**Detecting Malware with Machine Learning Based on Combination of Features**

Yubin Yang1,Wei Jiang2, Deyuan Li3

1School of Computer Science & Engineering, South China University of Technology, 510641 Guangzhou, China

2Tandon School of Engineering, New York University, 11201, Brooklyn, NY, USA

3Bluedon Information Security Technologies Co., Ltd , 510641 Guangzhou, China

[ronald\_yang@126.com](mailto:ronald_yang@126.com), [wei.jiang@nyu.edu](mailto:wei.jiang@nyu.edu), [deyuan123321@gmail.com](mailto:yxu@scut.edu.cn)

**Abstract:** Malware has become one of the most serious threats to computer information systems today, but current malware detection approaches still have significant limitations. With the explosive growth of new malware, a highly efficient detection tool is urgently needed. We propose a lightweight static malware detection approach based on machine learning in this paper. According to our in-depth analysis of differences between malicious and benign files, we extracted a large number of suspicious characteristics from file headers and the called APIs in the file. We then used both manual and machine selection methods to select around 600 of them as the most effective features. We used the random forest method to build the predictive model with about 9000 files used for training and testing. The experimental results show that our model has an accuracy of 99.6% but a low 0.47% false positive rate, indicating that our model can distinguish malicious files from benign ones in an effective way. Finally, our model prediction time offers consistent fast performance, which makes it suitable for real-time use.

**Key words:** Computer security; Machine learning; Static malware analysis; Malware detection

1. **Introduction**

The rapid growth and influence of information technology has increasingly led to growing concerns about cyber security and privacy [11,18,39]. The open nature and lack of central control of the internet makes it difficult to protect networked computers fully, especially in the case of shared systems [4,19]. In such a networked environment, malware spreads more easily and has even greater destructive ability [13,38,41].

Malware [7] is “any code added, changed, or removed from a software system to intentionally cause harm or subvert the system’s intended function.” A majority of antivirus vendors deploy signature-based malware detection techniques that utilize a predefined set of signatures such as the Black and White list. However, these techniques suffer from several shortcomings.

* The size of the signature set and the time required to match signatures increases rapidly.
* The number of new signatures added are too low to cover the emerging malicious programs [35]. Spinellis [32] has proven that the detection of bounded-length viruses is an NP-complete problem.
* During the period between the appearance of hitherto unseen malware and the update of the signature set on client computers, millions of devices are vulnerable to the new malware.
* Malware writers often use sophisticated packing techniques to evade signature matching. The advent of more sophisticated techniques, such as polymorphism and metamorphism, make it even harder to detect malware that uses them.

Many non-signature-based malware detection techniques have been proposed recently. These techniques detect malware using several characteristics, including the abnormalities in file structure [3,23,36], byte sequence [1,37], disassembled code [25], static calls from disassembled code [10], and run-time API calls [6]. They have been already used in practical applications to detect spam [42] and juice filming charging attacks on smartphones [5,17,24,48].

The structure of a binary executable is affected by its behavior. Executables follow one of several common formats, such as PE (portable executables) and ELF (executable and linking format), consisting of headers and sections telling the operating system loader how to map the file into memory. Malware and infected executables also follow this format but differ in structure when compared with benign executables.

The prevalence of Microsoft Windows creates an attractive environment for malware creation. The chief executable format for Windows is PE, a Microsoft standard. Because of its dominance, we focus on malware detection in PE files, but our ideas can apply to other formats, such as APK and PDF, as well.

In this paper, we present a malware detection engine based on machine learning (ML) that uses static features of file , such as file header, to detect previously unseen malware efficiently. We extract high-dimensional features based on domain experts and provide insight into the underlying problem. We extract such features from headers of all major file sections. We then apply feature selection methods to reduce the dimensionality and size of the feature space. We then use selected features to train different classifiers, including random forest (RF) and support vector machine (SVM).

The highlights of our contribution include the following.

* Identification of a set of useful and comprehensive static features for detection
* A typical accuracy rate of 99%, with a false positive rate of less than 1%
* Use of a lightweight process that is usable in real-time applications

The rest of our paper is organized as follows. Section 2 discusses Related Work. Section 3 describes our use of novel and effective features. Section 4 presents our experimental results and analysis. Section 5 discusses the advantages and limitations of our technique. Section 6 presents our conclusions.

1. **Related Work**

This section presents prior work that is significant to the development of our proposal. Strong emphasis is placed on machine learning and malware detection.

2.1 General Machine Learning Framework

Machine learning is a vast and ever-changing field[30,43]. The creation of our predictive model requires gathering a large numbers of both malicious and benign files and then extracting features and labels (malicious or benign) for each file. We use these features to train the model, a process that finds patterns and clusters in the data. After training, the model can examine an unknown file and return two outputs: a confidence score indicating the file’s similarity to benign or malicious samples and a rule set detailing the most important feature set used to make that prediction. Figure 1 presents a simplified illustration of the training and prediction process.

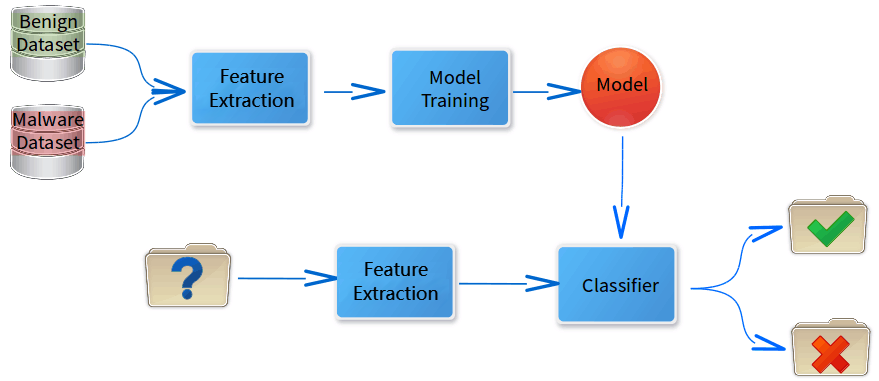


Figure 1: Machine learning process for malware detection

Although the machine learning technique is not 100% precise, detecting malware with an acceptable detecting rate is possible even for the unseen ones.

* 1. Malware Detection & Analysis

Traditionally, there are two main approaches to malware detection with machine learning: static analysis and dynamic analysis: naming the static and dynamic one.

**2.2.1 Static Analysis**

Static malware detectors have been used since at least 2001 [27]. Static analysis checks structural information, executable binaries, and assembly code without executing any of the codes.

