TALLINN UNIVERSITY OF TECHONLOGY

Faculty of Information Technology

Department of Computer Science

Chair of Network Software

ITI70LT

Kazuaki Tsuchimoto 121728

DETECTING MALWARE BY MACHINE LEARNING

Master's thesis

Supervisors:

Toomas Lepik

Master of Science in Engineering

Malware lecturer

TALLINN 2014

Declaration

I hereby certify that this master thesis is entirely the result of my work and I have faithfully and properly cited all sources in thesis.

…... …..

(Date) (Author's signature)

Abstract

The complexity of system also has increased and the clue of attacking simultaneously increased. Malware is rapidly spreading in the Internet and attacking the system which has sensitive data. Although anti-virus software detects and prevents malware from exploiting the target's system, there are some of malwares that may pass though the detection. Because the main force of anti-virus software to detect them is based on the signature method. Only with this signature based method, anti-virus software is thought as it is impossible to detect malware for sure. Today the heuristic method that detects malware by monitoring its own behavior is being developing and implemented on realistic product.

In this thesis, I studied about classification of goodwares and malwares by using machine learning. I focused on statical and dynamical analysis by extracting the characteristic quantities. I applied perceptron and AROW algorithm for calculating the similarity in the learning machine. Characteristics quantities that I extracted on our experiments were rank of appearance frequency for operation code, instruction code combined with N-gram, list of API stored in IAT and API call sequences obtained from Cuckoo Sandbox. In taking API call sequence as characteristic quantities with 2-gram of word unit, I could confirm that machine learning perform classification with the high accuracy rate.

CONTENTS

1. Introduction......6

2. Background......8

3. Approach......9

    3.1 Experiment procedure......10

    3.2 Machine learning......12

        3.2.1 Learning method......12

        3.2.2 Creating learning data......16

        3.2.3 Jubatus framework......20

4. Classification by ranking appearance frequencies......21

    4.1 Classification results by ranking appearance frequencies......21

5. Classification by using N-gram......26

    5.1 Applying N-gram......28

    5.2 Classification by N-gram......29

    5.3 Classification results by N-gram......30

6. Classification by using Import Address Table......31

        6.1 IAT......31

        6.2 Classification results with IAT......33

7. Classification by using API call sequence..39

7.1 Cuckoo Sandbox..39

7.2 Experiment of the classification with API call sequence...........................42

7.3 Classification results of API call sequence..44

8. Summary..47

References..49

1 Introduction

The malware's target had been personal and a large indefinite number of people before Internet was popular. However some specific person and company is recently targeted by the hacker. The purpose for generating malware has been shifted to the monetary one. According to the quarterly report of PandaLab in 2009, 71.32% among the new kinds of malware was Trojan [1]. These malwares are specialized in remote control. The way to send malware to the target is of via email combined with social engineering [2]. Even if the company train the employee for security issues, it may happen that they may be tricked the email contents. Finally they would open a file attached in email or click the URL putting on it. Once malware is executed, malware hide in a system process by process injection and silently work around. Some malware which is called rootkit disables anti-virus detection [3]. To intrude the target's system for certain, hacker implements 0-day exploits into malware. In this way, the more malware's detection is delay, the scale of damage would expand over the network.

In this thesis, I classified applications of PE format by machine learning which implemented the perceptron [5] and AROW [25] as algorithms. I focused on characteristic quantities included in operation codes, instructions, API lists in import address table. The number of learning dataset for goodware is 500 and for malware is 500. The number of testing dataset is 300 for goodware, 300 for malware. Analysis objects are limited to Win32 executable files which are PE format. I excluded the packed executable files in experiments of statical analysis. Because this packed ones hide the original code which might contain malicious code by compressing it. If I included them, it would indicate that we classify whether the packed-ware or not.

In the first experiment with statical approach, I ranked appearance frequency for operation codes as characteristic quantities. Then I provided them to the learning machine which calculated similarity and classified them. In conclusion, the machine could classify them with around 0.7 of accuracy rate.

In the second experiment with statical approach, I provided the instruction code which is evaluated by 2-gram and 3-gram extracting characteristic quantities to the learning machine. In conclusion, the machine could classify them with around 0.8 of accuracy rate.

In the third experiment with statical approach, I provided lists of API stored in IAT with the learning machine. In conclusion, the machine could classify them with around 0.83 of accuracy rate.

In last experiment with dynamical approach, I took the API call sequences which was obtained from Cuckoo Sandbox [41]. As characteristic quantities, I took arguments for some specific API. In addition, I cut high and low appearance frequency that were commonly seen between goodware and malware. As a result, the accuracy rate was around 0.90 on average.

2 Background

There are some reasons that it has been difficult for signature based method to detect malwares. According to McAfee, One reason is that malwares are automatically generated by some tools. 100,000 of subspecies are generated day by day [4] all over the world and it is thought that it will be more difficult for signature based method to detect malwares in the future. Also McAfee reported that the rates of increasing new malware samples grows year by year [4]. To fight back against unknown malwares, heuristic engine that can detect them by analyzing the behavior based on some characteristics was developed and implemented on some realistic anti-virus products. And further research for heuristic engines are proceeding as ever.

I expected that malware's behavior might be characteristic compared with goodware's one. As the view of functions of malware, for example, malware download the extension module from Internet to add new functions to itself [6]. Some malwares often register themselves into startup of Windows registry. Not to be killed themselves easily from task manager and not to be discovered with process monitoring tools, they tend to hide into system process by process injection [7]. To prevent an analyst from peeking the logic, they detect debugger and kill themselves [8]. These activities are not normally seen in the goodware. I thought that these special activities might make the potential bias among them. Therefore I expected that the similarity for the group of malware was seen even in the level of operation code, instruction, imported API and API call sequences.

3 Approach

We have mainly two approaches (statical approach and dynamical approach) to extract characteristic quantities for goodwares and malwares. In this paper, I took three statical approaches and one dynamical approach. The reason why I took place classifications by static analysis is because it does not need to run malware on the real or virtual machine and analysis would be done quickly. Also it is needed to use samples as much to calculate more accurate rates. However I have some issues to disassemble some dataset. Because, in statical approach, it was not reasonable to use packed executable files. Malwares themselves are compressed by some packers and malware's behavior is almost invisible. I considered that operation codes related for packing would cause the noises and be disturbed classifications. So I skipped files that were recognized as packed executable files by pefile module [9] which detects packed-ware with signature database [28] when the Jubatus client extracts characteristic quantities. In addition, this detection of packware is not perfect way and it pass through packed-ware not contained in the database. However I did not skip packed file when I implemented on the classification by API call sequences which is categorized into the dynamical approach. Because unpacking process is executed by a packer in the virtual machine. That is the advantage for the dynamic analysis.

I used the Jubatus framework as a classifier [10]. Because the usage of classifier is simple and I can concentrate to write the client code which provides the learning and testing data for it.

3.1 Experiment procedure

I show a flow chart of experiment step for statical approach in Figure 3-1. Firstly I prepare learning datasets for both classifying goodware and malware which are Win32 executable files. So I did not include the pdf, doc, xls, jpg, avi and the other format style. Although some malwares were often seen that they were embedded in the pdf file which exploits Adobe Reader [11], I focused on the windows executable files.

I scanned them with ClamAV [12] and checked not to immix datasets of malware with datasets of goodware. Next I gave learning dataset to the analyzer to extract characteristic quantity. Analyzer extracts appearance frequency from disassemble code and ranks operation codes. So I treat with ranked data as characteristic quantity. When getting classifier to learn characteristic quantities, I evenly have to give them for classifier. After I gave all learning datasets of malwares for classifier, then I gave all learning datasets of goodware for the learning machine. After getting the machine to learn all datasets, the classifier could not classify the test datasets correctly at all. So I noticed that it was important to learn each data for classifier alternately such as learning goodware, malware, goodware, malware and repeating. Finally I input test datasets to classifier which judge malware or not.

I evaluated each rates of FP(False Positive), TN(True Negative), TP(True Positive), False Negative(False Negative) as results. These rates are represent from 0 to 1.0. I summarized each meaning in Table 3-1.

