2018 42nd IEEE International Conference on Computer Software & Applications

The Impact of Lightweight Disassembler on Malware Detection: An Empirical Study

Donghong Zhang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangdh@ios.ac.cn

Zhenyu Zhang [*]
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

Bo Jiang
School of Computer Science and Engineering
Beihang University
Beijing, China
jiangbo@buaa.edu.cn

T.H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

***Abstract*—Malicious software poses serious threats to our lives, and the activity to detect malware is becoming more and more important. An effective approach is to train a classifier using known software samples and malware samples, and recognize malware from new software. To do that, a recent popular trend is to use OpCode, which is extracted from executable modules, as an expression of software entities to drive machine learning. However, we found that the effectiveness of such a framework highly suffers from having insufficient samples, which is caused by the low success rate of disassembly due to the intrinsic complexity of the problem. In this paper, we propose to increase the success rate of disassembly by allowing inaccurate disassembling, with the attempt to increase the number of successful disassembled samples to improve OpCode-driven malware detection. We built a lightweight disassembler D-light based on the linear swap disassembly method to avoid known issues with the recursive descent manner of IDA Pro. We carried out experiment to evaluate the performance, effectiveness, and other design factors of adopting D-light and IDA Pro as disassemblers for malware detection. The empirical study shows the D-light is both more efficient and more effective than IDA Pro in supporting malware detection.**

***Index Terms*—Malware detection, OpCode, disassembly, D-light, IDA Pro, linear sweep.**

I. Introduction

The Internet is constantly undergoing attacks from different attack surfaces. A typical way to launch such attacks is through malicious software (aka "malware"), which may include worms, viruses, Trojan horses, etc. Worms are self-contained programs that exploit vulnerabilities in the software running within the network. In contrast, virus infects an existing program by injecting malicious code into it. When the infected program executes, the malicious code will be propagated to other programs. Trojan horses usually pretend to be benign programs, but behave maliciously [17]. When such malware are spread over the computer networks, they will damage computer systems, delete user data, or leak privacy information. Due to the rapid development of computer technologies, there is an exponential growth in the number of new signatures released every year [23]. In [9], Symantec reported 2,895,802 new software systems in 2009, as compared to 169,323 in 2008. Panda Security reported the amount of discovered software is 70 million in 2014, however, it increased rapidly to 3,050 million in 2015. It is crucial to detect and eliminate malware in a prompt manner [38].

There are many widely used techniques for detecting malicious software. The dynamic analysis technique is based on behavior analysis. The malware detection system analyzes the information collected at runtime, such as system calls, network access, file manipulations, and other malicious behaviors [16], [29], [31]. Although dynamic analysis is more effective and does not need the executable to be unpacked or decrypted, it is not perfect and often both time- and resource-consuming, thus raising scalability issues. First, it is not always easy to simulate the conditions where the malware executes. Second, the required time to observe the appearance of a malicious activity is unclear for each malware, which makes it very time-consuming. Third, more and more malware can detect the running environment, such as network adapter, processor type, and so on. Once they find the runtime environment is suspicious of being monitored, they will behave normally [42]. Moreover, anti-virus software vendors are facing huge numbers of suspicious files every day [10]. Those files are collected from various sources, such as honeypots, third-party providers, and the files reported by customers. Furthermore, some malicious behaviors are hard to observe. It is therefore very difficult to detect unknown threats using traditional analysis method, especially facing so many types of software [38].

The main advantage of static analysis is that it is able to detect a malware without actually executing it. In static analysis, information about the executable or its expected

[*]: All correspondence should be addressed to Dr. Zhenyu Zhang at Institute of Software, Chinese Academy of Sciences. Tel: (+8610) 62661630. Fax: (+8610) 62661627. Email: zhangzy@ios.ac.cn.

behavior is analyzed based on binary or source code of the malware. So it is very efficient and could provide rapid classification results [28]. One of the most widely used static analysis technique is the signature-based method, which relies on the identification of unique strings in the binary code [10]. The signature-based method extracts some unique strings from binary code, and then analyzes the strings to obtain the signatures for malware identification. While being very effective to detect known malware, signature-based methods are useless against unknown malicious code [21]. Recently, the basic signature-based method is extended into heuristic-based method by employing classification algorithms. In these extended methods, the executable files are represented in the form of sequences such as byte sequence and code sequence. Then the classifiers are applied to recognize patterns in the sequences in order to classify the new executable as malicious or benign [19]. Recent studies show that when using n-grams bytes to represent the binary file features, more accurate classification results can be obtained.

Operation Code (OpCode) is the portion of a machine level instruction that specifies the operation to perform[1], and as a predictor for malware [4], it is a widely studied and used representation method. Since it is obtained by disassembling the inspected executable files, a majority of current malware detection framework [1] embed third-party disassembling tools, like IDA Pro [30], W32DASM [41], to extract OpCode. To generate accurate OpCode, these tools often work conservatively to alleviate false positive. For example, in the disassembling process of IDA Pro, recursive descent algorithm [4], [13], [38]–[40], [42] is used to take care of the control flow behavior of a program. It may go around the instruction stream, and jump to a new address whenever it encounters a branch or function call. When there are a lot of such jmp instructions, it may nearly enter dead circulation, spend a very long time (say 1 day or longer) to finish, or finally quit and leave some intermediate data files. For example, in Table II of Section IV-B1, we input 30,859 sample files to drive IDA Pro with the default configuration, and only 14,612 (about 47.38%) of them can be successfully disassembled with meaningful results. The malware detection framework mentioned above are data-driven machine learning systems, and the *quality of service* provided by the disassembler is no doubt of great importance to them. However, the *quantity of training* is also a dominant impact factor to the performance of such systems. We cannot help asking the following question. Since the disassembly problem is complex and we are limited by the capability of state-of-art technologies, what if *we increase the number of successfully disassembled data by sacrificing disassembling accuracy marginally*? Will it finally improve the performance of malware detection or not?