Schultz et al. [27] presented a data mining framework to detect new, previously unseen malware and proposed three different methods to detect malicious executables on the Windows platform. The first technique was based on a list of DLLs (dynamic link libraries), the list of DLL API function calls, and the number of different function calls used by the binary feature within each DLL. The second technique treated the file content as some printable string sequences . The third technique used *N*-gram of length 2 (2 byte sequences of contiguous characters) to extract features from the raw byte sequence of the file. Their approaches achieved a detection rate of 97.76%.

Kolter and Maloof et al. [14,15] improved Schultz’s third technique by applying byte *n*-grams instead of non-overlapping sequences. *N-*grams of length 4 were used as features, and the top 500 *n*-grams produced from single bytes were selected through the Information Gain algorithm. Their experiment showed that boosted decision trees achieved a true-positive rate of 98%.

However, malware creators can add long sequences of meaningless bytes into the file to overcome binary feature detection. Therefore, many researchers have instead concentrated on the executable machine instructions known as opcodes (operational codes). They extract opcode sequences from the file after disassembly and use *N*-gram techniques to extract representative features from the executable. The experimental results indicate that the use of opcode *n*-gram analysis achieves better result than the byte *n*-gram analysis. Rezaei et al. [25] proposed a malware detection model using opcode statistical analysis based on the premise that malware was uninfluenced by obfuscation and disassembly methods. They achieved a detection rate of 90%.

Other researchers have tried to detect malware through recognizing behavior hidden in the malware. Typically, malware requires access to operating system APIs to carry out malicious intent. Elhadi et al. [9] transformed input samples into API call graphs then used a longest common subsequence (LCS) algorithm to detect malware. Experimental results on 85 samples demonstrated a 98% detection rate and a 0% false positive rate.

Shafiq et al. [23] proposed using just seven features from the PE header, including the debug data size, the image version, the resource data size, and the virtual size of the second section, based on the fact that malware samples typically exhibit differently in those elements. Though the author reported good experiment results, the method had limited application as these superficial features are easily modified without any change to the malicious code.

Most of above algorithms use hand-crafted features obtained from parsing the file. This requires domain knowledge and significant effort as a genuine adversary wrote the file. Recent advances in deep learning have dramatically improved the state of the art, especially in image and natural language classification, and could simplify the feature extraction process. It is also a good research direction for static malware detection [44,45]. However, it suffers from limited samples and high computational cost for most researchers.

**2.2.2 Dynamic Analysis**

Dynamic techniques aim to test and evaluate a program through actual execution of code inside a specially instrumented environment such as a virtual machine (VM), leaving it far less vulnerable to code obfuscation than the static technique. However, the method requires creation of a target environment for the malware to exhibit all malicious behavior.

Stopel et al. [33] employed ANNs (artificial neural networks) to detect the presence of computer worms based on the computer’s performance. The method examined sixty different indicators of the infected computers. The results indicated that the method had computational advantages and could detect previously unknown worms. Stopel et al. [34] further improved worm detection with ANNs using three temporal analysis preprocessing techniques and reducing the number of features.

Moskovitch et al. [20,22] applied data mining to detect an unknown computer worm’s activity based on behavioral features. Several computer configurations and background applications were used to test this new method. The results indicated that average accuracy was over 90% using just 20 features. The environment appeared to have no influence on the detection rate.

However, it is easy to confuse whether the execution environment is changed by malware or the OS itself. Therefore, newer research has examined patterns of API used by programs that might reveal malicious intent. Ahmed et al. [2] introduced a malware detection technique combining two different dynamic features (from spatial and temporal information) available in a run-time API. Lim [16] abstracted malicious behaviors as sets of *k*-grams of API call sequences to increase the efficiency and flexibility of the malware analysis.

External behaviors such as abnormal network access are also good indicators for malware detection. Zhang et al. [21] proposed discovering triggering relations on network requests and leveraging the communication protocol structural information to identify stealthy malware activities that cannot be attributed to a legitimate cause. The method successfully detected various malware types such as spyware.

**2.2.3 Static and Dynamic Analysis**

Although dynamic detection detects malicious behavior effectively, it is less successful when malware uses evasive methods. Therefore, other researchers have begun exploring a combination of static and dynamic analysis to improve detection performance. Shijo et al. [31] extracted features by analyzing both binary code and dynamic behavior. They achieved static accuracy of 95.8%, dynamic accuracy of 97.1%, and combined accuracy of 98.7%.

2.3 Our Improved Technique

The major advantage of the static approach over its dynamic counterpart is that it is free from the overhead of execution time. It requires no sandbox and has a higher speed of analysis and detection. Our work attempts to build a lightweight detection model with machine learning through static analysis of PE files, remaining effective even when the malware uses file packing and code obfuscation.

The portable executable format is straightforward to parse and encapsulates the information necessary for the Windows operating system loader to manage the wrapped executable code. Figure 2 [26] depicts the native Win32 executable format. Its specification is derived in part from the Unix common object file format (COFF).

