Detective: Automatically Identify and Analyze Malware Processes in Forensic Scenarios via DLLs

Yiheng Duan[1], Xiao Fu[1*], Bin Luo[1], Ziqi Wang[1], Jin Shi[2]
Software Institute[1], School of Information Management[2]
Nanjing University, Nanjing, China
Email: duanyiheng@gmail.com, fuxiao@nju.edu.cn, luobin@nju.edu.cn

Xiaojiang (James) Du
Department of Computer and Information Sciences
Temple University, Philadelphia, USA
Email: dxj@ieee.org

***Abstract*—Current memory forensic methods mainly focus on evidence collection and data recovery. A little work is about how to automatically identify malwares from many unknown processes and analyze their behaviors in high semantic level so as to collect related evidences. In fact, in real cases, investigators are often faced with large number of processes that they have no knowledge of. Although current malware detection tools could provide some help, they usually can't illustrate the purposes, abilities and behavior details of malwares and are thus often not fit for the forensic requirements. In this paper, we present a framework named Detective to cope with these issues. Given a set of unknown processes, Detective can classify benign and malware processes automatically. This is implemented by HNB classifying algorithm and a Dynamic-Link Libraries-based model. Detective could then explain malware behaviors in high semantic level through clustering and frequent item sets mining techniques. Besides, Detective sheds light on evidence collection by the information obtained from previous steps. Detective is applicable for both online and offline forensic scenarios. Experiments on real-world malware set have proved that the accuracy of Detective is above 90% and the time cost is only several seconds.**

Keywords—malware processes; memory forensics; DLL; data mining

I. INTRODUCTION

Memory forensics developed greatly in past ten years, as memory can provide information disks don't contain, such as running processes, network connections and so on. Moreover, although the binary codes can be encrypted or obfuscated, all illegal processes still have to be executed in memory and leave some footprints inevitably. So evidences from memory are more reliable. Researchers in the area of malware investigation are especially interested in this kind of technology as some malwares only exist in memory. However in memory-based malware forensic cases, investigators are often faced with large number of processes they have no prior knowledge of. Understanding all these processes is a time consuming task even for skillful experts. Thus how to automatically identify malwares from unknown processes and analyze their behaviors in high semantic level so as to obtain the related evidences have become some of the key issues for malware investigators.

Current methods in the area of memory forensics mainly fall into two categories: one focuses on dumping memory completely and reliably, and the other concentrates on recovering data accurately. Little work is about automatic analyzing of data or evidences. Some powerful tools, such as Volatility[1] offer certain analyzing functions, but they are usually simple and depend highly on users' prior knowledge, making them unable to automatically identify and analyze malwares. Although current antivirus software can identify malwares with high accuracy, they are that not fit for forensics for some reasons. Firstly, they need to be installed on the target system before analysis which will cause contamination. Secondly, they can also be compromised by malwares, so the result of detection is not reliable. Thirdly, the goal of antivirus software is to find and remove the malwares, not to capture, analyze them and obtain related evidences. So the help from them for investigators is limited. Although current process analysis methods, such as some dynamic analysis approaches[6] can provide the details of processes, these methods are usually based on VMI and taint analysis technique, which need high cost and have to be performed on preinstalled specific virtual machines (e.g. QEMU). It is not practical for investigators.

Therefore, in this paper we present Detective, a novel framework to address the automated malware identification and analysis issues in memory forensic context. Detective mainly achieves three goals: classifying benign and malware processes automatically, explaining malware behaviors in high semantic level and shedding light on evidence collection. In order to identify malware processes from unknown process automatically, we propose a novel Dynamic-Link Libraries (i.e. DLLs) based malware detection technique. It is composed of a HNB classifying algorithm and a DLLs-based process model. We choose DLLs to profile processes not only because they are good indications of the real goals and behaviors of processes, but also because they can be obtained both online and offline. We also design a data mining based analysis method to explain malware behaviors in high semantic level. This is implemented by clustering and frequent item sets mining techniques. Finally, Detective will shed light on evidence collection by the information obtained from previous steps.