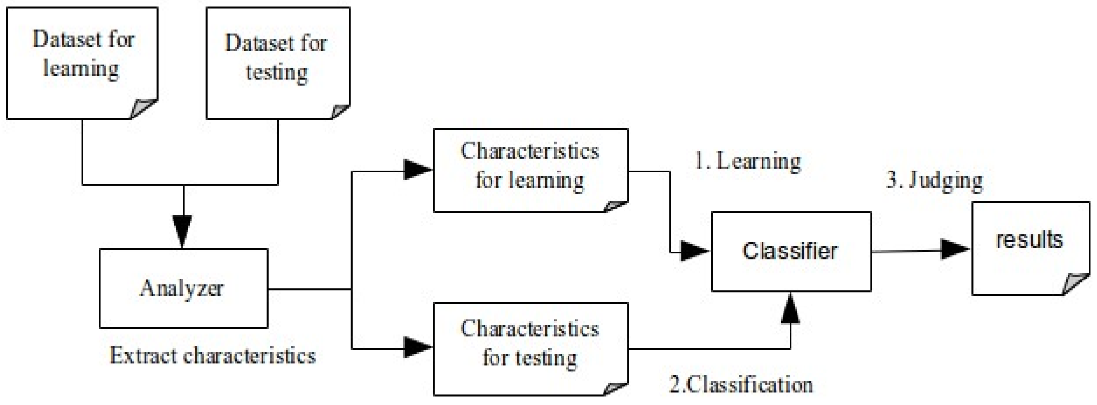


Figure 3-1: flow chart for experiment in statical approach

FP	Judging goodware as malware
TN	Judging goodware as goodware
TP	Judging malware as malware
FN	Judging malware as goodware

Table 3-1: four rates for evaluating classifications. These rates were calculated throughout all of the experiments.

3.2 Machine learning

I introduced perceptron algorithm as machine learning. Perceptron algorithm was published in 1957 by Frank Rosenblatt who was psychologist as well as computer scientist. He suggested that perceptron algorithm was the model of information processing with neural network (Figure 3-2). Multiplying corresponding weight with each inputs which are calculated as sums. Finally machine judges the results with binary value (Yes or No). Nowadays perceptron algorithm has been utilized as linear classifier in a wide range of area although it had been thought as importance for brain science field in the past.

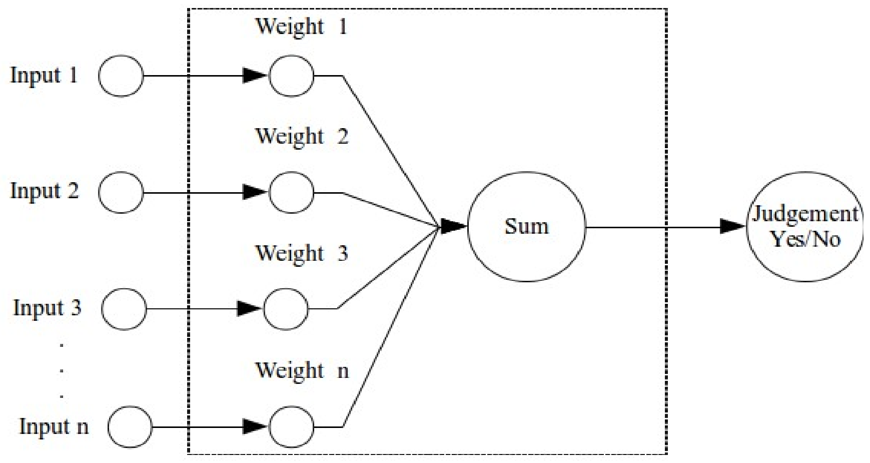


Figure 3-2: a classification model of perceptron

3.2.1 learning method

I show learning algorithm in Equation 3-1 [13]. T are learning data and are distributed in 2 dimensional coordinate. The learning step is following.

1. I decides hyperplane with properly. (To decide constant values a, b and c)

2. I get the machine to learn data with discriminant function f(x,y). From the result of it, the machine improves the hyperplane with gradient and intercepts by following the equation. This step means rotation and parallel movement for hyperplane on 2 dimensional plane.
3. Repeat step 2.

$$\begin{gathered}
\mathit{If}\ T(k)\,f(x,y)>0\ \mathit{then\ do\ nothing} \\
\mathit{else}\ T(k)\,f(x,y)\leq 0\ \mathit{update\ weight} \\
f(x,y)=ax+by+c \\
f:\mathit{discriminant\ function} \\
T:\mathit{learning\ data}\ ,\ k:\mathit{index\ for\ learning\ data}
\end{gathered}$$

Equation 3-1: learning algorithm

Obeying learning step 1, as an example, I let a = 0, b =1, in initial stage, it becomes in Figure 3-3. Circle and triangle shape represents learning data. I label 1 or -1 for each group that we want to classify. For example, Let goodware to be 1, malware to be -1. One of triangle data is not classified correctly as a first circumstance. When wrong data is input the discriminant function f(x,y), f(x,y) outputs negative value. Then gradient and intercept would be improved. Repeating this process, machine calculates the final optimum hyperplane.

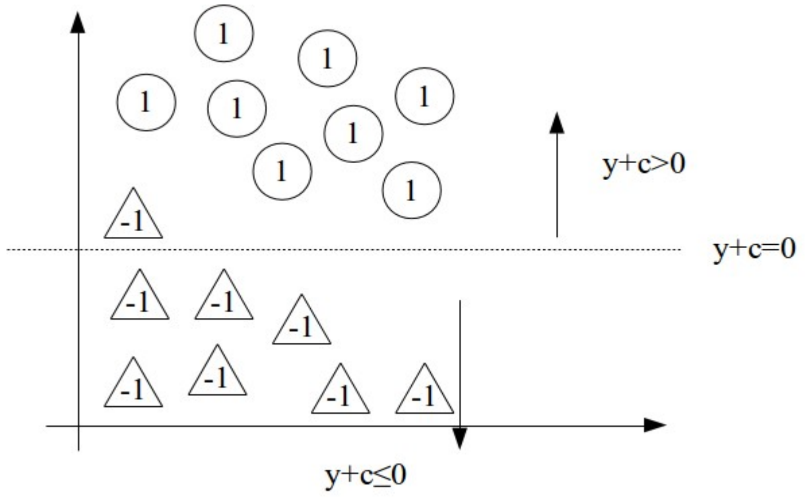


Figure 3-3: initial stage for classification in perceptron algorithm

I draw the image of the final stage for classification with 2-dimensional coordinate in Figure 3-4.

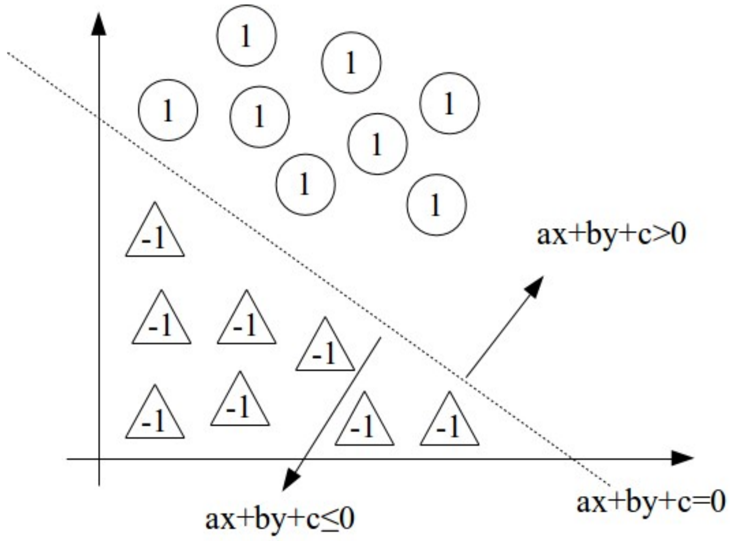


Figure 3-4: final stage in classification for perceptron algorithm

This figure represents the final discriminant function which had been updated by perceptron algorithm. We can understand that the accurate hyperplane to classify the data is drawing. In this way, the thing that can be classified completely is called linear separable. However succeeding in classifying the data always does not happen. Because some noise are usually included in realistic cases. For example, we have to consider about the goodware's sample that behave as if it is malware. In perceptron algorithm, we ignore such a noise. This problem could be mitigated in Adaptive Regularization of Weight Vectors (AROW) [25]. This algorithm is of improvement for Confidence Weighted Linear Classification (CW) [24]. CW method could deliver higher performance in case that the characteristic dimension number is big. But that tends to be weak for noise, while AROW method improved the weakness.

Lastly I summarize the general expression in N-dimensional plane for perceptron algorithm in Equation 3-2.

$$\begin{gathered}
y = \boldsymbol{w}^{T} \boldsymbol{x} \\
\textit{if } y > 0 \textit{ then do nothing} \\
\textit{if } y \leq 0 \textit{ do update weight vector} \\
\textit{update with } \boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \mu \boldsymbol{x} \\
y : \textit{classification results} \\
\boldsymbol{w} : \textit{weight vector} \\
\boldsymbol{x} : \textit{input vector} \\
\mu : \textit{learning constant}
\end{gathered}$$

Equation 3-2: perceptron algorithm in N-dimention

3.2.2 Creating learning data

First of all, I disassembled executable files and took appearance frequency for operation codes. Operation codes which represent machine code as mnemonic are stored in executable file as PE format. Mnemonic for assembly code is replaced to machine code for readability. And it corresponds with machine code on 1-to-1 level.