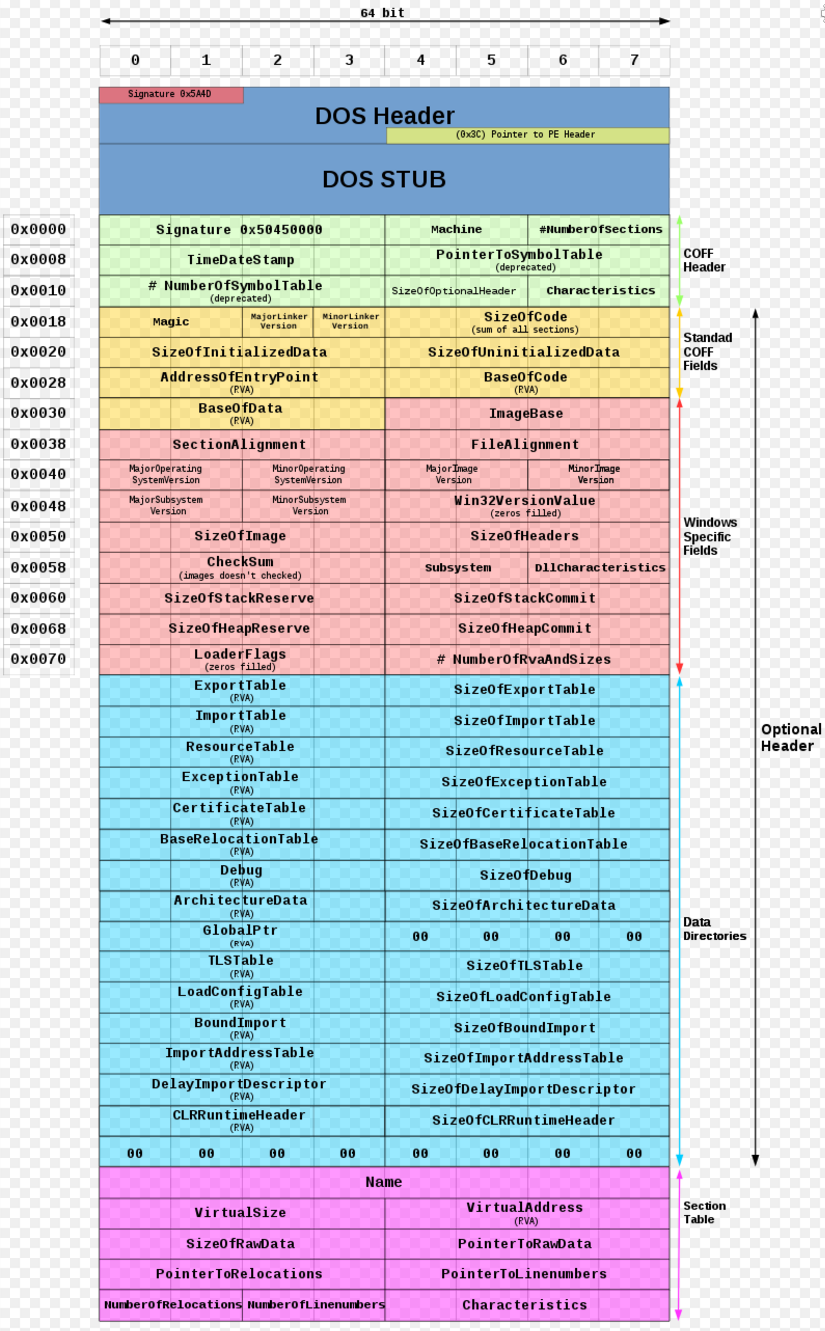


Figure 2: The PE File Format

A PE file consists of a file header and section table (section headers) followed by the sections’ data. The file header consists of an MS-DOS header, the PE signature, the image file header, and an optional header. The section headers immediately follow the file header.. Each section header provides information about its associated section, including location, length, and characteristics. The section is the basic unit of code or data within a PE or COFF file. Different functional areas, such as code and data, are separated logically into sections. In addition, an image file can contain a number of sections, such as *.tls* and *.reloc*, which have special purposes.

Many fields in a PE file have no mandatory constraints. There are a number of redundant fields and spaces in PE file, creating opportunities for propagating and hiding malware. Some anomalies in PE binaries include:

* code execution starting in the last section
* suspicious section characteristics
* suspicious code redirection
* suspicious code section name
* entry points not pointing into any of the sections
* import address table not patched
* multiple PE headers
* incorrect code size in the header

The idea of using file format features in malware detection is not new but a comprehensive study of their use with machine learning is lacking. Szor [36] summarized format abnormalities of the malware and infected files, then discussed a heuristic detection method based on format abnormalities. Weber et al. [40] used format features of executables that were likely to indicate the presence of inserted malicious code. Dai [8] reproduced Kolter and Maloof’s experiment, analyzed filtered features, and found that most of the features related to the PE format.. Those works used static format abnormalities as heuristic malware detection rules, but none of these works employed data mining to address the problems.

Researchers have also used data mining to detect unknown malware in recent years. Shafiq et al. [28] presented a PE-miner framework that examined PE file headers to identify unknown malware. Later, Shafiq et al. [29] used packer detection to improve this method. However, many automatic feature mining and selection algorithms [46,47] have been proposed with claims of effectiveness, but we find that the most robust features are extracted mostly manually based on a deep understanding of the learning problem and the significance of the features.

Therefore, compared with previous studies, our approach uses data mining skill and feature selection methods to extract effective and robust features from the file header with a deep understanding what constitutes abnormal or suspicious data. We then designed different experiments to evaluate our classifier. Although format analysis is not very advanced, it is an effective way to detect polymorphic malware.

Our technique applies a sequence of steps to detect malware:

Step 1: Parse the PE tables of all files to extract suspicious PE header characteristics, DLL names, and API functions inside each DLL as raw features.

Step 2: Apply information gain and other feature selection algorithms to reduce the size of the raw feature set from the PE headers and other sources.

Step 3: Transform each file to its corresponding feature vector and then use a learning algorithm to derive a classification prediction model offline.

Step 4: Use the model for online prediction.

1. **Feature Extraction and Selection**

There are numerous pieces of signatures that could be mapped into the feature vector space, necessitating careful selection. We first calculate mean values to show the differences between malware and benign files. Such basic data analysis steps are critical for establishing the foundation of the ML work (feature selection, feature space reduction, model training, etc.) and demonstrating the theoretical possibility of solving this problem.

Table 1: Mean values between benign and infected samples

|  |  |  |
| --- | --- | --- |
| Features | Benign Files | Malware Files |
| **No. of APIs** | **139.93** | **64.33** |
| CheckSum=0 | 0.04 | 0.59 |
| No. of Sections | 5.18 | 4.03 |
| **No. of Symbols** | **2.87** | **24016342.70** |
| SizeofCode | 238969.77 | 3874030.04 |
| SizeofImage | 522040.54 | 1114646.24 |
| SizeofStackReserve | 757006.20 | 1173450.41 |
| SizeofInitialData | 222293.25 | 741627.82 |
| SizeofUninitialData | 760.76 | 3201414.71 |
| ExportTableSize | 921.05 | 1208299.54 |
| Import Table Size | 164.01 | 1078800.52 |
| **Config Table size** | **31.46** | **2.57** |
| InternetOpenUrlA | 0.00 | 0.03 |
| LoadLibrary | 0.45 | 0.54 |
| DeleteFileA | 0.05 | 0.15 |
| \_initterm | 0.67 | 0.15 |
| mscoree.dll | 0.11 | 0.01 |

Table 1 presents the mean values of some selected structural features used in our study. As shown, benign files have some common characteristics: more API calls, a normal checksum, more detailed debug information, and fewer debugging symbols. Similarly, infected binaries have commonalities, particularly in their use of the *LoadLibrary* call to obscure malicious purposes and of *DeleteFileA*, *InternetOpenUrlA*, and the like to erase local files and retrieve network content. One interesting data point is seen in the references to *mscoree.dll.* This library is part of Microsoft’s .NET platform and is used by any application using it. Among benign files, 11.2% used it, but less than 1% of malware ones did. The reason is that *.NET* tends to increase the file size of dependent programs greatly. Few types of malware want that attention.