In this paper, we propose to increase the success rate of disassembly by building a lightweight disassembler D-light, with a view to increasing the number of successfully disassembled samples to improve OpCode-driven malware detection. We implement our tool D-light, which adopts a simple disassembly algorithm, linear sweep, to alleviate the problems existing in the recursive descent algorithm of IDA Pro. In particular, we translate the binary code into assembler code directly. Although D-light reduces the accuracy of results a little bit, it is much more efficient in successfully generating disassembly results. We carry out an empirical study to compare D-light with IDA Pro. The empirical results show that we can handle more executable files and obtain higher disassembly success rate (about 87.16%). Given a fixed number of executable files, D-light uses less time to perform disassembling and training. Furthermore, it also increases the classification accuracy from 48.40% to 93.36% due to more effective disassembling.

The main contribution of our work is threefold. First, we confirmed that not only the quality of service provided by the disassemblers but also the quantity of training in the machine-learning process has a significant impact on the effectiveness of malware detection tools. Second, we proposed a lightweight disassembler D-light to extract OpCode for training classifier. It is much more efficient and has higher (successful) disassembly rate than IDA Pro. Third, we conducted an experiment on a common data set to compare the effectiveness and efficiency of using D-light and IDA Pro to support malware detection and reported optimal configuration for the tools.

The rest of the paper is organized as follows. Section II uses some unexpected observations to motivate this work. Section III presents our approach, which is evaluated in Section IV. Section V introduces relevant studies on malware detection. Finally, we conclude our work in Section VI.

II. Motivation

IDA Pro is recently the world's smartest and most feature-full disassembler [30]. It has been applied to many existing projects [4], [17], [38], [39]. IDA Pro can automatically analyze executable files, and disassemble them with the most appropriate configuration. However, it is not perfect to handle all kinds of realistic files due to the complexity of disassembly. In this section, we report such findings and inspire our work.

A. Unexpected Observations

IDA Pro can only disassemble the program code that it can identify [30]. Those unidentified file format will be regarded as binary blocks. After processing an executable as binary blocks, the disassembled result is very different from real assembly code.[2] According to our experiences of using IDA Pro, such failures are very common.

For example, in Table II of Section IV-B1, about 52.62% of the input executable files cannot be successfully disassembled by IDA Pro. Among these files, we randomly choose one called *Microsoft.Vbe.Interop.dll*, which is a win32 dll from the 32 bit Windows XP. The tool IDA Pro cannot analyze the file as expected. It reports the error "decode row: Out of

[1]A complete machine level instruction contains OpCode and, optionally, the specification of one or more operands. The operations of an OpCode may include arithmetic, data manipulation, logical operations, and program control.

[2]Browsing through the disassembled results of such "binary blocks", we found that most of them only contain one kind of operation code, namely, *db*, which is a storage definition operation defining one storage byte.

TABLE I
EXAMPLE OF UNSUCCESSFUL DISASSEMBLY BY IDA PRO

1	`void Fun(void* p) {`	`00401070 push ebp`
2	`  __asm {`	`00401071 mov ebp, esp`
3	`    add p, 3`	`00401073 sub esp, OCCh`
4	`    push p`	`00401079 push ebx`
5	`    pop eax`	`0040107A push esi`
6	`    mov esp, ebp`	`0040107B push edi`
7	`    pop ebp`	`0040107C lea edi, [ebp-OCCh]`
8	`    push eax`	`00401082 mov ecx, 33h ; '3'`
9	`    ret 4 } }`	`00401087 mov eax, OCCCCCCCCh`
10	`int _tmain( ) {`	`0040108C rep stosd`
11	`  int i = 0;`	`0040108E mov dword ptr [ebp-8], 0`
12	`  __asm {`	`00401095 push offset $yyy$3`
13	`    push yyy`	`0040109A call ?Fun@@VAXPAXeZ`
14	`    call Fun`	`0040109A ; ----------------------`
15	`    _emit 0xE8`	`0040109F db 0E8h`
16	`  yyy:`	`004010A0 +yyy dd 33C58BE8h, 5B5...`
17	`    _emit 0xE8`	`004010A0 + ; DATA XREF: _wmain+25`
18	`    mov eax, ebp }`	`004010A0 + dd 5DE58B00h, 0CCCCC...`
19	`    ...`	`004010A0 +_wmain; sp-analysis fail`
20	`  return 0; }`	`004010A0 +`

bounds: 485 (table has 485 rows). Error at GetParamProps code 0x1". We read the assemble code at the error point in the file and show the error using a short code excerpt. Table I shows the excerpt of sample code demonstrating the problem.[3] In this code excerpt, we address a pointer "`p`" (line 3) to the address of statement "`return 0`" (line 20). Then we push the pointer "`p`" and pop it to the register "`eax`". Finally, we push the register "`eax`" to the head of stack. When the program execution reaches the statement "`ret`" (line 9), the program will return to the address of "`return 0`" (line 30). We compile the code using Microsoft Visual Studio 2012 on Windows 10, and disassemble the executable file using IDA Pro. Then we obtain the result as table I.

In the table, we can find that there is an error *'sp-analysis fail'* at address *.text:004010A0* (the line in a frame). It means that some code destroyed the balance of call stack and confused the tool, causing the disassembling to fail. When we dive into the mechanism of the tool IDA Pro, we locate the problem in IDA Pro's disassembly algorithm.

B. Understanding the Problem

Algorithm 1 shows the main part of IDA Pro's disassembly algorithm — recursive descent disassembly. In the process of recursive descent disassembling, it checks whether an instruction code is a branch or function call (line 6). According to the result of checking, the process may jump into each possible target address of the instruction code and start a recursive process (line 8), until all branches and function calls have been explored (line 3). This approach seems perfect in a conservative manner since it processes all targets of each jump operation including direct jump, indirect jump, making sure that all reachable codes are disassembled. Unfortunately, the key assumption of the algorithm that "the set of control flow successors of each control transfer operation in the program can always be correctly identified" is too strong in practice.

For example, not all indirect jumps are through jump tables, and checking the bounds of all jump tables[4] can be either

[3]All irrelevant details in the sample have been removed from Table I.

[4]A jump table refers to an array of address that is commonly used to implement multi-way control transfers.

