executables from our data collection, in which 1,207 files are benign and 1,636 executables are malicious. Then we convert the transactional sample data in our signature database into a relational table, in which each column corresponds to an API and each row is an executable. This transformation makes it easy to apply feature selection methods and other classification approaches.

First, we rank each API using Max-Relevance algorithm introduced in Sect. 6.2, and then choose top 500 API calls as the features for later classification. In the experiments, we use the Naive Bayes classifier and J4.8 version of Decision Tree implemented in WEKA [31], and also the SVM implemented in LIBSVM package [12]. For the OOA_Fast_FP-Growth mining, we select thresholds based on two criteria: setting *moc* as close to 1 as possible; and selecting a big *mos* without exceeding the maximum support in the data set. Then, in the experiment, we set *mos* to 0.294 and *moc* to 0.98. Tenfold cross validation is used to evaluate the accuracy of each classifier. The following evaluation measures are used in the results:

- **True positive (TP)**: the number of executables correctly classified as malicious code.
- **True negative (TN)**: the number of executables correctly classified as benign executables.

Table 6 Results by using different classifiers

Algorithms	TP	TN	FP	FN	DR (%)	ACY (%)
Naive Bayes	1,340	1,044	163	296	81.91	83.86
SVM	1,585	989	218	51	96.88	90.54
J4.8	1,574	1,027	609	62	96.21	91.49
IMDS	1,590	1,056	151	46	97.19	93.07

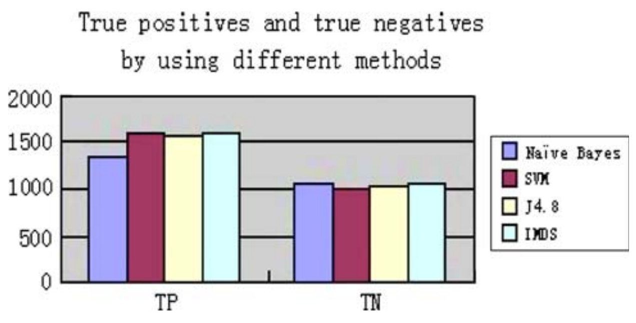


Fig. 11 True positives and true negatives of different classification methods

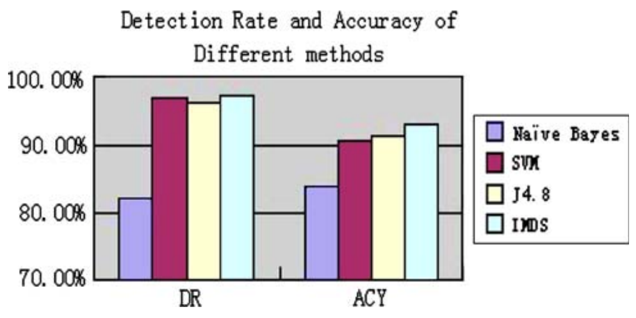


Fig. 12 Detection rate and accuracy of different classification methods

- **False positive (FP)**: as discussed in Sect. 7.2.3, it is the number of executables mistakenly classified as malicious executables.
- **False negative (FN)**: the number of executables mistakenly classified as benign executables.
- **Detection rate (DR)**: $\frac{TP}{TP+FN}$.
- **Accuracy (ACY)**: $\frac{TP+TN}{TP+TN+FP+FN}$

Results shown in Table 6 indicate our IMDS system achieve most accurate malware detection.

Figures 11 and 12 gives a graphic illustration of the effectiveness of different approaches. From the comparison, we observe that our IMDS outperforms other classification methods in both detection rate and accuracy. This is because "frequent patterns are of statistical significance and a classifier with frequent pattern analysis is generally effective to test datasets" [4].

8 Conclusions and future work

In this paper, we describe our research effort on malware detection based on window API calls. We develop an integrated IMDS system consisting of PE parser, OOA rule generator and rule based classifier. First, a PE parser is developed to extract the Windows API execution calls for each Windows portable executable. Then, the OOA_Fast_FP-Growth algorithm is adapted to generate association rules with the objectives of finding malicious and benign executables. This algorithm achieves much higher efficiency than previous OOA mining algorithms. Finally, classification is performed based on the generated rules. Experiments on a large real data collection from anti-virus lab at Kingsoft Corp. demonstrate the strong abilities of our IMDS system to detect polymorphic and new viruses. And the efficiency, accuracy and the scalability of our system outperform other current widely used anti-virus software and other data mining based malware detection methods. Our work results in a real malware detection system, which has been incorporated into the scanning tool of KingSoft's AntiVirus software.

In our future work, we plan to extend our IMDS system from the following avenues: (1) Collect more detailed information about the API calls such as their dependencies and timestamps and use it for better malware detection. We will investigate methods such as frequent structure mining to capture the complex relationships among the API calls. (2) Predict the types of malware. Our IMDS currently only provides binary predictions, i.e., whether a PE file is malicious or not. A natural extension is to predict the different types of malware. (3) Improve the efficiency and effectiveness of our frequent pattern based classification.