Compared with current memory forensic tools (e.g. the work [7, 8] focusing on recovering a complete list of processes

*Corresponding author. This work is supported by the National Natural Science Foundation of China (61100198/F0207, 61100197/F0207). And we are grateful for the valuable data provided by Virus Total.

or enumerating all loaded programs), Detective pays more attention to automated identification and analysis of interested processes, which is important but has not been well solved. Although Detective is currently designed for Windows, the key ideas are also fit for other operating systems. Experiments on real world malware set have proved that Detective could achieve a practical accuracy rate at a reasonable cost.

II. BACKGROUND AND OVERVIEW

A. Observations

DLLs that a process loads offer insight into the functions of the corresponding program. Windows API is implemented as a set of DLLs, so the list of DLLs can tell us what this process could probably do, and this information is in small size and moderate semantic granularity. In addition, DLLs can be obtained not only from memory dump but also by certain online obtaining tools [9], so DLLs-based method is fit for both online and offline forensics. Moreover, DLL patterns of different categories of processes are essentially different. It can not only tell the difference between benign and malware processes, but can also be used to distinguish between different kinds of malware processes by clustering techniques. If we cluster one unknown malware process into a known group formed by lots of known malware processes, the behaviors of this unknown sample can be appropriately represented by features extracted from that group. And those features also give us the clue to collect the evidences of the sample.

B. System overview

In order to automatically locate and analyze malware processes in forensic scenarios, Detective is divided into three main components. Fig. 1 shows the key components and basic inputs and outputs.

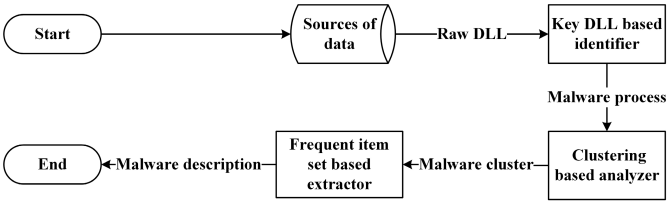

Fig. 1. Overview of Detective

III. SYSTEM DESIGN AND IMPLEMENTATION

A. The key DLL based identifier

As is shown in Fig. 2, identifier can be divided into four main steps with an assistant process.

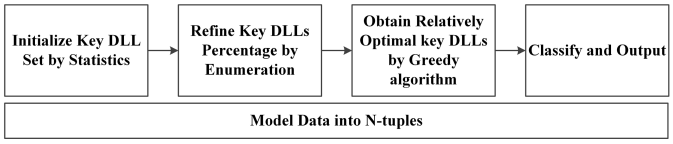

Fig. 2. Main Steps of Identifier

1) Modeling of n-tuples.

DEFINITION 1. *An n-tuple is a sequence of 1 or 0, where n is a non-negative integer and indicates the number of 1 or 0 in the tuple. An example of n-tuple is {1, 0, 0, 1, 1, 0}.*

DEFINITION 2. *A flag bit is a bit added to the end of an n-tuple to indicate whether the process it represents is a malware process or not. It is used in the classification process.*

The goal of this step is converting a DLL list into a data structure that can be processed by analyzing algorithms easily and quickly. In order to achieve this goal, identifier chooses n DLLs as key DLLs. They form the basis of n-tuple. Then it searches the DLL list of each process to check if there are key DLLs in it. If any key DLL is found, the corresponding bit in the tuple is set to 1, otherwise the bit is set to 0. For example, if the key DLLs are kernel32.dll and shell32.dll while the list contains kernel32.dll and wininet.dll, then the n-tuple is {1, 0} as kernel32.dll is in the list and shell32.dll is not. We add a flag bit 0 to the end to make the tuple {1, 0, 0} eventually.

2) Initializing key DLL set.

Key DLLs are the base of n-tuple model. Firstly, process-specific DLLs cannot be key DLLs because key DLLs are used to describe the common characters of many processes in each category. Although DLLs under system32 folder are common DLLs, the quantity of them is large, for example, there are more than 1600 DLLs in System32 folder in a fresh 32-bit Windows 7. If we choose all of them as key DLLs, the dimensions of our model will become very large. In fact, less than one third of them appear frequently in the list of processes (either malware or benign ones) according to our statistics. In order to reduce the number of Key DLLs, we initialize the key DLL set with those appear in DLL lists of both malware and benign processes in training set. Experiments have proved that this method can achieve higher accuracy.