Algorithm 1 Recursive Descent Disassembly Algorithm [37]

```
Input: $addr, disStrList$
Output: $disStrList$
 1: function RcrsvDscntDsPrc($addr, disStrList$)
 2:     if not $addr.visited$ then
 3:         while $addr$ is valid do
 4:             $disStr = Disassemble(addr)$
 5:             push $disStr$ into $disStrList$
 6:             if $disStr$ is a branch or function call then
 7:                 for each possible target $tAddr$ of $disStr$ do
 8:                     RcrsvDscntDsPrc($tAddr, disStrList$)
 9:                 end for
10:             else
11:                 $addr = addr + length(disStr)$
12:             end if
13:             $addr.visited = True$
14:         end while
15:     end if
16: end function
```

unnecessary or infeasible in practice. Take the example in Table I to illustrate. Because we change the return address of the function "`void Fun(void* p)`", the algorithm cannot statically predict the correct target address. When it jumps to the incorrect target address, it continue to check that block as a call stack and immediately fails since the "call stack" seems problematic.

In such a case, imprecisely identifying the set of possible target address of a jump operation will result in a failure in disassembling. In other words, the complex analysis process and uncertainty existing in possible target address increase the risk of unsuccessful disassembling.

C. Our Idea

Let us recall the goal of malware detection, which is to train a detector or classifier for an explored executable file. We know that machine learning algorithms are data driven and the disassembler used to generate input data is crucial. On the other hand, both the quality of service provided by the disassembler and the quantity of training determine the effectiveness of malware detection. We thus consider the problem that "*can we finally improve the performance of malware detection if we enhance the training step by increasing the amount of disassembled code while sacrificing the disassembly accuracy a little bit?*"

A straightforward idea is to adopt a more aggressive algorithm to disassemble executable files. We foresee the main benefit is to have higher success rate in disassembling. However, we also realize the challenge that aggressive disassembling algorithm may come with inaccurate disassembly results. In the next section, we will introduce our method and give further discussion.

III. OpCode-Driven Malware Detection

In this section, we will revisit the current mainstream malware detection framework, and elaborate on our proposal.

A. IDA Pro and Current Popular Framework

Taking the popular tool IDA Pro as example, the framework of malware detection is shown in Figure 1. Generally, the pro-

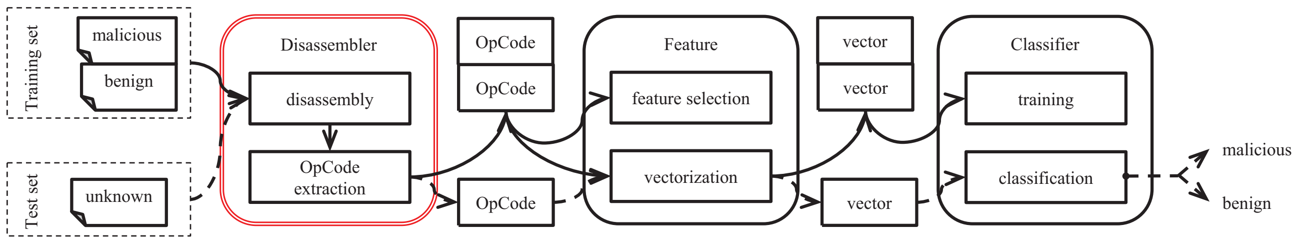


Fig. 1. Workflow of Current OpCode-Based Malware Detection Framework

cess of classifying an unexplored executable file as malicious or benign is divided into two phases: training and test.

As showing in Fig. 1, during the training phase (all arrows in solid lines), executable files including malicious files and benign files are input into the system. Each file needs to be disassembled and OpCode sequence is extracted from it. The OpCode sequences extracted from the files in the training set and their labels (namely, malicious and benign) are the inputs of the training algorithm.[5] The disassembler is marked as red rectangle in the figure. After completing the training, we obtain a detector, also called a classifier, which can determine whether an unexplored executable file is malicious or benign.

Next, in the test phase (all arrows in dashed lines), a set of files as the test set is classified by the trained classifier. Certainly, the files in the test set also need be disassembled (and OpCode sequences need to be extracted from them) before they are input into the classifier. Based on the trained model, the classifier will determine a file as malicious or benign. More details about the training and test framework will be introduced in the subsection III-C.

B. Our Proposal

We have shown in Section II-B that IDA Pro works in a conservative manner to alleviate inaccurate disassembling. As a result, it reports error and stops disassembling, mostly in the cases where there are many `jmp` instructions.

The goal of our present work is to obtain a detector or classifier for unexplored executable files with good effectiveness, including high accuracy, low false positive rate (FP), and so on. To accomplish the goal, we attempt to provide more training samples in order to get a more effective classifier. If we can disassemble more files in an effective way, we will obtain more data for training, thus improving detection performance. Given this assumption, we propose to use another disassembly technology, namely, the linear sweep disassembly algorithm.

We have highlighted in Section II-B that the recursive descent algorithm takes into account the control flow behavior of the program and tries to get around every piece of data linked in the instruction stream by branch or calls. Generally, recursive descent and linear sweep are two commonly used disassembler technologies [37]. It is believed that recursive descent provides more accurate disassembling than linear sweep, although the latter is more efficient. However, we revisit the problem and raise the supposal that "***it is quantity of training rather than quality of service provided by the disassembler that impacts the performance of classification-based malware detection framework dominantly***". As a result, we venture to abandon the popular recursive descent manner and propose to use a lightweight candidate — linear sweep.

[5]Each OpCode will be vectorized according to selected representative features. We do not go deep into this step here since it is the standard machine learning methodology.

C. Framework Details

In this section, we will present our training and test framework as shown in Fig 1, as well as how to select representative OpCode as features and how to transform OpCode files into vectors for training.

1) Basic Pattern: To understand the meaning of one word, it is important to understand its context first. To train the classifier, we use n-gram [6] OpCode as the basic pattern. An n-gram is a contiguous sequence of n items from a given sequence of text or speech. The value of n is called the size of the n-gram. The n-gram model is one type of probabilistic language model for predicting the next item in such a sequence in the form of an $(n-1)$ order Markov model. It is widely used in probability, communication theory, computational linguistics (such as statistical natural language processing), computational biology (such as biological sequence analysis), and data compression. In our experiments, we identified 933 1-gram sequences, 92464 2-gram sequences, 2281065 3-gram sequences, 24700730 4-gram sequences, and 99926702 5-gram sequences. It is difficult to narrow down on the basic classes of n-grams for the empirical study. If we choose a small n, the number of features will not be excessive and the training will be easy. But the accuracy can be also low. In contrast, if we choose a large n, the number of features will be huge and the training will be difficult. Finally, based on the suggestions from existing work [39] and our problem, we choose to use 2-gram OpCode sequences as the basic pattern in this work.