As the first step of taking characteristic quantities, I looked for code section and disassembled with libdasm [14] when selecting files in jubatus client. I sorted them by ranking for each operation code. Operational code is basically extracted from .text section (code section). To understand the structure of windows executable files, I show the PE header structure in Figure 3-5.

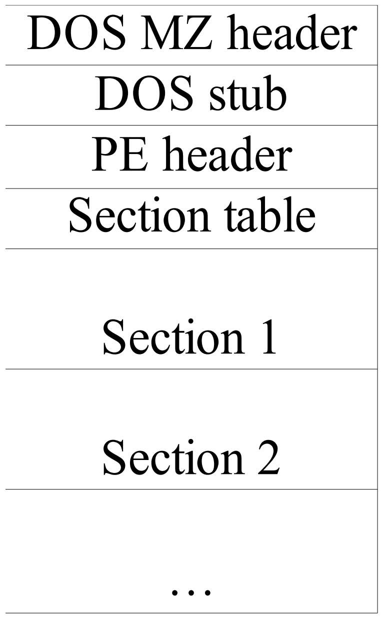


Figure 3-5: The structure of PE header

DOS MZ header is first area in the structure of the PE file. It is defined as IMAGE_DOS_HEADER [15] defined in winnt.h. The first member e_magic is 2 bytes. In the case of Windows executable files, character string “MZ” is set as a signature. Last member e_elfnew is offset to the PE header region stored as Little-endian. This offset means offset used for the file not used for the memory loaded .

In DOS stub region, it can work only on MS-DOS. Normally the code that may show the message “This program cannot be run in DOS mode” is written in.

PE header is defined as IMAGE_NT_HEADERS [16] in winnt.h. It has 3 members, Signature, FileHeader, Optional_Header. Moreover Optional_Header has important members to calculate the address of some section. First we need to know AddressOfEntryPoint and ImageBase. AddressOfEntryPoint is the address to start the program as RVA (Relative Virtual Address) which indicates the relative offset when the program has loaded. ImageBase represents ideal starting address when the program has loaded. Normally address loaded in the memory for executable file is 0x00400000 and for DLL file is 0x10000000. However it always wont be loaded on the same address since memory relocation might be taken place. Therefore we can calculate address of the entry point by adding AddressOfEntryPoint with ImageBase (Equation 3-3). FILE_HEADER has NUMBEROFSECTIONS member which represents the number of sections.

$$EntryPoint = AddressOfEntryPoint + ImageBase$$

Equation 3-3: getting staring address of section

Section table is defined as IMAGE_SECTION_HEADER [17] in winnt.h. There are Name, VirtualSize, VirtualAddress, SizeOfRawData, Characteristics as interesting member. We can know the code section from Name member. Basically code section is named as “.text”. When I disassembled the executable file, I found the string “.text” and took frequency for operation codes. VirtualSize represents RVA for the starting address when the program has loaded. SizeOfRawData represents section size on the disk. Actually We can calculate the end of section by adding VirtualAddress + SizeOfRawData. However insignificant codes are included at the end of code section as padding or something for some reasons. I thought this would affect taking the appearance frequency. That's why I calculated with the end of section as VirtualAddress + VirtualSize.

Packer compresses real code section and has the special section for unpacking. For example typical packer UPX (Ultimate Packer for executable) [18] creates UPX0 and UPX1 section during packing an application. I show the packer's unpacking image in Figure 3-6 [19].

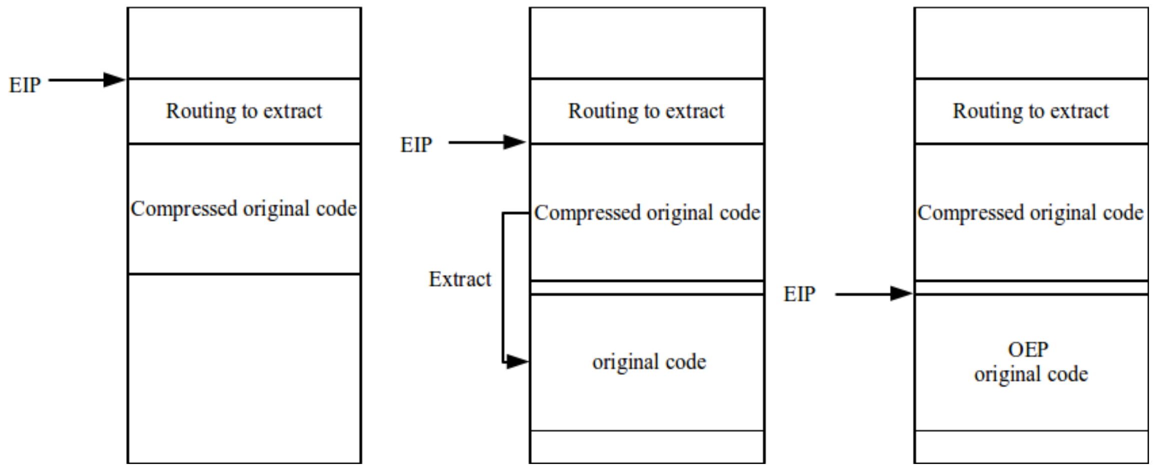


Figure 3-6: Unpacking process by a packer

I explain how the unpacking step takes place. As the first step, packer jump to subroutine for extracting. After extracting original code, packer sets the EIP register to OEP (Original Entry Point) pointing original code. Then the original code correctly is executed. In statical analysis, original codes are compressed and are invisible. However some of solutions is being suggested.

One of the way is to unpack them on the virtual machines by dumping memory image. The good point is that it does not depend on kinds of packer's algorithm. However some malware is able to detect that it is being running on virtual machine and will change behavior. So the countermeasure to pretend the realistic machine is recommended for additional treatment.

In particular, it is thought as effective way to detect the execution for protected page. When packer finished extracting original codes, packer would attempt to jump and execute the head of original code. If we use memory-breakpoint, it observes execution, read and write on the memory. OllyBoneE [20] changes page table entry to prohibit access from user mode. When reading and writing for page memory are taken place, page fault exception is triggered on the page where breakpoint is set. After page fault exception, it judges whether accessing memory is for execution or for reading or for writing. If memory accessing is for execution, it calls exception handler for step execution (INT 1). When detecting execution for protected page, it dumps all memory image including original codes. This software that utilize the technique is famous for OllyBonE which works as OllyDbg [21] plugin.

In the case of applying it on Vmware, OllyBoneE is needed to modify and rebuild the source codes. Because the way to treat with TLB (Translation Lookaside Buffer) on is different from realistic machine. TLB exists for both read and write independently [22].

3.2.3 Jubatus framework

Jubatus framework provides learning machine which implements classification, regression, recommendation, graph mining, anomaly detection, clustering. In my thesis, although I focused on classification, it is also helpful to find malware having similar habitats by recommendation.

Perceptron algorithm is categorized into classifier. Jubutus framework supports another classification algorithm such as PA [23], CW [24], AROW [25], NHERD [26]. In the case of choosing these algorithm, we can specify the sensitivity parameter for learning as constant value. After deciding on implementation, I give a type of algorithm for Jubatus server as configuration file which are written in JSON format. When starting Jubatus server, It must provide configuration file as mandatory argument. Then Jubatus server works and waits on TCP port 9199.

The client application sends learning datasets to Jubatus server. At this moment, learning datasets must be set as associative array. Associative array consists of label and value. For example, If label is “mov”, value is 1, associative array becomes {“mov”, 1}. In addition, it needs to convert it to Datum data format [30]. Client application sends correct label and Datum data as tuple to get the Jubatus server to learn. After completing to learn, the client send the test dataset to the server and server start the classification.

During classifying an application, it gives testing data in Datum format to server. Server answers as a label which is either goodware or malware and the corresponding score. Client treat with the label which has higher score than the other label's one as correct answer. Lastly comparing correct label with the label server answered, the client calculates rates of FP, TP, TN and FN.

4 Classification by ranking appearance frequencies

I used 500 samples for learning and 300 for testing dataset in Table 4-1. I prepared sample malwares as datasets from VirusSignList [27]. I excluded packed executable files from goodware and malware. As a signature database file, I used Bob /Team PEiD Signature [28]. I prepared a set of goodwares by independent way such as copying from “Program Files” folder in Windows XP and collected some softwares from Internet randomly.