Thus, there are numerous characteristic differences between the types of files. Combined with malware analysis, we can extract useful features for ML. Our feature classes are DLL dependencies and PE structure.

**3.1 DLL Info**

**3.1.1 APIs**

Through the detailed analysis of the API calls, we divide relevant APIs APIs into 7 classes (see Table 2). Note that our data comes from static analysis and extraction of the import tables in the files.

Table 2: Lists of API Function Calls in Groups

|  |  |
| --- | --- |
| Behavior | Description |
| File-related behavior | * Create a specific file in sensitive folders * Delete, corrupt, cover system files or application files * Edit or encrypt a file * Traverse file directories or search for target files |
| Process-related behavior | * Release a DLL file to inject into system process and create a new thread to hide itself * Create new processes * Create threads to search for and terminate anti-virus or protection software processes * Create a matrix process to prevent repeated execution |
| Memory-related behavior | * Free, move, and replace blocks of memory; * Allocate additional memory space or decrease the total available memory size * Prevent memory allocation and reclaim memory space * Change the interrupt vector address to the initial address of the malicious code |
| Register-related behavior | * Add or delete a system service * Automatically run while the system is starting up * Hide itself * Undermine system functions |
| Network-related behavior | * Open or listen to a specific port * Transmit through chat software * Send email * Enumerate computers with weak passwords * Allocate resources without freeing them |
| Windows service behavior | * Terminate Windows Update service * Terminate windows firewall * Open telnet service |
| Other behaviors | Hook keyboard input; hide a window; alter system time to disable software prevention; restart the computer; scan existing vulnerabilities of the system |

We selected 136 common sensitive API functions from experience, covering a range of suspicious API behaviors. Space limitations prevent listing all of them, but the ones of greatest interest in the case of malware are as follows:

* ***DeleteFile***: Used to delete a file without user knowledge.
* ***CreateRemoteThread****:* Inject a custom DLL or executable code into other process’s address space.
* ***RegDeleteKey***: Delete registry entries used by the Windows system.
* ***GetSystemTime***: Determine the local time to be used with a time trigger.
* ***URLDownload****:* Download a file using a pre-defined URL.
* ***ReadProcessMemory***: Read a process’s memory space.
* ***ShellExecute***: Open or execute a file automatically.

**3.1.2 Alert API sequence**

From our statistical results about API use, both benign and malicious samples may use the same API functions. However, the combination of certain sensitive APIs is suspicious. For example, a virus might infect a system file, call *FindNextFile* to search for target format files, call *ReadFile* to read those files, and use *SetFileAttributes to* modify each file’s attributes. By looking at combined API sequences, we hope to explore the hidden relevance in sensitive APIs and reveal malicious activity. However, there are many APIs in the import table, so that the number of the combined API sequences (in groups of three) will be huge, with most combinations of no significance. Thus, we select an information gain algorithm to handle the problem. Finally, we select the 300 most significant API combinations, such as ‘*Sleep, \_initterm, GettickCount’*, ‘*GetCurrentAddress, \_initterm, UnhandleExceptionFilter’,* etc.

**3.1.3 API Counts on Top DLLs**

System DLLs in the Windows environment provide specific interfaces for accessing and manipulating system resources. For example, *kernel.dll* provides user access to the system kernel, IO, memory management , and the like. By extracting the number of API calls from key DLLs, we can get a preliminary idea of the degree of interaction between the sample code and the system. For example, a large number of API calls to functions in kernel.dll indicates a greater likelihood that the sample is a driver file.

We ranked the Windows system DLLs by the number of function calls made to each in our malware dataset and selected 40 of them as feature items based on the sensitivity and frequency of use. The final list includes:

***kernel32.dll, user32.dll, advapi32.dll, gdi32.dllwininet.dll, mshtml.dll***

***msvfw32.dll, ntoskrnl.exe, dbghelp.dll, ole32.dll, shlwapi.dll, winnm.dll***

***rpcrt4.dll, psapi.dll, hal.dll, netapi32.dll, avicap32.dll, imm32.dll***

***msvcrt.dll, comctl32.dll, shell32.dll, wsock32.dll, oleaut32.dll, msvbvm60.dll***

***ws2\_32.dll, ntdll.dll, urlmon.dll, version.dll, avifil32.dll, comdlg32.dll***

***rasapi32.dll, cygwin1.dll, mscoree.dll, imagehlp.dll, cryptui.dll, crypt32.dll***

***netshell.dll, wtsapi32.dll, setupapi.dll, dpnet.dll***

**3.2 PE Structure Format Information**

The structure of a malware PE file typically contains multiple indicators of suspicious behavior. The number of sections may be smaller than a normal file, the size of debug table might be smaller, there might be no debugging information, the entry point of the program might be in the last section, etc. In our process, we first extract all the characteristics of the PE header and then select the most important structural features based on Table 1.

Other features become relevant in comparison with each other, so our process attempts to find the deep features between them. The following points list some of the items we examine in each part.

1. Start section (the one containing the entry point): *SizeOfRawData,* the difference between the *virtualSize* and the *SizeOfRawData* values, whether the start section is the last section and writable etc.

2. Section cluster: executable section counts, executable and writable sections count, zero section counts, uncommon section name counts, with or without *.reloc* and *.tls* section etc.

3. Executable code: the difference between *SizeOfCode* and the sum of all executable section's *SizeOfRawData.*

Table 3 gives the detailed distribution of the features selected from the PE header.

Table 3: Raw Features extracted from PE Header

|  |  |  |
| --- | --- | --- |
| Module | Number | Description |
| DOS Header | 1 | e\_lfanew |
| PE file header | 6 | NumberOfSections, NumberOfSymbols |
| PE optional header | 45 | CheckSum, DllCharacteristics, SizeOfCode |
| PE data directories | 18 | Export Table Size, Load\_config table size |
| PE section header | 23 | Characteristics, entropy, section names |
| PE import description | 5 | Import function by name, Total numbers of Function |
| PE export description | 1 | Total numbers of Function |
| PE debug info | 1 | The No. of debug info |
| PE resource table | 13 | No. of Message Table, No. of Group Icon |

After combining the results from the DLL and PE structure analyses, we extract around 600 features for model training. Although static analysis has the reputation for weak performance when identifying malicious behaviors, our static process is able to detect suspicious programs based on their DLLs source, file structures source, etc. Our comprehensive experiments below show the feature effectiveness.