3) Refining and obtaining the "optimal" key DLL set.

Identifier needs enough training data containing both malware and benign processes to achieve high accuracy. The intuitive way for creating training dataset (Volatility-based method) requires lots of time and spaces. To address this issue, we introduce an extra validation procedure to the normal training and testing process. Data that can be obtained easily like reports from Virus Total can be used as training set. And data obtained by Volatility-based method can be used as validation set. The key difference between training and validation set is the latter is smaller and more similar to the data we get in real forensic cases. When both sets are ready, we could list all the possible combinations of DLLs in the key DLL set obtained by statistics to find out the best subset theoretically. When we say subset A is better than subset B, we mean the classifier built on A performs better than the classifier built on B on validation set. However, it is impossible to do the exhaustive listing as the number of combinations goes up exponentially. For instance, if we start with 200 DLLs, the number of combinations is 2^{200}-1. Here we present a two-step way to solve the problem. The first is shown in algorithm 1.

Algorithm 1 Refining Key DLL set

```
bestPairList consists of pairs of upper and lower bounds
1: Subset Getter(upper, lower){
2:     If (pair. Lower<=Percentage of some DLL<= pair. Upper)
3:         Add the DLL to subset;
```

Algorithm 1 Refining Key DLL set

```
4:      Return subset;
5: }
6: Iterator(gap){
7:    for(start=0; start<=0.5; start + gap){
8:      for(end=0.5; start<=1; end + gap){
9:        test Performance on (subset Getter(start, end));
10:         if(performance is better)
11:              update bestPairList;
12:    }
13:   }
14:   Return bestPairList
15: }
```

Subset Getter in line 1 uses two parameters namely upper bound and lower bound to generate a subset of the initial set. For instance, when upper bound equals 90% and lower bound equals 1%, it means a subset containing all DLLs that have an appearance rate between 1%(excluded) and 90%(included) in both malware lists and benign lists. The enumeration happens in Iterator in line 6. If we set the step length to be 0.5%, we make the number of possibilities less than 10000. Code on line 9 test the performance of the model built on that subset. After rounds of enumerations, we get the best pair list in line 14 and they can be converted to best subsets by Subset Getter. The second is shown in algorithm 2.

Algorithm 2 obtain the "optimal" key DLL set

```
Sb adds Sr equals the initial set
1: Best Subset Getter(subset list){
2:      for(subset in subset list){
3:          if(subset performs better than best);
4:             best = subset;
5:      }
6:       return best;
7:  }
8:  Optimizer (Sb, Sr){
9:     iteratively add on element in Sr to Sb to form St list
10:    Sb = Best Subset Getter(St list);
11:    update Sr;
12:    if(performance of Sb does not improve anymore)
13:       return Sb
14:    else
15:       Optimizer (Sb,Sr);
16: }
```

Here we call the best subset(s) got in step one Sb, the set containing DLLs appear in initial set but not in Sb is called Sr. Each time we take one DLL from Sr iteratively and add that to Sb to form a set named St in line 9. If there are n elements in Sr and m elements in Sb, we will have n different St, each containing m+1 elements. Best Subset Getter in line 1 builds models on all these St and chooses the best one(s) to be the new Sb, meanwhile, Sr is updated each time. Optimizer stops if the performance of Sb on validation set does not improve anymore. This greedy algorithm offers us a relatively optimal key DLL set at a reasonable cost.

4) Classifying and output.

We need a proper classification algorithm to be more efficient. After trying many mainstream algorithms including decision tree induction, Bayes classification, rule-based classification, support vector machines and so on, we choose a novel Bayes model named hidden naive Bayes (HNB) finally. That's mainly due to the highly coupled character of DLLs. HNB can deal with high coupled attributes. In HNB, a hidden parent is created for each attribute which combines the influences from all other attributes.

Each version of Windows has a limited and stable collection of system processes, such as lsm.exe and lsass.exe. Experiments show that they act like "noises" in the classification. Hence, these system processes are filtered out in the first place before classification. The elimination of these processes improves the accuracy of identifier.