	Malware	**Goodware**
Learning datasets	500	500
Testing datasets	300	300

Table 4-1: dataset used in classification

I took only operation code for ranking appearance frequency. For example, if disassembled code is “xor eax, eax”, I pick up only “xor” and count as +1 for operation code xor. Operation code always does not appear in the head of the instruction. In the case that segment over ride prefix and data segment over ride prefix include in instruction code, those prefix would appear in the head of instruction. So I skipped those code and searched operation code from the instruction and extracted as characteristic quantity.

4.1 Classification results by ranking appearance frequencies

I show the results of FP, TP, TN and FN rates for classification by ranking appearance frequency of operation codes in Table 4-2. This rank represents the maximum rank number when ranking appearance frequency for each datasets. In the case of rank 10, I took the rank of operation code up to 10 as characteristic quantity. As looking at column of rank 37, both TN and TP rates were the highest number among all ranking. This indicates the machine could classify malware and goodware with approximately 74% of accuracy. In rank 30, difference between TP and TN rates were big. This classification was unstable. Because the way to cut noises was wrong. I had to exclude the quantities that commonly highly seen in goodware and malware. As increasing ranking from 10 to 20, TN and TP rates were equal a little by little. From rank 50 to 200, TN and TP rates gradually became bigger difference. From the results, as comparing bias for operation code between goodware and malware, it is thought that it is the most effective that we take rank 20 to 50. But it cannot be applied to realistic heuristic detection with this rate. In the security field, detection rates must always keep the high accuracy more than 90%. Because the risk of false negative is quite serious in security field. However I could see bias for some operation codes through this experiment and relevance between frequency of operation code and malware.

Rank	**FP rate**	**TN rate**	**TP rate**	**FN rate**
10	0.34	0.66	0.63	0.36
20	0.29	0.70	0.78	0.21
30	0.12	0.88	0.55	0.45
37	0.25	0.75	0.74	0.26
40	0.12	0.88	0.54	0.46
50	0.22	0.78	0.69	0.31
60	0.12	0.87	0.55	0.45
70	0.06	0.93	0.44	0.56
80	0.21	0.78	0.68	0.31
90	0.89	0.50	0.5	0.5
100	0.07	0.93	0.34	0.65
200	0.08	0.92	0.37	0.63

Table 4-2: rates categorized in rank by perceptron algorithm

Next I changed the algorithm from perceptron to AROW under same condition of Table 4-1. AROW takes an advantage against noises when learning the dataset. I show the

results for AROW in Table 4-3. As we look in the most effective rank, it is rank 41 at 0.73 and 0.70 for TN and TP respectively. Compared with perceptron algorithm, it was not seen big difference among methods. The more the rank increases, however, the difference for TN and TP rate in perceptron algorithm was extremely bigger where the biggest difference for TN and TP rate was approximately 0.55. In contrast with that, the difference in AROW was more stable where the biggest difference for TN and TP rate was about 0.36. Therefore AROW method could provide the stable results than perceptron algorithm.
I took the average of ranking for learning data within top 50 and compared the average ranking for goodware and malware in Table 4-4. Note that I extracted only operation code which had big difference. Average ranking was calculated for each goodware and malware respectively and took difference with them. If ranking of one operation code become same as another one, I recognized them as same rank. Negative values represents that ranking of corresponding operation code is lower than goodware' rank, positive values are above than goodware's rank.

I analyzed that difference of these ranking affected classification. Malware used operation code treating with floating point than goodware. Especially fcmovne, fbstp and fsave operation codes had quite big difference and were often used by malware because difference value is negative value. Probally malware's code would tend to hold floating point operation. Otherwise the packeware not included in the signature database might have been used as normal executable files.
In addition, pcmpqtd, pmulhw and pandn also had showed big difference at rank 3, 7 and 25 respectively. The common point with them is operation for MMX which is SIMD extended operation. This difference affected greatly to classify malware and goodware.
Nop code is comparatively often seen at rank 10.
In the view of heuristic detection, this accuracy 70 % does not achieve to a practical level overall. There are various kinds of malwares as realistic problem. For example,

malwares must be categorized into some groups such as trojan, warm, rootkit, backdoor (RAT), downloader, fishing, botnet, spyware and so on. This time the machine classified them in a comprehensive manner. So malware's bias as characteristic function was not revealed dominantly because of mixing up all kinds of malware. I analyzed that I should classify them after malwares were categorized by their function and characteristic.

Although I detected packedwares to skip them with the method based on signature, some of packed applications might be minor and pass the detection with signature. They might be included in the datasets. To solve the packer's problem, it needs to implement automated unpacking system.

Rank	**FP rate**	**TN rate**	**TP rate**	**FN rate**
10	0.31	0.69	0.7	0.3
20	0.21	0.79	0.64	0.36
30	0.38	0.62	0.76	0.24
40	0.30	0.69	0.79	0.20
41	0.27	0.73	0.70	0.29
50	0.45	0.54	0.83	0.17
60	0.49	0.51	0.82	0.18
70	0.5	0.5	0.86	0.14
80	0.50	0.49	0.83	0.16
90	0.53	0.47	0.82	0.18
100	0.49	0.51	0.78	0.22
200	0.46	0.54	0.76	0.24

Table 4-3: rates categorized by rank by AROW algorithm

Operation code	**Malware's ranking**	**Goodware's ranking**
pcmpqtd	3	98
adc	7	39
pandn	7	114
popa	10	44
nop	10	41
movaps	12	50
xchg	13	53
fbstp	15	74
fist	15	51
into	15	70
insb	16	53
pushf	18	60
insv	18	51
scasd	19	74
bswap	19	63
pusha	20	61
jno	24	63
loop	24	64
lock	24	73
fcmovne	24	107
fsave	25	98
pmuhw	25	94

Table 4-4: Average ranking for goodware and malware.

5 Classification by using N-gram

N-gram is a kind of the way to take characteristics quantity from text. N-gram model is the language model that counts the combination of N length character string in some text message. Claude Elwood Shannon who was known as the father of information theory invented as the language model of N-gram. There are mainly 1-gram, 2-gram and 3-gram which depend on taking minimum unit of character. As problem to attribute to take N, if we make N bigger, it would tend to lower accuracy and the combination of co-occurrence rise exponentially. In the case of 1-gram, it creates an index which is based on 1 character. In the case of bi-gram, it creates index for sequence of 2 characters. N-gram is practically implemented in Google search engine to find some word since when the word had been used and how much of the appearance frequency on each year [29].

I explain how the N-gram works with following example sentences.

Example sentence1: This is an apple.

Example sentence2: That is an apple

Now we consider how the sentence is separated when I apply the 2-gram to this sentence. I separate them on each 2 characters for each sentence. And I assign the sentence ID for separated characters. I show the 2-gram table for both sentence1 and 2 in Table 5-1 respectively. Next I combined the sentence1 applied 2-gram with of sentence 2 in Table 5-2. In the table, under score (_) represents space. Now we think of an example sentence “An apple is eaten by that boy” to classify it. First of all, We divide the string apple into “ap”, “pp”, “pl” and “le”. As we refer to Table 5-2, these strings have ID 1 and 2. Secondly we search “that”. As same way, “that” is taken part in “th”, “ha” and “at”. As narrowing a search from the Table 5-2, we can find that sentence 2 is possibly candidate. In this way, N-gram would be useful to calculate the degree of similarity of the text message. I would apply N-gram to classify goodwares and malwares by a machine learning.

Sentence1		Sentence 2	
2-gram	Sentence ID	2-gram	Sentence ID
th	1	th	2
hi	1	ha	2
is	1	at	2
s_	1	t_	2
_i	1	_i	2
is	1	is	2
s_	1	s_	2
_a	1	_a	2
an	1	an	2
n_	1	n_	2
_a	1	_a	2
ap	1	ap	2
pp	1	pp	2
pl	1	pl	2
le	1	le	2

Table 5-1: a piece of characters applied 2-gram for sentence 1 and 2

2-gram	Sentence ID
th	1,2
hi	1
is	1,2
s_	1,2
_i	1,2
is	1,2
s_	1,2
_a	1,2
an	1,2
n_	1,2
_a	1,2
ap	1,2
pp	1,2
pl	1,2
le	1,2
ha	2
at	2
t_	2

Table 5-2: Two sentences applied 2-gram are merged

5.1 Applying N-gram

Although it is effective to classify the similarity of some documents, I considered it was not good to apply N-gram with character unit in the disassembled code. Moreover there are some similar operation code for example mov, movs and movsx. So I would suggest N-gram with word unit. For example, assemble code “push dword [ebp+0x18]”. I divide them into 2-gram of word unit. Then it becomes “push dword”, “dword [ebp+0x18]” and “[ebp+0x18]”. In this way, similar operation codes such as “mozx”, “movl”, “movsx” are treated as independent operation code. I expect that N-gram with word unit could classify malware better than character unit. Because characteristics of assembly structure might be more emphasized if we consider as word unit. I used the plug-in of word unit for N-gram from github of Jubatus [31].