1. **Experiment**

4.1 Dataset Description

We collected 5060 recent malicious PE files in a total of 219 classes, with no duplicate files. Table 4 shows the diversity of our dataset. We did not exclude any known to have polymorphic or metamorphic malware. We then obtained about 3500 benign files from the Windows and Program Files folders.

Table 4: Diversity of Malware Dataset

|  |  |  |  |
| --- | --- | --- | --- |
| Malware Species | No. | Malware Species | No. |
| W32.Changeup | 441 | Trojan.Dropper | 45 |
| Trojan.Gen | 439 | W32.Changeup!gen35 | 44 |
| TrojanHorse | 283 | Trojan.Zbot | 42 |
| Infostealer.Gampass | 247 | Backdoor.Ripinip | 42 |
| Backdoor.Trojan | 192 | W32.Pilleuz!gen19 | 42 |
| Downloader | 189 | Downloader.Lofog!gen4 | 41 |
| Suspicious.Cloud.5 | 128 | Trojan.Koutodoor!gen | 36 |
| W32.Changeup!gen20 | 106 | Trojan.Pramro | 34 |
| WS.Reputation.1 | 87 | Bloodhound.Exploit.343 | 33 |
| W32.SillyFDC | 87 | Trojan.ADH.2 | 26 |
| W32.Changeup!gen44 | 80 | Adware.Rugo | 26 |
| Infostealer | 65 | W32.IRCBot.NG | 25 |
| W32.SillyDC | 60 | Packed.Generic.438 | 25 |
| Trojan.Gen.2 | 59 | W32.Waledac.C!gen2 | 23 |
| Packed.Protexor!gen1 | 58 | Packed.Generic.437 | 23 |
| Trojan.Fakeavlock | 54 | W32.Spyrat | 21 |
| W32.Sality!dr | 53 | Trojan.Startpage | 20 |
| Trojan.Ransomlock!g55 | 47 | Trojan.Usuge!gen3 | 19 |
| Adware.Clkpotato!gen3 | 47 | Packed.Generic.347 | 18 |
| W32.Changeup!gen15 | 47 | Trojan.Bamital!gen2 | 17 |

We used Symantec’s names for the malicious programs. The major types were trojan horses, worms, and spyware. We believe this sample dataset is quite comparable to the distribution of malware in the wild.

4.2Model Selection and Parameter Tuning

**4.2.1 Random Forest**

We experimented with many different models for the proof of concept. Eventually, we chose the random forest model. We selected RF for the following reasons:

* Simplicity: The model consists only of some decision trees with a simple voting mechanism.
* Interpretability: Following the rules and each fork (tree node), we can easily determine why a file has been predicted as malicious or benign.
* Predictability: Unlike other ML models, RFs naturally overcome the problem of overfitting naturally. This gives our model a high degree of generalization in production, which is critical for model predictability.

When classifying a new object, the feature vector will be distributed to each of the trees in the forest. Each tree will then result in a classification; these are the trees’ “votes” for that class. The forest than chooses the classification having the most votes.

**4.2.2 Parameter Tuning**

The best combination of parameters for the model is needed for production. We performed a grid search for the random forest and then focused on the following settings:

* numbers of estimators: the number of decision trees
* max\_features: the number of features to consider when looking for the best split
* min\_sample\_split: the minimum number of samples required to split an internal node
* oob\_score: whether to use out-of-bag samples for the random forest

We tuned the learning model using a grid search to try all permutations of certain parameters. In our case, we simply gave a range for each of the above parameters for building a random forest and ran a grid search. Each iteration of the grid search trained a random forest using all of the input data from our sample corpus and then did a fast two-fold cross validation and recorded the accuracy. Then, we plotted the top five results to see how well the features performed at different values and to predict how one feature might further improve if we continued tuning it beyond what we initially set as the bounds of the grid search.

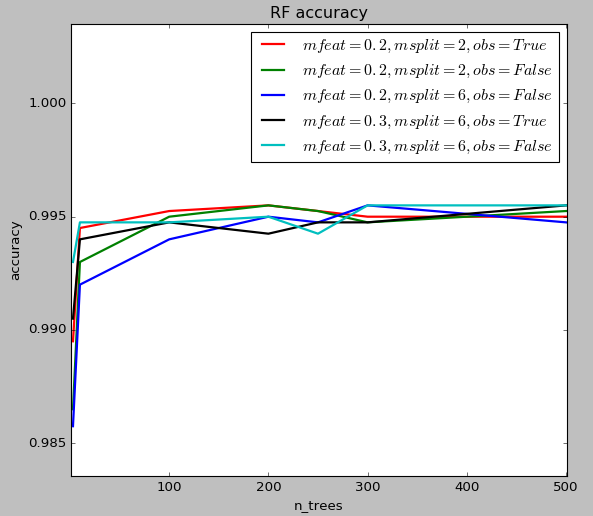


Figure 3: Grid Search for Model Parameters

As seen in Figure 3, the y-axis is model accuracy and the x-axis is the number of estimators. The optimal set of parameters we finally used was the following:

* number of estimators = 300
* max\_features = 0.3
* min\_sample\_split = 6
* oob\_score = False

In other words, , our model contained 300 trees, and randomly chose 30 percent of all features when building each tree, and the leaf nodes of a tree will not be split when the sample size is less than 6.

4.3 Feature Importance Analysis

The random forest model made it easy to export import features, shown in Figure 4. These results agree with our data in Table 1. The key to this learning problem is feature selection. The selected feature (whether performed by a human or a machine) must identify whether a program is malicious or not. At the core, the learning problem is the malware detection problem.

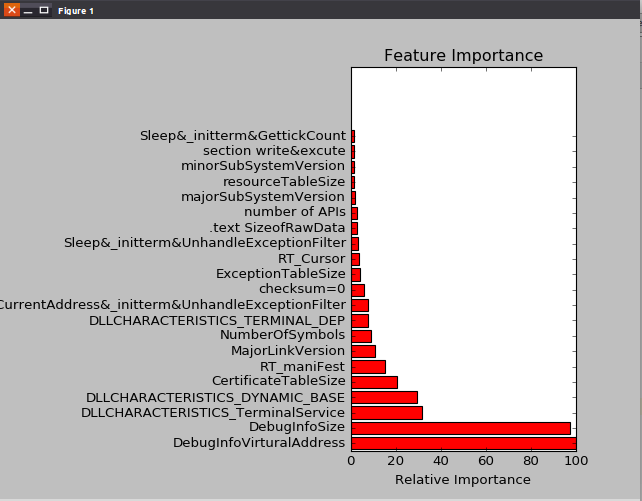


Figure 4: Important Analysis of Features

Table 5 presents our intuition as to why the above features are useful for malware detection.