References

1. Adleman, L.: An abstract theory of computer viruses (invited talk). In: CRYPTO '88: Proceedings on Advances in Cryptology, pp. 354–374, New York, NY, USA. Springer, New York (1990)
2. Agrawal, R., Imielinski, T.: Mining association rules between sets of items in large databases. In: Proceedings of SIGMOD (1993)
3. Agrawal, R., Srikant, R.: Fast algorithms for association rule mining. In: Proceedings of VLDB-94 (1994)
4. Cheng, H., Yan, X., Han, J., Hsu, C.: Discriminative frequenct pattern analysis for effective classification. In: Proceedings of IEEE 23rd International Conference on Data Engineering (ICDE-07) (2007)
5. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium (2003)
6. Fan, M., Li, C.: Mining frequent patterns in an fp-tree without conditional fp-tree generation. J. Comput. Res. Dev. **40**, 1216–1222 (2003)
7. Filiol, E.: Computer Viruses: from Theory to Applications. Springer, Heidelberg (2005)
8. Filiol, E.: Malware pattern scanning schemes secure against black-box analysis. J. Comput. Virol. **2**(1), 35–50 (2006)
9. Filiol, E., Jacob, G., Liard, M.L.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. J. Comput. Virol. **3**(1), 27–37 (2007)
10. Han, J., Kamber, M.: Data Mining: Concepts and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2006)
11. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of SIGMOD, pp. 1–12, May (2000)
12. Hsu, C., Lin, C.: A comparison of methods for multiclass support vector machines. IEEE Trans. Neural Netw. **13**, 415–425 (2002)
13. Jain, A., Duin, R., Mao, J.: Statistical pattern recognition: A review. IEEE Trans. Pattern Anal. Mach. Intell. **22**, 4–37 (2000)
14. Kephart, J., Arnold, W.: Automatic extraction of computer virus signatures. In: Proceedings of 4th Virus Bulletin International Conference, pp. 178–184 (1994)
15. Kolter, J., Maloof, M.: Learning to detect malicious executables in the wild. In: Proceedings of KDD'04 (2004)
16. Kwak, N., Choi, C.: Input feature selection by mutual information based on parzen window. IEEE Trans. Pattern Anal. Mach. Intell. **24**, 1667–1671 (2002)
17. Langley, P.: Selection of relevant features in machine learning. In: Proceedings of AAAI Fall Symposium (1994)
18. Lee, T., Mody, J.: Behavioral classification. In: Proceedings of 2006 EICAR Conference (2006)
19. Liu, B., Hsu, W., Ma, Y.: Integreting classification and association rule mining. In: Proceedings of KDD'98 (1998)
20. Lo, R., Levitt, K., Olsson, R.: Mcf: A malicious code filter. Comput. Secur. **14**, 541–566 (1995)
21. McGraw, G., Morrisett, G.: Attacking malicious code: report to the infosec research council. IEEE Softw. **17**(5), 33–41 (2002)
22. Peng, H., Long, F., Ding, C.: Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. IEEE Trans. Pattern Anal. Mach. Intell. 27 (2005)
23. Rabek, J., Khazan, R., Lewandowski, S., Cunningham, R.: Detection of injected, dynamically generated, and obfuscated malicious code. In: Proceedings of the 2003 ACM Workshop on Rapid Malcode, pp. 76–82 (2003)
24. Schultz, M., Eskin, E., Zadok, E.: Data mining methods for detection of new malicious executables. In: Security and Privacy, 2001 Proceedings. 2001 IEEE Symposium on 14–16 May, pp. 38–49 (2001)
25. Shen, Y., Yang, Q., Zhang, Z.: Objective-oriented utility-based association mining. In: Proceedings of IEEE International Conference on Data Mining (2002)
26. Sung, A., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: Proceedings of the 20th Annual Computer Security Applications Conference (2004)
27. Swets, J., Pickett, R.: Evaluation of Diagnostic System: Methods from Signal Detection Theory. Academic Press, New York (1982)
28. Tan, P., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison Wesley, Reading (2005)
29. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, Heidelberg (1999)
30. Wang, J., Deng, P., Fan, Y., Jaw, L., Liu, Y.: Virus detection using data mining techniques. In: Proceedings of IEEE International Conference on Data Mining (2003)
31. Witten, H., Frank, E.: Data Mining: Practical Machine Learning Tools with Java Implementations. Morgan Kaufmann, San Francisco (2005)
32. Xu, J., Sung, A., Chavez, P., Mukkamala, S.: Polymorphic malicous executable sanner by api sequence analysis. In: Proceedings of the International Conference on Hybrid Intelligent Systems (2004)
33. Ye, Y., Wang, D., Li, T., Ye, D.: IMDS: Intelligent malware detection system. In: Proccedings of ACM International Conference on Knowlege Discovery and Data Mining (SIGKDD 2007) (2007)

34. Yin, X., Han, J.: Cpar: Classification based on predictive association rules. In: Proceedings of 3rd SIAM International Conference on Data Mining (SDM'03), May (2003)
35. Zuo, Z., Tian Zhou, M.: Some further theoretical results about computer viruses. Comput. J. **47**(6), 627–633 (2004)
36. Zuo, Z., Zhu, Q.-x., Zhou, M.-t.: On the time complexity of computer viruses. IEEE Trans. Inf. Theory **51**(8), 2962–2966 (2005)