5.2 Classification by N-gram

As same as the experiment by ranking, I prepared 500 samples for learning dataset, 300 for testing dataset (Table 5-3).

	Malware	**Goodware**
Learning datasets	500	500
Testing datasets	300	300

Table 5-3: dataset used in classification

This time I implemented experiment with perceptron and AROW algorithm combined with 2-gram and 3-gram for character unit and word unit. AROW has sensitivity parameter for learning data. This time I set sensitivity parameter as constant value 1. In addition, I changed the way to treat with weight for characteristic quantity on Jubatus configuration of the server. Jubatus's configuration is JSON formatting. There are two kinds of weight configuration [30], sample_weight and global_weight. Sample_weight is related to pair of key and value uniquely. Global_weight is calculated from current results. About “bin” as one of an option, weight is always constant value 1 and I set sample_weight and global_weight as string “bin”.

5.3 Classification results by N-gram

I show the classification results by 2-gram and 3-gram in Table 5-4. In N-gram column, 2(word) represents that 2-gram implemented by word unit. Otherwise it is N-gram with character unit.

Method	**N-gram**	**FP rate**	**TN rate**	**TP rate**	**FN rate**
perceptron	2	0.7	0.3	0.96	0.03
perceptron	3	0.87	0.12	0.98	0.01
perceptron	2(word)	0.19	0.8	0.86	0.14
perceptron	3(word)	0.15	0.85	0.81	0.18
AROW	2	0.77	0.22	0.97	0.02
AROW	3	0.31	0.69	0.85	0.15
AROW	2(word)	0.23	0.76	0.89	0.11
AROW	3(word)	0.25	0.74	0.91	0.09

Table 5-4: N-gram classification results for perceptron and AROW algorithm

From the result, we can see that the perceptron with both 2 and 3 grams recorded high accuracy rate for TN and TP at (0.8, 0.86) and (0.85, 0.81) respectively. In contrast with 2-gram and 3-gram with character unit both for perceptron and AROW algorithms, the gap of TN and TP was large. This indicates the classification by N-gram with character unit did not work well. Therefore n-gram for word unit is said to be effective for classification based on disassemble code. AROW with 2 and 3 word gram showed the lower accuracy than perceptron for TN and TP rates.

6 Classification by using Import Address Table

Import Address Table (IAT) is stored to call API which exists in DLL in PE format file. For example, MessageBox function is registered in User32.dll. Windows application call it by importing API into IAT. Win32 API is involved in a wide range of function, controlling process, hardware device, file system, creating process, drawing the picture, synchronizing thread and so on.

As seen the behavior on malware, API to register malicious executable files to startup is also imported in IAT when loading application. It is not always malware import API which is related to registry. In addition, some goodwares also import such kind of API. However what malware would like to achieve must be different from goodware's one. So I expected that the bias could be seen when comparing imported API with goodware. I treated with list of API as characteristic quantity to classify them with.

6.1 IAT

To obtain the list of IAT, we need to access IAT which is stored in PE format file. I show the IAT structure in Figure 6-1 [32].

First of all, start point to access IAT is to examine at DataDirectory[1] which is member of Optional Header in PE header. DataDirectory[0] is for export table. DataDirectory[1] is for import table, pointing to structure IMAGE_IMPORT_DESCRIPTOR in import table. In IMAGE_IMPORT_DESCRIPTOR, member OriginalFirstThunk is a pointer to IMAGE_THUNK_DATA in import-lookup-table and FirstThunk is a pointer to IMAGE_THUNK_DATA in IAT. The member Name represents the name of DLL. So IMAGE_IMPORT_DESCRIPTOR exists for each imported DLL. If we import the API of LoadLibraryA and MessageBox, the number of IMAGE_IMPORT_DESCRIPTOR would be 2, each member Name in IMAGE_IMPORT_DESCRIPTOR are Kernel32.dll and User32.dll.

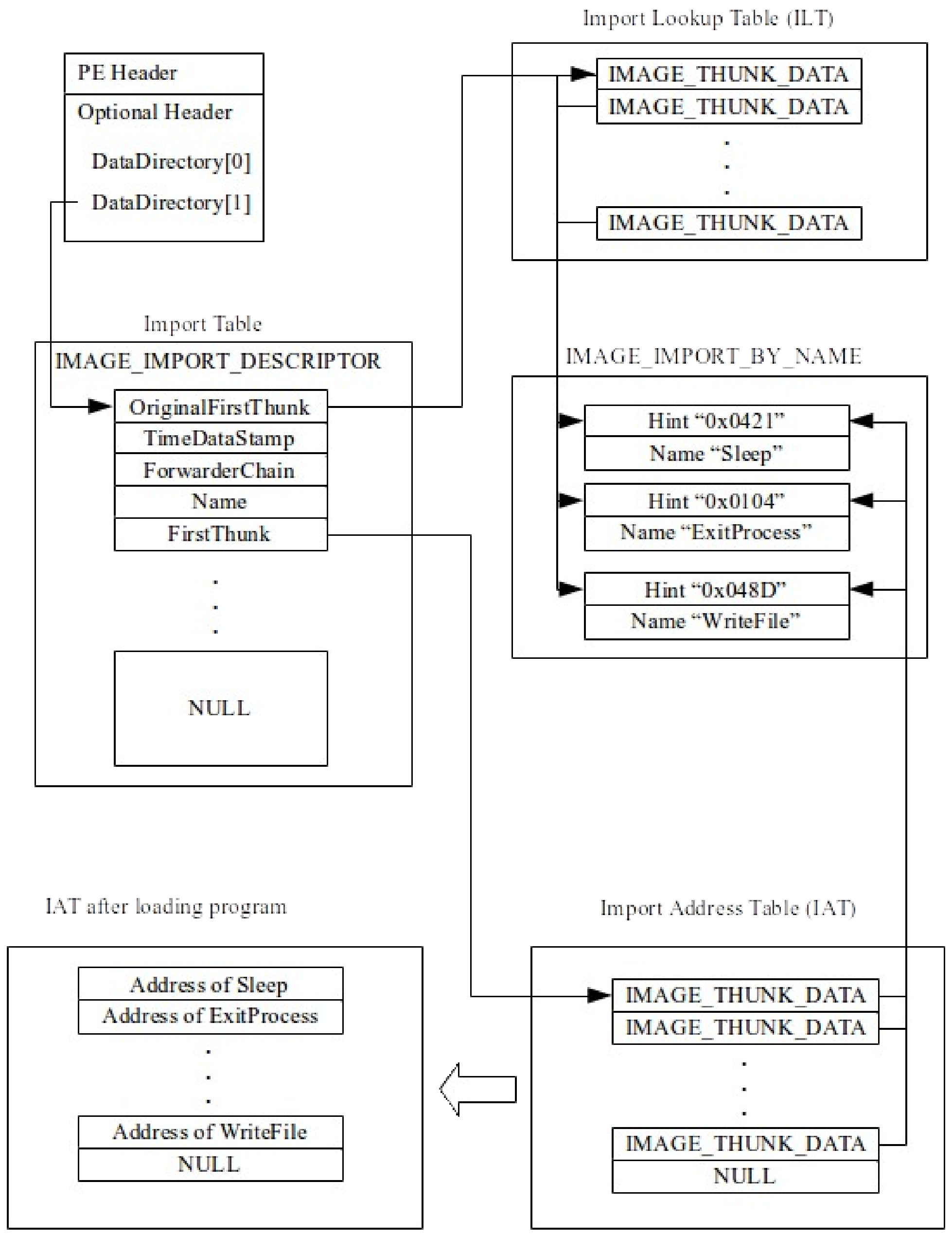


Figure 6-1: IAT structure

IMAGE_THUNK_DATA is pointer to structure IMAGE_IMPORT_BY_NAME which has member hint and API's name. Before loading application, IAT and ILT points to same IMAGE_IMPORT_BY_NAME. After an application has been loaded by a loader, API address is resolved and IAT rebuild with actual virtual address for it. However it is not always that IAT and ILT point to same structure for each API. Because Bound-import method should be considered as resolving the address of API before loading application. The method is to replace IAT's value with actual address of API beforehand. In instance, cmd.exe and calc.exe are implemented bound-import method [32].