2) Feature Selection: To train a classifier for malware detection, we have to vectorize disassembled executable files such that a machine learning model can understand.

However, when we finish disassembling those files and obtain the OpCode, we find that there are too many statements. If we choose every OpCode as a feature, the large number of features presents a significant problem because many of them do not help improve the accuracy and even may decrease it [39]. Meanwhile, it is of importance to identify the terms that appear in most of the files in order to avoid vectors containing too many zeros, which will affect performance. So before

Algorithm 2 Linear Sweep Disassembly Algorithm [37]

```
Input: $addr$, $startAddr$, $endAddr$, $disStrList$
Output: $disStrList$
1: function LnrSwpDsPrc($addr, disStrList$)
2:     while $startAddr \leq addr \leq endAddr$ do
3:         $disStr = Disassemble(addr)$
4:         push $disStr$ into $disStrList$
5:         $addr = addr + length(disStr)$
6:     end while
7: end function
```

converting file into vectorial representation, we need to select some common and frequent terms as *features* for vectorization, as to improve the effectiveness and reduce the training time.

In the process of selecting features, a measure is used to quantify the correlation of each feature to the class, and to estimate its expected contribution to the right classification [20]. The measure used in feature selection is independent of all kinds of classification algorithms, so that comparison of different classification algorithms can be fair. The document frequency measure (DF) [33], information gain ratio (GR) [14] or Fisher Score (FS) [11] are all typical such measures. In this paper, similar to [6], [39], we use the simple but effective measure DF to select top 300 highest terms as features.

3) Vectorization: In text categorization field, TF and TFIDF are well known measures [33]. In this paper, each 2-gram OpCode are regarded as one term. While we generate 2-gram OpCode sequences, we count the times of each term (2-gram OpCode) appearing in one file as TF, and the number of files containing the term as DF. In order to represent all samples fairly instead of favoring long files, we normalize TF as normalized-TF by dividing the frequency of the term in the file by the biggest frequency in the same document.

After acquiring features mentioned above, we use the selected set of terms to vectorize each file. We use the normalized-TF as weight of each term in features. The features are used to vector samples not only in the *training set*, but also in the *test set*. In other words, when we process an unexplored file, we vectorize it with the same features after disassembling and extracting 2-gram OpCode sequences from it, too.

4) Training and Classification: With thus represented profiles, now we can train a classifying model to process unexplored executable files. Referring to the results in [39] and considering in accessibility, we choose Support Vector Machine (SVM) [3] as our classifying algorithm.

D. Implementation Details

The basic algorithm of linear sweep is showing in Algorithm 2. Different from the recursive descent disassembly algorithm, linear sweep begins disassembling from the first executable byte of the input file (line 1), and simply sweeps through the entire text section disassembling each instruction as it is encountered (line 5). Its main advantage is simplicity. Especially, there is no need to buffer intermediate results (so no need of recursion), the input stream can be disassembled with one pass, and each instruction string can be directly discarded after processing. Among the many implementations of linear sweep, we stick to existing work [7], [30] to design our own tool, which can handle all kinds of executable files without the limit of operating system. We name our tool as *D-light* to hight that it is based on ***light***weight ***D***isassembler. Besides the substitution of main algorithm, we also made other optimizations when implementing our tool.

Code obfuscation is an important technique used by attackers in order to avoid detection by security mechanisms, such anti-virus software, IDS (Intrusion Detection System), IPS (Intrusion Prevention System), and so on [18]. These obfuscation techniques are also used on benign software for copyrights protection purposes. Security experts often spend a lot of time dumping the files from memory and manually unpacking them. In our experiences, we would like to point out that a large number of obfuscation techniques are not used by attackers, because attackers want the malicious files to appear benign. So that malware can pass security mechanisms, which are deployed to block content that is encrypted or obfuscated against inspection [38]. In our method, we do not unpack any input data, but disassemble them directly and let classifiers learn from the disassembled code.

Finally, the code excerpt in Table I is successfully disassembled by our method. However, two complicated instructions are not recognized and misunderstood in the disassembled result since D-light has no transversal process to understand them. We judge that it will have no adverse effect on the accuracy of the malware detection system. The reasoning is that in so long as the disassembler generates stable results, any correct or incorrect instruction sequence will be remembered by the machine-learning-enabled classifier. In the next section, we will evaluate our tool and validate that.

IV. Evaluation

In this section, we raise research questions, introduce experiment setup steps, and report empirical observation in comparing the performance of IDA Pro and D-light.

A. Research Questions

Regarding our proposal, we raise the following research questions. They will be answered by comparing the performance of IDA Pro and D-light.

Q1: Which disassembly algorithm has better performance in disassembling?

Q2: Which disassembly algorithm has better performance in supporting malware detection in the current framework?

Q3: How do quality of service and quantity of training impact the performance of detecting malware?

B. Experiment Setup

1) Subject Samples: We refer to existing work [38], [39], [42] to select subject programs in our experiment. We obtain 35,611 samples in total, consisting of 24,865 malicious files from VX Heaven [12], and 10,746 benign files from Windows XP operating system. After picking out the executable file, deleting duplicated ones, and passing anti-virus checks, we obtain the final subject set, which includes 30,859 files (20,113 malicious and 10,746 benign). Table II shows the statistics.

TABLE II
DESCRIPTIVE STATISTICS OF SUBJECTS

File Format	Count
COM executable for DOS	2123
DOS executable (COM)	4770
MS-DOS executable, for MS Windows	5566
PE32 executable (console) Intel 80386, for MS Windows	1144
PE32 executable (DLL) Intel 80386, for MS Windows	7927
PE32 executable (GUI) Intel 80386, for MS Windows	3475
PE32 executable (native) Intel 80386, for MS Windows	384
PE32+ executable (console) Intel Itanium, for MS Windows	136
PE32+ executable (console) x86-64, for MS Windows	45
PE32+ executable (DLL) x86-64, for MS Windows	5179
PE32+ executable (native) x86-64, for MS Windows	42
PE32+ executable (GUI) x86-64, for MS Windows	41
Linux-Dev86 executable, headerless	27
Malicious samples	20,113
Benign samples	10,746
Total	30,859

2) Target Disassembly Algorithms: In this experiment, we process each subject file using two disassemblers, namely, IDA Pro and D-light.