When real targets come, they are converted into n-tuples based on key DLLs obtained in last step. Then they are regarded as test set and identifier will generate the flag bits for them, indicating whether they are malwares or not.

5) Implementation.

We implement identifier with five modules. N-tuples converting module executes the modeling task. As the forms of data vary, the converting process might be slightly different. The module initializes the tuple with zeros and alters the bit to one if corresponding DLL is found. It adds a zero to the end of each tuple as flag bit, which means that the tuple stands for a malware process. Key DLL statistic module does the statistics work for training data so as to initialize the key DLL set. Percentage enumeration module generates the best pair(s) of upper bound and lower bound for identifier. It enumerates all possible percentages at a certain gap. This module works together with Weka, a collection of machine learning algorithms for data mining tasks. Implemented in python, greedy algorithm module aims to find the relatively optimal key DLLs on the base of best pair(s). Finally, output module ends the mission of this component.

B. The clustering based analyzer

As is shown in Fig. 3, analyzer can be divided into three steps with an assistant process.

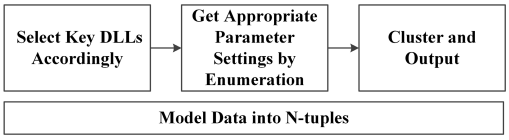

Fig. 3. Main steps of Analyzer

1) Modeling of n-tuples.

Just like the modeling process in identifier, training data needs to be converted into n-tuples. However, the goal of selection of key DLLs differs slightly. In identifier there are no parameter settings and the key DLLs are meant to identify malware processes from benign ones. In analyzer, however, key DLLs are chosen so that features of different malware processes could be represented and similar processes come into same clusters. Analyzer uses the same key DLLs as in identifier by default. Personalized DLLs whose frequencies of appearing in training dataset within an upper bound and lower bound are also supported based on statistics over training data.

2) Parameter settings.

As for the parameter settings, investigators could of course assign all parameters the algorithm (in our framework, it is DBSCAN) needs, they can also just give conditions like the

max number of clusters and max noise rate they could tolerate. Then analyzer can calculate the possible parameter settings.

3) Clustering and output.

When investigators get malware processes from identifier, the category of them becomes the next goal. Analyzer groups a set of malware processes based on their DLL lists in such a way that malware processes in the same group are more similar to each other than to those in other groups.

Analyzer uses DBSCAN as the clustering algorithm as it does not require investigators to specify the number of clusters in the data a priori, which is consistent with "no prior knowledge" purpose of Detective. Being able to find arbitrarily shaped clusters, being robust to outliers, being mostly insensitive to the ordering of the points in training data and low complexity make DBSCAN a reasonable candidate for analyzer. We could not expect a comprehensive cover of malware categories of all time if training data does not cover that much. For instance, if training data covers popular malware samples of the recent months, then the groups generated probably include most malware categories appeared in that time period. It grants investigators the freedom to choose the coverage of analyzer from another perspective.

When the model is built, analyzer convert targets into n-tuples based on key DLLs and regard them as test set. It will give the number of clusters each target belongs to, indicating its family.

4) Implementation.

We implement analyzer with four modules. N-tuples converting module does the same job as before. Personalized key DLL module enables investigators to change key DLL set according to their needs by giving an upper bound and lower bound. Appropriate parameter setting generating module generates appropriate parameter settings for investigators when they just want to give conditions like the max number of clusters and max noise rate they could tolerate. In order to generate an appropriate parameter setting, this module iteratively tries a large range of parameters and tests these results to see if they comply with the conditions given by investigators. This module is practical due to the low complexity feature of DBSCAN. We build our own version of DBSCAN in weka. It first calculates the instance between of existing instances. If it is within the range the parameter specifies, the module then check the minimum points to see if it is qualified. If so, the module returns the number of clusters and break. Or it will continue this process. If no match is found, it marks the instance as noise.

C. The frequent item set based extractor

As is shown in Fig. 4, extractor can be divided into three steps with an assistant process.