Table 5: Descriptions of Some Important Features

|  |  |
| --- | --- |
| Extracted Features | Description |
| DebugInfoSize | Denotes the size of the debug-directory table. Usually, Microsoft-provided and other common executable files have a debug detail. in other words, many clean programs have a non-zero value for *DebugSize* in file header. |
| DLLCHARACTERISTICS | Denotes the file attributes in the DLL characteristics. Usually, benign file has a larger value for this than malware, which means the benign file has more embedded details about the file. |
| CertificateTableSize | Denotes the size of the certificate table. Usually, Microsoft-provided executable files have Microsoft certificate information, as well as other formal software company. in other words, many clean programs may have a non-zero value for *CertificateTableSize*. |
| RT\_maniFest | Denotes the size of the manifest resource lists the DLLs needed by the program. Clean programs usually need more DLLs to do their work. |
| MajorLinkVersion | Denotes the version of the program Linker. According to the statistical result, malware is more often to link to some versions of the program Linker. |
| NumberOfSymbols | Denotes the number of program symbols. Clean files usually have fewer symbols than malware. |
| GetCurrentAddress, \_initterm, UnhandleExceptionFilter | Denotes the combination of APIs. The frequency of some API calls differs between malicious and benign programs. |

4.4 Time Consumption

We have designed our model prediction process to be lightweight, making use of simple computations. We ran our experiment on a general-purpose computer (3GHz Core Duo with 8 GB RAM). The relationship between the prediction time overhead and file size is shown in Figure 5.

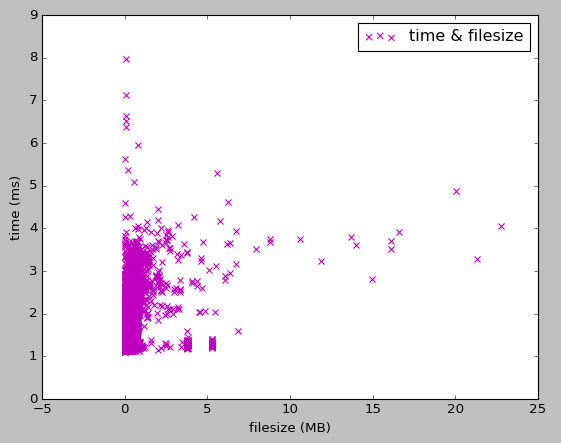


Figure 5: Time overhead versus the size of the file

Our prediction time is stable, being within 10 milliseconds for each new incoming file, and unaffected by file size We see a PE file requires more detection time than a non-PE file of the same size because PE files reference more APIs in the import table and resources in the resource table. Our training process required about 3 seconds to build the 300 decision trees for 5000 training samples. Our model fits nice in real-time situations.

4.5 Detection Rate and Analysis

To evaluate the predictive performance of our detection model, we randomly divided the entire dataset into two parts: training (90%) and testing (10%). We repeated the process 10 times for every combination to validate the results. This methodology helped to evaluate the robustness of a given approach to detection without a priori information. Table 6 and 7 present the standard measures of accuracy using the mean value of the 10 runs.

We generated the ROC curve by varying the threshold on output class probability. Figure 6 provides the plot.

Table 6: Confusion Matrix

|  |  |  |
| --- | --- | --- |
| Class | Labeled | |
| True | False |
| Positive (Malware) | 499 | 1 |
| Negative (benign) | 348.95 | 1.05 |

Table 7: Detection Rate

|  |  |  |
| --- | --- | --- |
| Detection Rate | False Positive Rate (FPR) | [Accuracy](file:///C:\Users\Administrator\AppData\Local\youdao\dict\Application\7.2.0.0703\resultui\dict\?keyword=accuracy) |
| 99.8% | 0.3% | 99.7% |

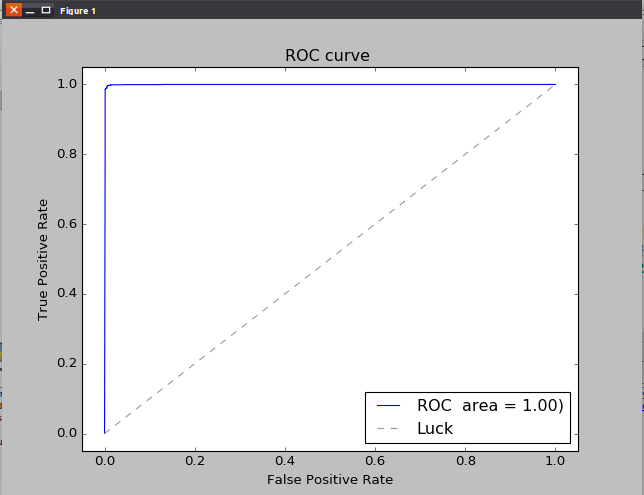


Figure 6: ROC curve

As shown in Tables 6 and 7, our detection rate exceeded 99% with a false positive rate under 1% the RF classifier. As shown in Figure 6, the area under the ROC Curve is approximately 1, indicating an effective features and robustness model. In other words, the static mining of features is effective in distinguishing benign and malicious files. We can efficiently recognize an unknown file as good or bad using static analysis to find suspicious information.

It is also interesting to compare the differences between various ML models (NB, SVM and RF) and feature set combinations (PE Header, API Sequence, DLLs and APIs, and all combined) to find the best classifier and feature combination. The entire dataset was divided into two parts: training (60%) and testing (40%).

Table 8: Comparison of Different ML Algorithms and Combination of Features

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | Classifier | Detection Rate | FPR | Accuracy |
| PE Header | NB | 2.64% | **0.00%** | 39.13% |
| SVM | 53.00% | **0.00%** | 70.62% |
| RF | 99.40% | 0.40% | 99.47% |
| API Sequence | NB | 0.12% | 4.47% | 35.88% |
| SVM | 98.16% | 24.08% | 89.82% |
| RF | 95.48% | 6.80% | 94.62% |
| DLL & API | NB | 50.54% | 13.80% | 63.92% |
| SVM | 70.71% | 2.13% | 80.90% |
| RF | 97.88% | 1.40% | 98.15% |
| All | NB | 2.76% | **0.00%** | 39.21% |
| SVM | 50.92% | **0.00%** | 69.32% |
| RF | **99.60%** | 0.47% | **99.57%** |

Table 8 shows that the best classifier is random forest with all of the selected features. It is unsurprising that the PE Header is the best single feature as shown in Figure 4. In our sample analysis, we found that some import tables in PE files were obviously smaller than others because the executable has less interaction with the operating system (benign) or is using a dynamic API loading method to avoid detection. This fact decreased our identification ability when using DLL and API features alone. The API sequence feature achieved the lowest accuracy rate because of sparsity in the feature space, but it improves detection results in two ways: overall accuracy and confidence level for a file.