The implementation of IDA Pro is directly taken from its official site. By default, IDA Pro does not export assembler code. We write *idc* scripts[6] and python scripts to drive it. We manage to invoke IDA Pro in command-line manner and interact with it in batch mode to accelerate the experiment. In such a way, we can also disassemble files concurrently. To compare with IDA Pro, we implement the algorithm of D-light, which details have been introduced in Section III-B.

To compare the performance of the two tools in supporting malware detection, we embed them separately in the current malware detection framework.

3) Settings of the Malware Detection Framework: After obtaining OpCode data from IDA Pro and D-light, we organize them to form training set and test set, and input them to a standard malware detection framework. In this experiment, we follow existing work [39] to specify the configuration.

As showing in Table III, we extract 2-gram OpCode sequences. In the feature selection phase, we pick the top 300 2-gram OpCode sequences as features according to their document frequency (DF). After the features are determined, we use the normalized term frequency (TF) for the features to vectorize each sample. In the training phase, we choose support vector machine (SVM) to train a classifier. Further, we use a polynomial kernel function for the SVM algorithm [3].

4) Performance Metrics: To measure the performance of classification, we refer to standard accuracy measurements.

First, we need to know how many (or what percentage of) files can be detected correctly, including both malicious and benign ones. Commonly, it is measured by the *Accuracy* metrics in Machine Leaning. To do that, we use *True Positive* (*TP*) to represent the number of recognized positive samples, *False Positive* (*FP*) to represent the number of unrecognized negative samples, *True Negative* (*TN*) to represent the number of recognized negative samples, and *False Negative* (*FN*) to represent the number of unrecognized positive samples. At

[6]A powerful C-like embedded programming language extending IDA.

TABLE III
CONFIGURATION AND VARIABLES OF THE FRAMEWORK

	Configuration
Size of n-grams	2
No. of features	300
Cross-validation	10 fold
Model	support vector machine (SVM)
Feature selection	document frequency (DF)
Vectorization	Normalized-TF
Kernel functions	polynomial

the same time, if a sample cannot be classified into any group, it is marked to be the opposite class. For example, if a benign executable file cannot be disassembled successfully by IDA Pro, we mark its class as malicious (False Negative, accordingly). The *Accuracy* metrics is given as follows.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \times 100\% \quad (1)$$

It stands for how many percentage of files (both malware and benign ones) are correctly recognized (detected). The larger the value of *Accuracy*, the more accurate will be the classifier.

Additionally, we also care about the error rate, which is the number of malicious files classified as benign or vice versa. Similarly, in Machine Learning, the *True Positive Rate* (*TPR*) metrics is used to measure the percentage of positive instances classified correctly, and the *False Positive Rate* (*FPR*) metrics measures the percentage of negative instances misclassified. We use these two factors to estimate error rate, which are calculated as follows.

$$TPR = \frac{TP}{TP + FN} \times 100\%, \quad FPR = \frac{FP}{FP + TN} \times 100\% \quad (2)$$

The larger the value of *TFP*, the smaller will be the error rate of the classifier. On the contrary, the smaller the value *FPR*, the smaller will be the error rate of the classifier.

In order to realize the negative effects of data, we also introduce another measure, *G-mean*, which is usually used in the data imbalance problem, to quantify the performance issue [2]. *G-mean* is calculated as follows.

$$G\text{-}mean = \sqrt{\frac{TP}{TP + FN} \times \frac{TN}{FP + TN}} \quad (3)$$

We may simply interpret that the bigger the value of *G-mean*, the more stable will be the classifier.

5) Experiment Environment: We implement the experiments using python Scikit-learn [26] on Linux Ubuntu 14.04 system. All the representative vector profiles for the subjects are stored in local text files. In order to avoid accidents such as program crashes, we also write the intermediate results into local files in the training phase. Besides, we dump the trained model into a local file for reuse. During the training phase, we record the time consumed from loading data to generate trained model. Since the time taken to detect a new sample is very short, we do not take it into consideration.

C. Experiment Result

We organize three tests to answer the research questions.

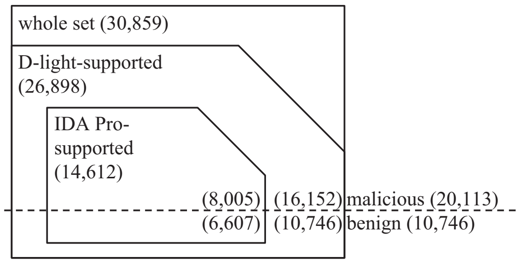
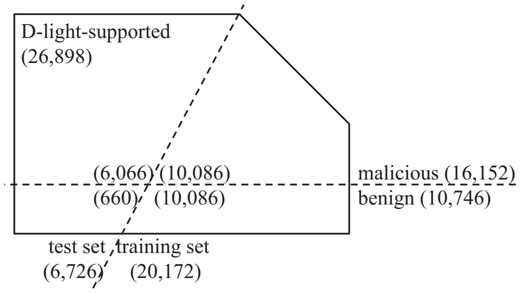
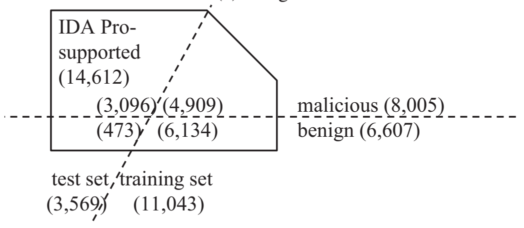


Fig. 2. Result of Test One



(a) D-light



(b) IDA Pro

Fig. 3. Data Preparation for Test Two

1) Test One: Table 2 shows the experiment results in disassembling. Among the 30,859 subject files, 26,898 (87.16%) of them can be successfully disassembled by D-light. In particular, 16,152 (80.31%) of the 20,113 malicious files and 10,746 (100%) of the 10,746 benign files are successfully disassembled. At the same time, we could only obtain 14,612 (47.38%) OpCode files by using IDA Pro, which consists of 8,005 (39.80%) malicious files and 6,607 (61.48%) benign files. Further, we notice that all the samples, which can be disassembled by IDA Pro, can be disassembled by D-light, but not vice versa.