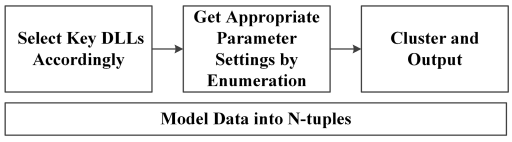

Fig. 4. Main steps of extractor

1) Modeling of n-tuples.

To find the meanings of each group, extractor tries to find the frequent item sets of each group. However, we need to change these n-tuples within each group by change key DLLs as the purposes in analyzer and extractor are different. In analyzer, they are meant to make similar processes come into same clusters while in extractor they represent all the possible behaviors of the group. Hence key DLLs here include all the DLLs that appear once or more in training data.

2) Calculating frequent item set and tags.

When the converting work is done, extractor begins to tackle the second problem. We observe that the number of each bit of the n-tuples within the same cluster tends to be the same, either 1 or 0. Actually, most bits get 90% of their numbers to be the same one, thus making frequent item sets obvious with high value of support. With this in mind, extractor get frequent item sets by statistics and validation. Extractor might not get all the frequent item sets, useful ones are obtained at a reasonable cost. Besides, if training data are from Virus Total, extractor uses hashes representing samples in each group to get the tags all these antivirus vendors give in their reports and does a little statistics work within each group. For instance, extractor could tell you what McAfee think of all these samples of some group and the concrete percentages of each tag. The clue is obvious if McAfee tags 90% of the samples in this group as something as Heuristic.BehavesLike.Win32.Suspicious.D.

3) Generating strategy.

DEFINITION 3. *A malware profile is a structure that describes a malware process in high semantic level. The structure consists of the meanings of the DLLs in the frequent item set it belongs to and the tags of the group it belongs to.*

With malware profiles in hand, the procedure of searching evidence starts. Extractor mainly offers strategy with the help of Volatility as it is widely used in forensic scenarios. First of all, if the tag is available, investigators might receive guidance from sites of these vendors. For instance, if the tag part of the profile indicates the target has something to do with the network, command 'netscan' is recommended. However, tag parts are not always available and not accurate sometimes. Frequent item set part of the profile tells a vivid story of what the malware could probably do. The purpose of each DLL in the profile draws a picture of potential behaviors of the target. All these behaviors become the clue to get the evidence. For instance, Rpcrt4.dll contains the routines, tables, and data that support communication between clients and servers. Its existence tells you command 'netscan' might be your choice.

4) Implementation.

We implement identifier with three modules. N-tuples converting module does the same job as before. Malware profile generating module focuses on frequent item sets and tags. Finally strategy module provides mapping relationship between details of malware profiles and practical strategies.

IV. EVALUATION

Evaluation focuses on three abilities of Detective, i.e. the ability of identifying the malware, the ability of classifying the malware and the ability of instructing the evidence collection.

A. Evaluation of identifying abilities

1) Data settings.

As is shown in Table 1, we have a training set, a validation set and two groups of test sets. The training set is comparatively large and the source of the data is not from Volatility for reasons stated in system design. We get the validation set from Volatility as we expect the final model to deal better with targets from Volatility in real forensics scenarios. The remaining two groups of test sets are meant to simulate data from two normal computers.

TABLE 1. DATA SETTINGS OF EVALUATION OF IDENTIFYING ABILITIES

Sets	Amount (malware)	Source	Amount (benign)	Source
Training	3300	Virus Total	181	Process Explorer
Validation	45	Volatility	40	Volatility
Test A	25	Volatility	5	Volatility
Test B	20	Volatility	4	Volatility

2) Correctness.

The number of key DLLs turns out to be 121 and then percentage enumeration module initializes the key DLLs for greedy algorithm module. Here we tried a few mainstream classifying algorithms on the basis of key DLLs and the result is shown in initial correctness in Table 2.