6.2 Classification results with IAT

I basically took only API name imported in IAT. Because it is not always that import address pointed to thunk value is not same even if the same API is imported in each applications. So I used API name imported in IAT as characteristic quantity. Finally I recognized DLL name as key for learning data and API name concatenated character space with next API name in same DLL name as value. In some application, the size of import address table is zero or the number of API in IAT is zero. I skipped to classify such a file. Also I did not include packed file as data.

Classification method was implemented with perceptron and AROW algorithms. I applied 2-gram to extract characteristic quantity. Text message is divided by space character and treat with lists of divided character as characteristic quantity. I show the result in Table 6-2.

Algorithm	**FP rate**	**TP rate**	**TN rate**	**FN rate**
Perceptron	0.17	0.82	0.85	0.15
AROW	0.23	0.77	0.92	0.08

Table 6-2: classification result with API name

As we can see the table, TN and TP were 0,82, 0.85 respectively. It suggested that it could possibly classify malware only with API name. I analyze that a set of API could be different and biased between goodware and malware.

To obtain the clue for the result, I took the appearance frequency for API stored in IAT for malware and goodware in Table 6-3. I focused on API which was higher frequency than goodware's frequency.

The way to read the table is ,for example, about ExitProcess. 285 in 500 malwares imported the API and goodware's appearance frequency was 275.

Now we consider about the reason ExitProcess function is often called. I analyzed that malware immediately terminates itself if detecting malware is attached by debugger. If not, malware tends to process injection to hide itself. At the time, I expected that ExitProcess would be called.

About appearance frequency of LoadLibraryA is 273, while goodware was 291. The rate of usage was 55 % in average in malware. It indicates that malware's behavior is strongly tied up with external DLL file than goodware. I believe that the function must be surely meaningful classification level even if the number of samples increases.

GetStartupInfoA was seen in malware name Generic.BackDoor.U which stole the system information of the created process [33]. As seen in malware name Generic.Fake.Alert.lw as well [34], GetStartupInfoA could be thought as possibly being abused even in another malware. The rate of appearance frequency was 34% and showed high value.
VirtualFree and HeapCreate are API related to control memory. VirtualFree was called after VirtualAlloc to release the reserved memory. Although it seemed to be same frequency with VirtualAlloc both in goodware and malware, VirtualFree imported with simgle was often seen in malware. The rate of appearance frequency was 33%.

API	**Rank**	**Mal's freq(x/500)**	**Good's freq(x/500)**
ExitProcess	2	279	269
LoadLibraryA	3	273	255
VirtualFree	7	208	191
GetStartupInfoA	12	169	152
HeapCreate	23	144	114
GetStringTypeA	39	124	120
LCMapStringA	68	99	82
FreeEnvironmentStringsA	82	87	70
__vbaExceptHandler	97	77	4
ShellExecuteA	107	74	63
CopyFIleA	112	72	41
GetSystemDirectoryA	190	51	38
SetFileAttributesA	197	50	35
UuidCreate	287	36	10
__vbaFreeStrList	425	28	3
__vbaVarMove	466	22	2
OpenSCManagerA	512	19	8
InternetReadFile	527	18	8
OpenMutexA	547	17	5
OpenServiceA	594	15	5
InternetOpenA	600	15	2
ControlService	614	15	8
WriteProcessMemory	624	14	9
CreateServiceA	865	9	3
CreateRemoteThread	915	8	3

Table 6-3: Average ranking for malware and appearance frequency for both malware and malware by API imported in IAT

CopyFileA would be called when malware copy itself and resources to the disk space. GetSystemDirectoryA is to obtain the path of system directory. It would be considered that malware duplicates itself to the system directory. According to analysis report of new threat by IPA in 2009, GetSystemDirectoryA was identified to be called [35]. After copied itself, SetFileAttributesA would be called to change it to hidden attribute.

InternetOpenA and InternetReadFile are a set of API to download files with HTTP and FTP protocol from Internet. Malware can download a malicious file encrypted with SSL/TLS because InternetOpenA supports SSL flag as an argument.

OpenSCManagerA, OpenServiceA, ControlService and CreateServiceA are to control service in windows. By registering malicious service on windows, malware can invoke its activity every system booting. These APIs are called with a bunch of processing service. At first, malware open service controller with OpenSCMManagerA. Then it can access to the service by OpenServiceA. ControlService is called at restarting and stopping service.

__vbaExceptHandler, __vbaFreeStrList and __vbaVarMove are imported in VBRuntime.dll. One could argue that malware developed in Visual Basic showed the characteristic is disposed to call these functions. The rate of use frequency was approximately 8%. However it does not work on the target machine that is not being installed VBRuntime library. As malware developer, they are supposed to develop malware which does not depend on the machine environment. My expectation attributes on the ease of development.

WriteProcessMemory is for process injection which is the technique to inject the malicious code to another process. For instance, malware inject malicious code into system process. Therefore we feel that it would not appear to be malicious only if we monitor process list in task manager. It is said to be great malware if as long as to hide and stay in the target system. CreateRemoteThread is called to execute the code copied by WriteProcessMemory. In addition, CreateRemoteThread is combinated with API LoadLibrary as DLL injection technique that force to inject malicious code into another process. The rate of frequency in use for these function were 2 % in malware. Despite invisibility is malware's important element, these function are implemented in minor ones.

ShellExecuteA is API for lunching another application and the percentage of appearance frequency was 17 % in total malware. In case of Worm/Conflicker, it would be used to execute another malware copied to temporary folder [36].

I analyzed the bias of API by appearance frequency affected the classification.

Next I show the list of API name which was used only for malware in Table 6-4.

API name	**Rank**	**Frequency(x/500)**
WSCEnumProtocols	308	34
DllUnregisterServer	546	17
GetProcessVersion	743	11
InternetCrackUrlA	827	9
GetVolumePathNameA	833	9
ExcludeUpdateRgn	875	9
DllGetClassObject	906	8
CopyFileExA	912	8
DebugSetMute	920	8
UpdateResourceA	944	8
CreateMailslotW	1036	6
InternetOpenUrlA	1053	6
EnumServicesStatusA	1109	6
CreateDesktopA	1139	5
URLDownloadToFileA	1172	5
WNetGetUserA	1635	3

Table 6-4: API list seen only in malware

WSCEnumProtocols is API in Ws2_32.dll to enumerate protocol that is available in the system. WnetGetUserA is to enumerate user account for establishing connection. GetProcessVersion is to get major and minor version for windows. These API could be helpful to steal information of target system.

InternetCrackUrlA, InternetOpenUrlA and URLDownloadToFileA are related to download another malware from Internet. InternetOpenUrlA supports protocol SSL/TLS and there are some malware that would bypass the sniffing by network administrator.

CreateMailslotW is used for communication with another process. I expect that the API is helpful for malware to communicate the other malware created by parent malware. In stuxnet 0.5, communication among malware process was established by a mailslot [37].

Although API name showed in Table 6-4 was low in appearance frequency in malware, it had never seen in goodware. It reflects the characteristics of minor malware. Almost of malware consists of subspecies generated by tool automatically. So I think that the this analysis would be helpful to find new species of malware.

7 Classification by using API call sequence

In this chapter, I classified the goodware and malware by API call sequences as characteristic quantities. API call sequences are obtained from dynamic analysis not from static analysis. As good point of dynamic analysis, it is possible to find activities of malwares in detail. For example, we can discover how, when, which files, registries and services the malwares access, write and read. Also we can trace procedures of generating child processes and injecting them. As first bad point of dynamic analysis, time to spend for analysis could be relatively longer than static analysis. For treating with more malwares, there are some ways to shape the method by parallelizing machines to save time or by sharing the analysis results in several machines. As second negative point, some malwares has the competence to detect that a malware itself is running on the virtual machines [38]. In the case of Virtual Box, it is quite easy for malwares to detect an execution environment by enumerating the process lists. Because windows operating system installed guest additions has the VboxService.exe as a process name. In the case of VMware, it is possible to detect an execution environment for malwares by sending some command to a backdoor port. If running on the virtual machine, it should response to malwares over the port. However it is avoidable to change the configuration of a guest machine [39]. Note that this way is not perfect way to avoid malware's detection. Because an exception should be caused at realistic machine when it is detected. By patching codes checking 'VMXh' with NOP codes, it is able to emulate the exception even on the virtual machine [40].

7.1 Cuckoo Sandbox

I introduced Cuckoo Sandbox as a tool of dynamical analysis [41]. It outputs the results formated JSON after analyzing executable files. The pros is that we can discover the lists of all API name imported and exported by malwares even if executable files are packed. In static analysis, the list of API name imported by a malware is hidden by a packer and actual API called by malware is restored after unpacking original code and rebuilding IAT. So we can skip the procedure of unpacking with Cuckoo Sandbox. A disadvantage point is that it takes more time to analysis compared with static analysis. In addition, I would like to recommend to divide the server of the Cuckoo Sandbox for goodwares and malwares to manage logs easily and to save time for dynamic analysis.