Finally, we looked at the validity of our selected features. In other words, the robustness of our model if some important features were missing. We sorted the most important features through the RF, and observed the change in accuracy of the RF classifier when the important features were removed

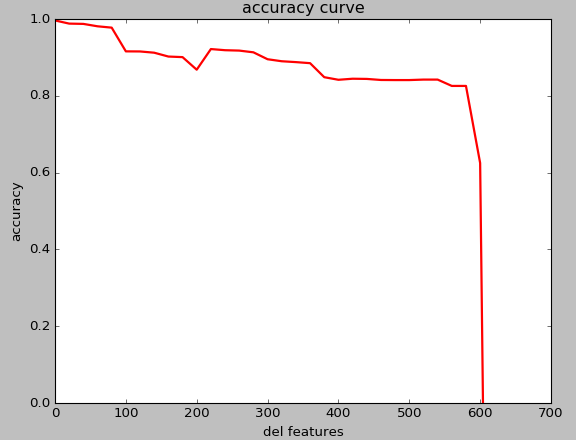


Figure 7: Accuracy versus deleted features

Figure 7 shows that, even with the deletion of 500 features, we still obtain an accuracy of 85%. There is a noticeable phenomenon that the accuracy went up when we deleted 200 important features. We think it was because of the random feature selection when we sorted the feature importance with the RF. This experiment has the practical significance that cyber security is a game between the offense and defense side. It is uncomfortable to see the hacker change just a few key characters in the PE header to in order to evade detection easily. From the results shown in Figure 7, we know that the rest of the features still have a good performance. Thus, we can remove some known features manipulated by the targeted malware writer, but the detection result of our model is still reliable. Considering that more features in our model to be justified for evading, the more work the adversary need to do to keep the file integrity , it would be a huge task to adjust all of our proposed features, hence, our model has a ability to fight against attack.

From our experimental results, we can conclude that our model successfully differentiates malicious and benign files effectively, with a detection rate for malware over 99%. It remains robust even when important characteristics are removed (or otherwise made to appear normal) by the malware creator, demonstrating that our model is suitable for on-site security analysis.

1. **Discussion**

Most types of malware use packing and obfuscation techniques to hide their malicious behavior and evade detection. However, malware that employs packing and obfuscation techniques still complies with the constraints of the format. Our proposed detection method extract features from the file structure and called APIs in file, in other words , our approach can still be valid even for file packing or code obfuscation.

In short, we identify the following major benefits of our technique.

* Can detect unknown malware effectively, while signature-based AV cannot
* Can detect packed malware
* Can provide a detailed explanation of why the file has been identified as malicious or benign
* Consumes no resources at run time because static analysis is done prior to execution
* Is computationally inexpensive

Last, it is uncomfortable to see a hacker easily evade detection if he justifies the PE header more formally that is similar to a clean program or hides alert APIs with dynamic function loading. This is a significant risk. Our model is unable to provide advice to users concerning how to prevent an attack because it has no idea of the malware behavior. In the future, we will evaluate the most significant and stable static features with a larger dataset and integrate static and dynamic analysis together.

1. **Conclusion**

The application of machine learning to malware detection is not a new concept, but it has only become realistic in production in the past few years. We have presented a static malware detection technique to mine structural information in files. With a deep understanding of suspicious info in files, our new detection technique makes highly accurate detection of unseen malware possible by using previous samples. At the same time, it maintains a low false-positive rate.

We have successfully deployed our malware detection engine into production. We first used the coarse-grained malware detector for fast filtering of malicious files and then conditionally applied a fine-grained malware family classification technique.

Our future research may include the following:

* Application of deep learning to static malware detection
* Application of the technique to a much larger dataset (10 million files)
* Implementation of an automatic pipeline for static and dynamic combination
* Research into APT detection
* Further classification of malware families
* Low latency and high-throughput detection engine optimization

**References**

[1] Ahmadi M, Ulyanov D, Semenov S, et al. Novel feature extraction, selection and fusion for effective malware family classification[C]//Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, 2016: 183-194.

[2] Ahmed F, Hameed H, Shafiq M Z, et al. Using spatio-temporal information in API calls with machine learning algorithms for malware detection[C]//Proceedings of the 2nd ACM workshop on Security and artificial intelligence. ACM, 2009: 55-62.

[3] Belaoued M, Mazouzi S. A Chi-Square-Based Decision for Real-Time Malware Detection Using PE-File Features[J]. Journal of Information Processing Systems, 2016, 12(4) :644-660.

[4] Cai J, Wang Y, Liu Y, et al. Enhancing network capacity by weakening community structure in scale-free network[J]. Future Generation Computer Systems, 2017: 1-7.

[5] Cai Z, Yan H, Li P, et al. Towards secure and flexible EHR sharing in mobile health cloud under static assumptions[J]. Cluster Computing, 2017, 20(3): 2415-2422.

[6] Choi S, Sun K, Eom H. Android malware detection using library api call tracing and semantic-preserving signal processing techniques[C]//Proceedings of the International Conference on Security and Management (SAM). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013: 1.

[7] Cohen F. Computer viruses: theory and experiments [J]. Computers & security, 1987, 6(1): 22-35.

[8] Dai J. Detecting malicious software by dynamic execution[M]. University of Central Florida, 2009.

[9] Elhadi A A E, Maarof M A, Barry B I A. Improving the detection of malware behaviour using simplified data dependent api call graph[J]. International Journal of Security and Its Applications, 2013, 7(5): 29-42.

[10] Faruki P, Laxmi V, Gaur M S, et al. Mining CFG as API Call-grams to Detect Portable Executable Malware[J].ACM-SIG,2012.

[11]Gupta B B, Agrawal D P, Yamaguchi S. Handbook of research on modern cryptographic solutions for computer and cyber security Google Scholar[J]. 2016.

[12] Stojmenovic I, Wen S, Huang X, et al. An overview of fog computing and its security issues[J]. Concurrency and Computation: Practice and Experience, 2016, 28(10): 2991-3005.