Meanwhile, D-light uses 2 hours to process all the files, while IDA Pro uses up to 48 hours to finish. From the results in Fig. 2, we can answer research question Q1 as follows.

A1: D-light is more efficient and comes with higher success rate in disassembling, while IDA Pro outputs more accurate results.

The second part is well known and also observed in the experiment — at least 12,286 (26,898−14,612) results of D-light are actually not accurate since they are not processable to the more capable disassembler IDA Pro. So we continue the next test to know how they support malware detection.

2) Test Two: Fig. 3 shows the data preparation for this test. From the 26,898 files (recalling Fig. 2) that can be disassembled by D-light, we randomly select 10,086 malicious files and 10,086 benign files to construct a training set and the rest (6,066 malicious and 660 benign files) files to construct a test set, for D-light. We do such arrangements in order to: (i) let the size of the training set to be three times that of the test set[7], and (ii) let the training set to contain identical number of malicious samples and benign samples. The former is to simulate a scenario where we have many samples for training followed by some unexplored files to process. The latter is to follow the basic rule in machine learning to keep the balance of different classes to alleviate data impact. The same training set and test set are inputted to IDA Pro (as shown in Fig. 3). However, only 4,909 malicious samples and 6,134 benign samples in the training set can be disassembled by IDA Pro. Only 3,096 malicious samples and 473 benign samples in the test set can be disassembled. The impacts will be discussed later in Section IV-C3.

TABLE IV
RESULT OF TEST TWO

	Accuracy	*TPR*	*FPR*	*G-mean*	**Time Taken**
D-light	93.44	95.15	6.74	0.9420	2 minutes
IDA Pro	48.26	61.97	53.23	0.5384	2 minutes

Table IV shows the classification results of D-light and IDA Pro, driven by the data sets in Fig. 3. Taking the first row to illustrate, in the 6,726 samples, D-light could correctly classify 6,285 of them, and the *Accuracy* is 93.44%. The error rate can be measured as 95.15% in *TPR* and 6.74% in *FPR*. By contrasting the performance results of D-light and IDA Pro, we find that D-light consistently and observably outperforms IDA Pro. For example, the result in *Accuracy* by D-light is 93.44%, while that by IDA Pro is only 48.26%. By using *TPR*, *FPR*, and *G-mean* to evaluate, D-light also outperforms IDA Pro (95.15% to 61.97%, 6.74% to 53.23%, and 0.9420 to 0.5384, respectively). As a result, we answer the research question Q2 as follows.

A2: The framework embedding the disassembler D-light detects more malware.

However, we also realize that unidentical number of training samples are used for D-light and IDA Pro in this test (see Fig. 3), which may cause unfair comparison when evaluating the accuracy of the a machine-learning system. We continue the last test to address this issue.

3) Test Three: In this test, we re-prepare the data as Table V, in three groups. Group 1 and Group 3 are exactly what we did in Test Two. We revisit them as follows. Group 3 is the test with D-light, where the training set is given as in Fig. 3. Group 1 is the test with IDA Pro, where the training set is as that in Group 3 but filtering out those samples that cannot be successfully disassembled by IDA Pro. Group 2 is a test with D-light, where the training set is as that in Group 1. Since the samples of the training set inputted to D-light in

[7]The ratio between training sets and test sets generally ranges from 2/3 to 4/5 in existing work.

TABLE V
DATA PREPARATION FOR TEST THREE

	Group 1 IDA Pro	**Group 2** D-light (limited)	**Group 3** D-light
Training set (malicious)	4,909	4,909	10,086
Training set (benign)	6,134	6,134	10,086
Subtotal	11,043	11,043	20,172
Test set (malicious)	6,066	6,066	6,066
Test set (benign)	660	660	660
Subtotal	6,726	6,726	6,726

Group 2 is intentionally limited for a comparison purpose, we use "D-light (limited)" to name this group.

We design such as to simulate the scenario that: (i) A training set is given, (ii) Not all samples can be disassembled by IDA Pro, so only the supported ones are used to test IDA Pro, that is Group 1, (iii) We apply the same sets (as Group 1) to test D-light, that is Group 2, and (iv) Since the capability of D-light is limited in Group 2, we further input the rest of it to test D-light, that is Group 3.

The results of the three groups are showed in Fig. 4. In this figure, the x-axis represents four different kinds of metrics used to measure the performance of the two classifiers. The y-axis represents the result of each concrete metric, and the range is from 0 to 100 (%). Four groups of columns in order show effectiveness results in *Accuracy*, *TPR*, *FPR*, and *G-mean*. In particular, the columns in white represent results for IDA Pro (Group 1), the columns in black represent results for D-light (Group 3), and the columns in brick pattern represent results for D-light (limited) (Group 2). Take the four columns in brick pattern to illustrate. They show the values in the four metrics, that is, 91.41% in *Accuracy*, 93.33% in *TPR*, 8.80% in *FPR*, and 0.9226 in *G-mean*, of D-light (limited) in Group 2. Recalling the meaning of the four metrics, this figure intuitively shows that D-light has better effectiveness in detecting malware than IDA Pro, which is consistent with what was stated in Table IV.

This figure also indicates that: (1) by limiting the size of the training set, the accuracy of D-light decreases marginally (from Group 3 to Group 2), and (2) given training sets of the same size, the accuracy of D-light is observably better than that of IDA Pro (Group 2 to Group 1). From such observations,

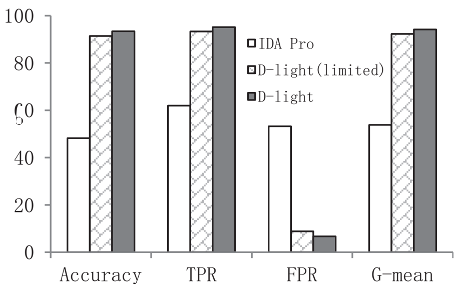


Fig. 4. Result of Test Three

we answer the research question Q3 as follows.

A3 The quantity of training has a more observable impact on OpCode-driven malware detection framework than the quality of service provided by disassembler.