Cuckoo Sandbox supports to capture the network traffic. We can check the lists of hosts that malwares accessed and what kind of traffic malware sent with servers. I believe that it could be possible to use characteristic quantity from network traffic.

Next we move to the log of API call sequences output by Cuckoo Sandbox. I show the log structure of calls field in the result obtained from Cuckoo Sandbox in Table 7-1.

Field	**Description**
category	Either file system or system or network or registry or process or service or misc or synchronization
status	True(Success) or false(Fail)
timestamp	Timestamp when the API was called
thread_id	Id of thread
repeated	Times to be called repeatedly
api	API name
arguments	Stored as sets of argument name and the value

Table 7-1: structure of call filed in the result formated in JSON file from Cuckoo Sandbox

From the table, I took API name as a characteristic quantity for classification. For some specific API, I took some argument to clarify the different activities among goodwares and malwares (Table 7-2). API VirtualProtectEx is supposed to change an attribute of the memory region into executable attributes by malware. APIs RegSetValueExW, RegCreateKeyExW, RegOpenKeyExW, RegOpenKeyExA could be often used both by goodware and malwares. I thought that it was good to take Subkey's value because Subkey indicates the path of some registry key. I expected to make the difference malwares often access. API FindWindowA is used when controlling window belong to another applications. For example it is called when checking whether particular application is running or not. In some case, malware lists up the list of processes in the task bar and be sure the existence of anti-virus and network firewall. API LdrGetProcedureAddress is called to resolve address of API imported from DLL. I thought that this API could be useful to trace beginning of malicious activity. In LdrGetDllHandle and LdrLoadDll, we can identify which DLL and the exported function has been loaded in the application. LookupPrivilegeValueW would be used to check what the range of activities malwares can do. In NtCreateFile, we can monitor files and directories created by a process. However I filtered file handle and file name because it depends on every process. I believed that DesiredAccess indicated the wish of an application to operate file system.

When I was taking categories, I filtered for a misc category. Because most of API are GetSystemMetrics function in a misc category. I thought it would affect to the accuracy of classification as noises and the function would not be important to classify malwares.

API name	**arguments**
VirtualProtectEx	Protection
RegSetValueExW	ValueName, Type
RegCreateKeyExW	SubKey
RegOpenKeyExW	SubKey
RegOpenKeyExA	SubKey
FindWindowA	ClassName
LdrGetProcedureAddress	FunctionName
LdrGetDllHandle	FileName
LdrLoadDll	Flags, FileName
LookupPrivilegeValueW	PrivilegeName
NtCreateFile	DesiredAccess
NtFreeVirtualMemory	FreeType
OpenServiceW	ServiceName, Desired Access
OpenSCManager	MachineName, DatabaseName, DesiredAccess

Table 7-2: sets of API and arguments took as characteristic quantity

7.2 Experiment of the classification with API call sequences

I show the dataset used in the classification in Table 7-3. A virtual machine for executing datasets is Windows XP SP2 32bits.

In this time, I targeted at unpacked file as well as packed file. Because it does not need to unpack it manually and packer automatically unpack a file and extract original code. That's why I included packed files.

	Malware	**Goodware**
Learning dataset	500	500
Testing dataset	300	300

Table 7-3: dataset used in the classification

I show the flow chart of the experiment in Figure 7-1. Firstly I gave all datasets for Cuckoo Sandbox and Cuckoo Sandbox outputs result files formatted JSON. Secondly I provided them for Jubatus client. Jubatus client eliminates the noise which is high frequency among all datasets [42]. Because it is able to make classification more stable. Jubatus client extracts the characteristic quantities from output file formatted in JSON.

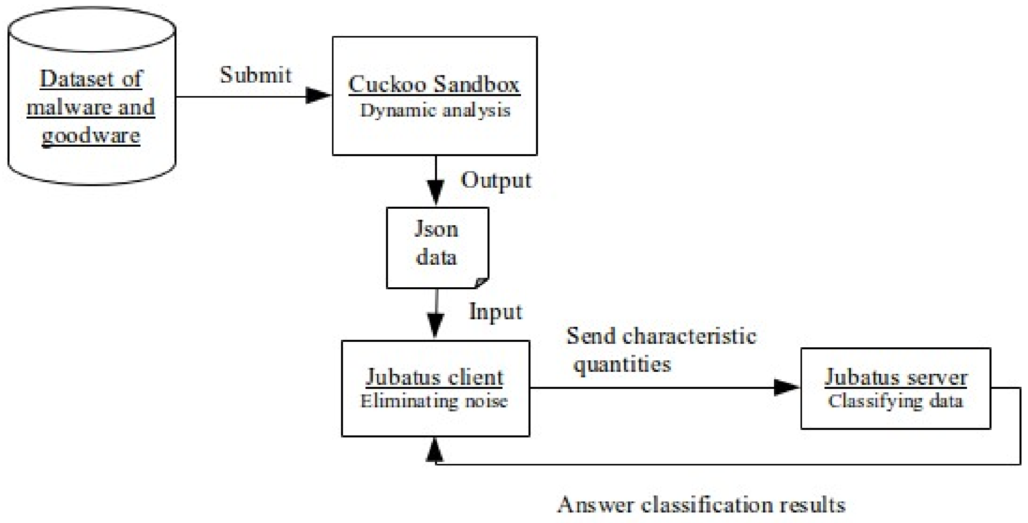
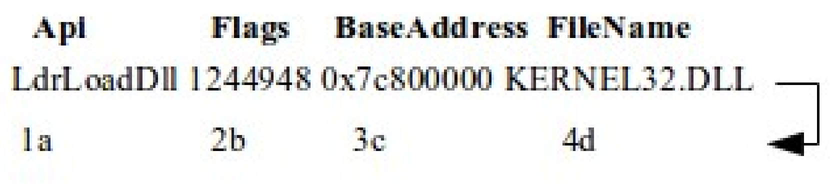


Figure 7-1: the flow chart of the experiment

Finally Jubatus client sends characteristic quantities to Jubatus server which responses with classification results to the client. Finally Jubatus server classify datasets with perceptron algorithm combined with 2-gram.

The rate for filtering noise is set to 0.90 as cutting high frequency. So the noise should be recognized to be cut off if appearance frequency is more than 0.90. The reason why I chosen those number is because they were the most stable among the experiments changed from 0.75 to 0.90 as high frequency.

While classifying datasets with string character which has long length, sometimes memory error happened on Jubatus server. So I dynamically assigned hexadecimal number with each fields to shrink the data length. (Figure 7-2). It could mitigate the risk for the memory error on Jubatus server.



Api	Flags	BaseAddress	FileName
LdrLoadDll	1244948	0x7c800000	KERNEL32.DLL
1a	2b	3c	4d

Figure 7-2: Convert API name to hexadecimal number

7.3 Classification results of API call sequences

Firstly I took only API name excluding its arguments as characteristic quantities. In addition, the process cutting noise was not implemented. I show the result in Table 7-4.

	FP rate	**TN rate**	**TP rate**	**FN rate**
Average	0.08	0.92	0.94	0.06
Minimum	0.15	0.85	0.98	0.02
Maximum	0.05	0.95	0.93	0.07

Table 7-4. Classification result for extracting only API name. The process cutting noise was not implemented. I experimented 20 times in total by randomizing the order of learning datasets.

I show the classification results in the case of including API's arguments in Table 7-5. I experimented 20 times in total and obtained the each accuracy rates. Comparing the result in Table 7-4, rates for an average and the gap for minimum are same. However we found that higher accuracy was obtained when including API's argument.

	FP rate	**TN rate**	**TP rate**	**FN rate**
Average	0.08	0.92	0.94	0.06
Minimum	0.15	0.84	0.96	0.04
Maximum	0.04	0.96	0.96	0.04

Table 7-5: classification result by API call sequence. Average, minimum and maximum rates are for 20 times of experiment randomizing the order of reading files.

As the evaluation from the results, I evaluated the average value, maximum value and minimum value among those experiments. On the average rate, TN rate was 0.92 and TP rate was 0.94. The gap for TN and TP at minimum rate were 0.12 . So the result became unstable in some case. In maximum rate, both TN and TP were 0.96 and it showed high accuracy and stable. We can see that the gap was not seen in TN and TP rates and the result indicates that the machine could classify correctly. If we always kept the maximum accuracy stable, we could implement it on realistic anti-virus products as heuristic engine.