[13] Jiang J, Wen S, Yu S, et al. Identifying propagation sources in networks: State-of-the-art and comparative studies[J]. IEEE Communications Surveys & Tutorials, 2017, 19(1): 465-481.

[14] Kolter J Z, Maloof M A. Learning to detect and classify malicious executables in the wild[J]. Journal of Machine Learning Research, 2006, 7(Dec): 2721-2744.

[15] Kolter J Z, Maloof M A. Learning to detect malicious executables in the wild[C] //Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2004: 470-478.

[16]Lim H. Detecting Malicious Behaviors of Software through Analysis of API Sequence k-grams i[J].Computer Science and Information Technology,2016,4(3): 85-91.

[17] Meng W, Jiang L, Wang Y, et al. JFCGuard: Detecting juice filming charging attack via processor usage analysis on smartphones[J]. Computers & Security, 2017.

[18] Meng W, Tischhauser E, Wang Q, et al. When intrusion detection meets blockchain technology: a review[J]. IEEE Access, 2018(6) 10179-10188.

[19]Stergiou C, Psannis K E, Kim B G, et al. Secure integration of IoT and cloud computing[J]. Future Generation Computer Systems, 2018, 78: 964-975.

[20] Moskovitch R, Detection of unknown computer worms activity based on computer behavior using data mining,in Proceedings of the IEEE Symposium on Computational Intelligence in Security and Defense Applications, Honolulu, HI,2007:169-177.

[21]Zhang H, Yao D D, Ramakrishnan N, et al. Causality reasoning about network events for detecting stealthy malware activities[J]. computers & security, 2016, 58: 180-198.

[22] Nissim N, Moskovitch R, Rokach L, et al. Novel active learning methods for enhanced PC malware detection in windows OS[J]. Expert Systems with Applications, 2014, 41(13): 5843-5857.

[23] Shafiq M Z, Tabish S M, Mirza F, et al. A framework for efficient mining of structural information to detect zero-day malicious portable executables[R]. Technical Report, TR-nexGINRC-2009-21, January, 2009, available at http://www. nexginrc. org/papers/tr21-zubair. pdf, 2009.

[24] Peng S, Yang A, Cao L, et al. Social influence modeling using information theory in mobile social networks[J]. Information Sciences, 2017, 379: 146-159..

[25] Rezaei S, Afraz A, Rezaei F, et al. Malware detection using opcodes statistical features[C]//Telecommunications (IST), 2016 8th International Symposium on. IEEE, 2016: 151-155.

[26]PortableExecutable,[Online].Available:https://en.wikipedia.org/wiki/Portable\_Executable#/media/File:Portable\_Executable\_32\_bit\_Structure\_in\_SVG\_fixed.svg

[27] Schultz M G, Eskin E, Zadok F, et al. Data mining methods for detection of new malicious executables[C]//Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. IEEE, 2001: 38-49.

[28] Shafiq M Z, Tabish S M, Mirza F, et al. Pe-miner: Mining structural information to detect malicious executables in realtime[C]//International Workshop on Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg, 2009: 121-141.

[29] Shafiq M Z, Tabish S, Farooq M. PE-probe: leveraging packer detection and structural information to detect malicious portable executables[C]//Proceedings of the Virus Bulletin Conference (VB). 2009: 29-33.

[30] Shen Y, Zhang S, Wong H S, et al. Characterisation of semantic similarity on gene ontology based on a shortest path approach[J]. International journal of data mining and bioinformatics, 2014, 10(1): 33-48.

[31] Shijo P V, Salim A. Integrated static and dynamic analysis for malware detection[J]. Procedia Computer Science, 2015, 46: 804-811.

[32] Spinellis D. Reliable identification of bounded-length viruses is NP-complete[J]. IEEE Transactions on Information Theory, 2003, 49(1): 280-284.

[33] Stopel D, Boger Z, Moskovitch R, et al. Application of artificial neural networks techniques to computer worm detection[C]//Neural Networks, 2006. IJCNN'06. International Joint Conference on. IEEE, 2006: 2362-2369.

[34] Stopel D, Boger Z, Moskovitch R, et al. Improving worm detection with artificial neural networks through feature selection and temporal analysis techniques[J]. Int J Comput Sci Eng, 2006, 15: 202-208.

[35] Symantec Inc., “Symantec Internet Security Threat Report, Trends for 2011, Volume XVII,April 2012.

[36] Szor P. The art of computer virus research and defense[M]. Pearson Education, 2005.

[37] Wang K, Parekh J J, Stolfo S J. Anagram: A content anomaly detector resistant to mimicry attack[C]//International Workshop on Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg, 2006: 226-248.

[38] Wang Y, Wen S, Xiang Y, et al. Modeling the propagation of worms in networks: A survey[J]. IEEE Communications Surveys & Tutorials, 2014, 16(2): 942-960.

[39] Wang H, Zheng Z, Wu L, et al. New directly revocable attribute-based encryption scheme and its application in cloud storage environment[J]. Cluster Computing, 2017, 20(3): 2385-2392.

[40] Weber M, Schmid M, Schatz M, et al. A toolkit for detecting and analyzing malicious software[C]//Computer Security Applications Conference, 2002. Proceedings. 18th Annual. IEEE, 2002: 423-431.

[41] Wen S, Zhou W, Zhang J, et al. Modeling propagation dynamics of social network worms[J]. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(8): 1633-1643.

[42] Wu T, Wen S, Xiang Y, et al. Twitter spam detection: Survey of new approaches and comparative study[J]. Computers & Security, 2017.

[43] Zhang S, Wong H S, Shen Y, et al. A new unsupervised feature ranking method for gene expression data based on consensus affinity[J]. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2012, 9(4): 1257-1263.

[44]Elmisery A M, Sertovic M, Gupta B B. Cognitive Privacy Middleware for Deep Learning Mashup in Environmental IoT[J]. IEEE Access, 2017.

[45]Raff E, Barker J, Sylvester J, et al. Malware Detection by Eating a Whole EXE[J]. arXiv preprint arXiv:1710.09435, 2017.

[46] Chang X, Yang Y. Semisupervised feature analysis by mining correlations among multiple tasks[J]. IEEE transactions on neural networks and learning systems, 2017, 28(10): 2294-2305.

[47] Li Z, Nie F, Chang X, et al. Beyond Trace Ratio: Weighted Harmonic Mean of Trace Ratios for Multiclass Discriminant Analysis[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 29(10): 2100-2110.

[48]Yang Y, Wei Z, Xu Y, et al. Droidward: an effective dynamic analysis method for vetting android applications[J]. Cluster Computing, 2016: 1-11.