D. Threats to Validity

IDA Pro needs to load a lot of plugins to analyze executable file at launch phase, and it also takes a lot of time to analyze samples and generate result files. Even when it is invoked in a command-line manner, it is still very time-consuming. In our experiments, one executable file called *import10.dll* from 32 bit Windows XP spent more than 24 hours to disassemble and finally fails. For such reasons, we have removed from our experiment all the samples that need more than 48 hours to disassemble. An experiment using more capable computing units may result in different results.

With IDA Pro, we find that most of the executable files failing in disassembling are treated as binary file. For those files, no operation code rather than *db*, which is a storage operation code defining one storage byte, is recognized. IDA Pro only generated some incomplete intermediate files (with suffixes like *.id0*, *.id1*, and *.til*) and gave up the disassembling task early. Exceptions are raised in the process, which include too many jump, identifying incorrect target address, and so on. We do not exclude the possibility that IDA Pro could be hacked to work well, by given particular configurations.

Although we refer to existing work [38], [39], [42] to select subject programs in our experiment, the dataset is a little out of date. It may be a small influence on the evaluation.

Our experiment is carried out with unpacked code, which also causes threats to the validity of the empirical observations. Normally, some off-the-shelf tools, like *UPX*, *PEcompact2*, and *Armadillo*, are used for packing and compressing software. In such a case, traditional static analysis methods may fail to obtain the correct result without unpacking a packed file [8]. Various proposals have been given to address this challenge [8], [27], [32]. However, for some new packing and compressing techniques, it may be hard to find an existing tool to unpack. Sometimes, it is also challenging to identify the correct packer or to unpack a file correctly.

We have discussed the impact of having different number of training samples to the comparison of malware detecting accuracies in Section IV-C3. At the same time, IDA Pro cannot disassemble some samples of the test set in the experiment. Samples unsuccessfully disassembled cannot be correctly classified, and are accordingly marked as either False Negative (for benign samples) or False Positive (for malicious samples). Not counting them on will affect the observations on empirical results. However, we insist that they are not ignorable in evaluating the performance of a malware detection system.

V. RELATED WORK

A. Bytecode-Driven Malware Detection

Over the past decade, several studies focused on the detection of unknown malware based on the binary code content of targets. Schultz et al. [36] firstly proposed to apply Machine

Learning (ML) technologies to detect different malware based on their respective binary codes. In their work, three different feature extraction methods are employed – the features extracted from the Portable Executable (PE) section, meaningful plain-text strings that are encoded in program files, and byte sequence features of the target files.

Abou-Assaleh et al. [1] introduced a framework that uses the common n-gram (CNG) method and the k-nearest neighbor (KNN) classifier for the detection of malware. This framework used a profile to represent malicious file and benign file. When an unexplored executable file is processed, it will first be represented by the profile, and then compared with the profiles of malicious and benign classes. Finally, it is classified to be of the most similar class. Cai et al. [5] conducted several experiments, in which they evaluated the combinations of seven feature selection methods, three classifiers, and n-grams of various sizes. Moskovitch et al. [22] present the results of a study using a test sample containing more than 30,000 files represented by the n-gram bytecode format.

Our proposal in this paper is different from the above techniques in that it focuses on Op-Code rather than bytecode as input to the machine-learning malware detection framework.

B. OpCode-Driven Malware Detection

Bilar [4] examines the difference of statistical OpCode frequency distribution in malicious and benign code. The statistics of total 67 malicious executables were compared with that of the 20 benign samples. The results showed that malicious software OpCode distributions differ significantly from those of the benign software. Therefore, the work concluded that OpCode is a good predictor for malware.

Data mining methods (such as logistic regression, artificial neural networks, and decision trees) are used in [40] to automatically identify critical instruction sequences that can distinguish malicious from benign programs. Karim et al. [13] proposed to track malware evolution based on OpCode sequences and permutations. The evaluation results showed a high accuracy rate of 98.4%. Similar to bytecode-driven framework, n-gram methodologies are also employed in OpCode-driven malware detection. Shabtai et al. [39] used n-grams of various sizes while Bilar [4] used only 1-gram. Based on their experiments, using OpCode sequences can improve the malware detection performance significantly. Furthermore, they summarized a set of good parameters to train the classifier for analyzing new executable. In this paper, we use a similar configuration to train the models. Santos et al. [34] used multiple OpCode representations (such as 1-grams and 2-grams) to classify malware instances by measuring the similarity between files. This is, however, different from our goal to classify unexplored suspicious files as malicious or benign.

Santos et al. [35] proposed a technique to detect unknown malware. Their model was based on the frequency of the OpCode sequences. In their results, the technique was capable of detecting unknown malware effectively. We are also interested in integrating our method with such approaches.

C. OpCode Extraction

Broadly, there are two disassembly approaches, namely static disassembly and dynamic disassembly. Static disassembly, is a method without executing the file during the process, and dynamic disassembly will execute the file on some input. Static disassembly can load the entire file at one time, while dynamic disassembly only can handle a part of the executable file. More importantly, the time complexity of static disassembly is approximately proportional to the size of the program, while the time taken by dynamic disassembly is typically proportional to the number of instructions executed by the file at runtime [18]. In this paper, we focus on the static disassembly to extract OpCode patterns.

Recursive traversal and linear sweep [37] are two generally used techniques for static disassembly. Linear sweep is a straightforward approach to decode everything appearing in sections of the executable that are typically reserved for machine code. It is prone to disassembly errors resulting from the misinterpretation of data that is embedded in the instruction stream, because it is the logic of linear sweep that does not take into account the control flow behavior of the program. Recursive traversal is more sensitive to jmp instruments and the validation of addresses. With branch condition taken in, the modified disassembly algorithm is called recursive descent [18], [43]. In this paper, we argue and compare the use of them to support state-of-the-art malware detection.

Additionally, there are also many studies to use not OpCode but other information such as control flow graph [24], api call [15], system call [25], and so on.

VI. Conclusion

Machine learning methods are used to classify unknown malware. The IDA Pro disassembler is extensively used in de facto state-of-the-art frameworks of such malware detection approaches. At the same time, the performance of such systems is inevitably affected by at least two factors, namely the quality of service provided by disassembler and quantity of training in the machine-learning process. We noticed that to generate accurate disassembled code, IDA Pro employs the recursive descent algorithm, which is both capable to track complicated program execution paths and also conservative to deny service when suspicious code section is reached. As a result, the malware detection framework could be short of training samples, causing low detection accuracy.