However the result depending on the order of learning datasets are negative points for machine learning. To fill in this gap, this classification method should not work alone. It might be better to classify malware by corporating with another classification algorithm. As an example of an open source classification tool, Adobe malware classifier implements 4 classification algorithms (J48, J48 Graft, PART, and Ridor) and classifies them by collegiate system [43]. Adobe malware classifier classifies them with characteristic quantities obtained from statical analysis. When judging it, the classifier returns 1 as malware if all classification algorithms answers that the file must be malware in a unanimous. If classifier vote to goodware in a unanimous, then it returns 0 as goodware. Otherwise, it returns string value “UNKNOWN”.

8 Summary

In summary, I summarized the findings about machine learning. I list up the key point, suggestions, improvement and reflections throughout my experiments.

1. In the classification testing by ranking of operation code, the rate of accuracy marked approximately 74% in maximum. To make the classification stable, I implemented to filter noise by taking top rank of appearance frequency for operation code. However evaluating about which upper ranking is best accuracy depends on the datasets that we use. By filtering from the top rank 30 to 50, it showed stable accuracy for both AROW and perceptron algorithm. In addition we have to implement another way to cut noise. Because the result was unstable though the experiments. I should have cut noise by excluding the high and low appearance frequency commonly used in goodware and malware.
2. In the classification testing using perceptron and AROW algorithm of 2-gram and 3-gram with word unit, the accuracy rate was approximately 0.83 in a maximum. The accuracy rate from n-gram with word unit showed higher rates than n-gram with character unit. Therefore I would suggest that implementation for N-gram should be word unit.
3. In the classification testing using perceptron and AROW algorithm of API imported into IAT, the accuracy was approximately 0.83 for perceptron. In this time, AROW algorithm did not show the stable results. However I believe that I have to evaluate more accurate results by randomizing reading files and increasing the number of datasets. Because the sequence of learning files affects the accuracy of classification. This can be said on all experiments of machine learning. Also we could not implement the automatic unpack for packed file. As an agenda, I am going to investigate the automated unpacking system. In

addition, by evaluating frequency of API, we can find the clues about what kind of activity that malwares prefer and what kind of the minor function that they have. We might find the new species of malwares by machine learning.

4. In the classification testing using perceptron algorithm taking API call sequence as characteristic quantity. As our agenda, we have to review more which arguments of API should be taken as characteristic quantity. And process for cutting noise should be improved. I cut the noise from only API name. But I noticed that I should cut it from API name and its arguments.
5. I could not prepare enough datasets of goodwares. So it is required to collect more goodwares to calculate more accurate rate. In this time, installer file formatted in win32 executable was included in goodware. I think that it is not recommended to include an installer. Because we purely need to extract the characteristic of goodware. Therefore it is highly required to install goodwares on the victim's virtual machine in advance that does not work alone or only submit the executable file that can work with single to Cuckoo Sandbox.
6. Jubatus framework supports some kinds of algorithm for classification. In this time, I implemented perceptron and AROW algorithm. However there are another possibility to try PA [23], CW [24], NHERD [26]. I have to evaluate the results with those algorithms. In addition, among these algorithm, it is interesting for us to take collegiate system for classifying malware.
7. From the results of my experiments, I think that it is still difficult to apply in realistic malware detection. Because the rates of the accuracy are unstable and getting rid of noise by screening and filtering should correctly be applied in my experiments. In addition, we should consider about how to unpack packed application.

References

[1] Quaterly REPORT PandaLabs (July-September 2009) http://www.ps-japan.co.jp/uploads/fckeditor/pdf/pandalabs_2009Q3.pdf

[2] News of Malicious Email Campaign Used As Social Engineering Bait http://blog.trendmicro.com/trendlabs-security-intelligence/news-of-malicious-email-campaign-used-as-social-engineering-bait/

[3] The Anatomy of RTKT_ZACCESS http://about-threats.trendmicro.com/RelatedThreats.aspx?language=en&name=The+Anatomy+of+RTKT_ZACCESS

[4] Infographic: state of malware 2013 http://www.mcafee.com/us/security-awareness/articles/state-of-malware-2013.aspx

[5] Perceptron learning: http://page.mi.fu-berlin.de/rojas/neural/chapter/K4.pdf

[6] Upatre : Another day Another Downloader http://www.secureworks.com/cyber-threat-intelligence/threats/analyzing-upatre-downloader/

[7] Blackout: What Really Happened: Jamie Butler and Kris Kendall https://www.blackhat.com/presentations/bh-usa-07/Butler_and_Kendall/Presentation/bh-usa-07-butler_and_kendall.pdf

[8] Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware https://web.eecs.umich.edu/~mibailey/publications/dsn08_final.pdf

[9] pefile module http://code.google.com/p/pefile/

[10] Jubatus http://jubat.us/en/

[11] Exploit:W32/AdobeReader.K (F-secure) http://www.f-secure.com/v-descs/exploit_w32_adobereader_k.shtml

[12] ClamAV http://www.clamav.net/lang/en/

[13] University of Birmingham, lecture note http://www.cs.bham.ac.uk/internal/courses/intro-nc/current/notes/08-perceptrons.pdf

[14] Libdasm https://code.google.com/p/libdasm/

[15] IMAGE_DOS_HEADER http://www.nirsoft.net/kernel_struct/vista/IMAGE_DOS_HEADER.html

[16] IMAGE_NT_HEADERS http://msdn.microsoft.com/en-us/library/windows/desktop/ms680336(v=vs.85).aspx

[17] IMAGE_SECTION_HEADERS http://msdn.microsoft.com/en-us/library/windows/desktop/ms680341%28v=vs.85%29.aspx

[18] UPX(Ultimate Packer For Executable) http://upx.sourceforge.net/

[19] Analyzing malware: *fighting against infection incidents with free tool* (O'Reilly Japan) Page 78-79

[20] OllyBoneE http://www.joestewart.org/ollybone/

[21] OllyDbg http://www.ollydbg.de/

[22] Analyzing malware: *fighting against infection incidents with free tool* (O'Reilly Japan) Page 87-88

[23] Koby Crammer, Ofer Dekel, Shai Shalev-Shwartz and Yoram Singer, Online Passive-Aggressive Algorithms, *Proceedings of the Sixteenth Annual Conference on Neural Information Processing Systems (NIPS)*, 2003.

[24] Mark Dredze, Koby Crammer and Fernando Pereira, Confidence-Weighted Linear Classification, *Proceedings of the 25th International Conference on Machine Learning (ICML)*, 2008

[25] Koby Crammer, Alex Kulesza and Mark Dredze, Adaptive Regularization Of Weight Vectors, *Advances in Neural Information Processing Systems*, 2009

[26] Koby Crammer and Daniel D. Lee, Learning via Gaussian Herding, *Neural Information Processing Systems (NIPS)*, 2010.

[27] Free VirusSignList http://freelist.virussign.com/freelist/

[28] Bob / Team PEiD Signatures http://code.google.com/p/reverse-engineering-scripts/downloads/detail?name=UserDB.TXT

[29] Google Ngram Viewer https://books.google.com/ngrams

[30] Data conversion http://jubat.us/en/fv_convert.html

[31] N-gram plugin of word unit https://github.com/jubatus/jubatus-example/tree/master/malware_classification

[32] Analyzing malware: *fighting against infection incidents with free tool* (O'Reilly Japan) Page 63-69

[33] McAfee Generic BackDoor.ru http://www.mcafee.com/japan/security/virG2005.asp?v=Generic%20BackDoor.u

[34] Generic FakeAlert.lw http://www.mcafee.com/japan/security/virG.asp?v=Generic+FakeAlert.lw

[35] analysis of new threat by IPA in 2009 https://www.ipa.go.jp/files/000017746.pdf

[36] Worm/Confliker http://www.avira.com/de/support-threats-description/tid/4474/tlang/en

[37] Stuxnet 0.5: Command-and-Control Capabilities (Symantec Official blog) http://www.symantec.com/connect/blogs/stuxnet-05-command-and-control-capabilities

[38] The Dead Giveaway of Vm-Aware Malware http://www.fireeye.com/blog/technical/malware-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html

[39] Bypassing Virtual Machine Detection on Vmware Workstation http://www.unibia.com/unibianet/systems-networking/bypassing-virtual-machine-detection-vmware-workstation

[40] Analyzing malware: *fighting against infection incidents with free tool* (O'Reilly Japan) Page 133-136

[41] Cuckoo Sandbox http://www.cuckoosandbox.org/

[42] Analysis of massitve amount of API call logs collected from automated dynamic malware analysis systems http://www.iwsec.org/mws/2013/manuscript/3A1-4.pdf

[43] Adobe Malware Classifier http://sourceforge.net/projects/malclassifier.adobe/