In this paper, we implemented our tool D-light to use a simple disassembly method, linear sweep, to address this issue with the recursive traversal manner adopted by IDA Pro. Our idea is to provide many marginally defective samples rather than few well disassembled samples to the OpCode-driven malware detection framework. We predict that the quantity of training plays a more important role than the quality of service provided by the disassembler, in determining the final performance of such a framework. The experiments reported that D-light outperforms IDA Pro empirically, which validates our proposal and confirms our supposal.

In the future, we will carry out experiments with other algorithms. We are also looking into classification methods used in other fields, like image recognition and speech recognition, to inspect similar problems.

References

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, vol. 2. IEEE, 2004, pp. 41–42.

[2] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[3] S.-i. Amari and S. Wu, "Improving support vector machine classifiers by modifying kernel functions," *Neural Networks*, vol. 12, no. 6, pp. 783–789, 1999.

[4] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, 2007.

[5] D.M. Cai, M. Gokhale, and J. Theiler, "Comparison of feature selection and classification algorithms in identifying malicious executables," *Computational statistics & data analysis*, vol. 51, no. 6, pp. 3156–3172, 2007.

[6] W.B. Cavnar, J.M. Trenkle *et al.*, "N-gram-based text categorization," *Ann Arbor MI*, vol. 48113, no. 2, pp. 161–175, 1994.

[7] S.P. Dandamudi, "Installing and using nasm," *Guide to Assembly Language Programming in Linux*, pp. 153–166, 2005.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62.

[9] M. Fossi, D. Turner, E. Johnson, T. Mack, T. Adams, J. Blackbird, S. Entwisle, B. Graveland, D. McKinney, J. Mulcahy *et al.*, "Symantec global internet security threat report," *White Paper, Symantec Enterprise Security*, vol. 1, 2010.

[10] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh, "Automatic generation of string signatures for malware detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2009, pp. 101–120.

[11] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," *arXiv preprint arXiv:1202.3725*, 2012.

[12] V. Heaven, "Computer virus collection," 2014. [Online]. Available: http://vxheaven.org/vl.php

[13] M.E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, pp. 13–23, 2005.

[14] J.T. Kent, "Information gain and a general measure of correlation," *Biometrika*, vol. 70, no. 1, pp. 163–173, 1983.

[15] H. Kim, J. Kim, Y. Kim, I. Kim, K.J. Kim, and H. Kim, "Improvement of malware detection and classification using API call sequence alignment and visualization," *Cluster Computing*, Springer, 2017, pp. 1–9.

[16] C. Kolbitsch, P.M. Comparetti, and L. Cavedon, "Methods and systems for malware detection based on environment-dependent behavior," U.S. Patent 9,361,459, Jun. 7, 2016.

[17] J.Z. Kolter and M.A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 470–478.

[18] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 290–299.

[19] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving malware detection by applying multi-inducer ensemble," *Computational Statistics & Data Analysis*, vol. 53, no. 4, pp. 1483–1494, 2009.

[20] T.M. Mitchell, *Machine Learning*, McGraw-Hill, New York, NY, 1997.

[21] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*. IEEE, 2007, pp. 421–430.

[22] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, N. Japkowicz, and Y. Elovici, "Unknown malcode detection and the imbalance problem," *Journal in Computer Virology*, vol. 5, no. 4, pp. 295–308, 2009.

[23] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*. ACM, 2011, p. 4.

[24] M.H. Nguyen, D. Le Nguyen, X.M. Nguyen, and T.T. Quan, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning," *Computers & Security*, Elsevier, 2018.

[25] S.D. Nikolopoulos and I. Polenakis, "A graph-based model for malware detection and classification using system-call groups," *Journal of Computer Virology and Hacking Techniques*, Springer, 2017, vol. 13, no. 1, pp. 29–46.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[27] R. Perdisci, A. Lanzi, and W. Lee, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE, 2008, pp. 301–310.

[28] D. Potashnik, Y. Fledel, R. Moskovitch, Y. Elovici *et al.*, "Monitoring, analysis, and filtering system for purifying network traffic of known and unknown malicious content," *Security and Communication Networks*, vol. 4, no. 8, pp. 947–965, 2011.

[29] P. Ren, W. Liu, D. Sun, J.-p. Wu, and K. Liu, "Analysis and forensics for behavior characteristics of malware in internet," in *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*. IEEE, 2016, pp. 637–641.

[30] D. Rescue, "Ida pro disassembler and debugger," Hex-Rays SA, 2017. [Online]. Available: https://www.hex-rays.com/

[31] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 108–125.

[32] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 289–300.

[33] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[34] I. Santos, F. Brezo, J. Nieves, Y.K. Penya, B. Sanz, C. Laorden, and P.G. Bringas, "Idea: Opcode-sequence-based malware detection," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2010, pp. 35–43.

[35] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P.G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013.

[36] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo, "Data mining methods for detection of new malicious executables," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 38–49.

[37] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Reverse engineering, 2002. Proceedings. Ninth working conference on*. IEEE, 2002, pp. 45–54.

[38] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *information security technical report*, vol. 14, no. 1, pp. 16–29, 2009.

[39] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, p. 1, 2012.

[40] M. Siddiqui, M.C. Wang, and J. Lee, "Data mining methods for malware detection using instruction sequences," in *Proceedings of the 26th IASTED International Conference on Artificial Intelligence and Applications*, ser. AIA '08. ACTA Press, 2008, pp. 358–363.

[41] "W32dasm," 2017. [Online]. Available: http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/WDASM.shtml

[42] C. Wang, Z. Qin, J. Zhang, and H. Yin, "A malware variants detection methodology with an opcode based feature method and a fast density based clustering algorithm," in *Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), 2016 12th International Conference on*. IEEE, 2016, pp. 481–487.

[43] J. Wu, L. Jiang, and P. Zhao, "Research on a static disassembly algorithm based upon control flow," *Computer Engineering and Applications*, vol. 30, pp. 89–90, 2005.