TABLE 2. CORRECTNESS BASED ON KEY DLLS

Algorithms	Initial correctness	Final correctness
AODE	69.4%	84.71%
WAODE	70.59%	84.71%
Naive Bayes	62.35%	75.29%
RBFNetwork	60.00%	76.47%
IBK	72.94%	64.71%
Kstar	62.35%	71.76%
LWL	55.29%	61.18%
BFtree	70.58%	76.47%
J48	65.88%	90.59%
RandomTree	54.11%	62.35%
HNB	80.00%	94.12%

HNB performs better than most other algorithms on initial key DLLs. Greedy algorithm module then comes to improve the key DLL set specifically for Volatility sources. Fig. 5 shows the trend of correctness for greedy algorithm. It finally stops at 94.12% and we get the final key DLL set. Then we do the test on mainstream classifying algorithms again. The result is shown in final correctness in Table 2. Correctness of most algorithms improves more or less and HNB still has the best correctness. It proves that the validation process really works.

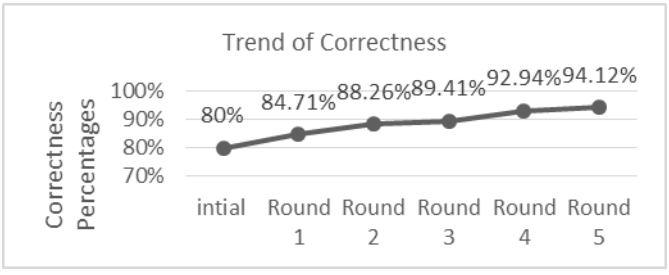

Fig. 5. Trend of correctness for Greedy algorithm

Finally we test our model on two test sets and the result is shown in Table 3. The test results turn out to be reasonable and identifier does not miss one malware in tests.

TABLE 3. CORRECTNESS OF TEST SETS

sets	Correctness	False positive	False negative
Test set A	93.33%	2	0
Test set B	91.67%	2	0

3) Performance.

The evaluation test of performance is done on a PC with Intel i5, and a RAM of 8GB. The execution time of all five modules of identifier each is under 15 seconds. The cost is based on the scale of data described in data settings. Besides, only identifier output module needs to be executed each time. Others are executed only when training data used last time does not suit the current situation anymore.

B. Evaluation of clustering abilities

1) Data settings.

we have a training set with 3300 malware processes from virus total and a test set with 4 malware process from Volatility. The training data set is supposed to be one that includes any categories of malwares that need to be clustered. It is actually a quite personal set. The set we use contains data that were active in the early 2014 recorded by Virus Total.

2) Correctness.

Unlike classifying, there are no obvious standard of correctness in clustering. The key lies in whether the group that the target is clustered into could really represent the target and we will see that in the case study of extractor. As training data is quite personal, the results are just for reference. Here the value of epsilon is 1.0, the value of minimum points is 4, the number of processes is 3300 and the noise percentage is 9.79%.

3) Performance.

On the same PC, The execution time of all four modules of identifier each is under 15 seconds. The cost is based on the scale of data described in data settings. If investigator gives the parameters directly, appropriate parameter setting generating module does not need to be executed.

C. Case Study

Here we use a case to evaluate the availability of guidance. Extractor gets a malware process from analyzer whose hash is eb227cc81f12afc367eea595c5c7adfe44443507ace8ab81b3bb30a14eb5626f in Virus Total and we know nothing about it except the name of the process (eb227cc81f12af), the process id (2892) and some DLLs in the frequent item set in its profile.

The challenge is to find out what it is and related evidence in Volatility. Normally we have a profile including the tag and the frequent item set and tag is easier to use. Here, we just come to the frequent item set to show the hard part. And often, we start with the most informative DLLs in the frequent item set of the profile. In this case, it is Ole32.dll. As it indicates communications between processes, strategy module suggests checking the relations of between different processes and command "pslist" is executed. Result shows that process "explorer" creates process 3016, process 3016 creates process

2892 and process 2892 creates 2896.(Process id might vary with different image, the relationship holds.) It is not surprised that there are three processes with the same name as malware programs might have more than one process and not every process is recognized or clustered into this particular cluster. However, if we get any one of them, we get the clue to others. Here we need to pay attention to these three processes.

As the number of processes are limited and they are created in a chain, strategy module suggests that we might consider command "malfind" for the last process created. One memory segment is then detected because it is executable, marked as private and has a VadS tag, which means there is no memory mapped file already occupying the space. This can be the evidence of an API hook. In order to verify our thoughts, we download this real world malware from Virus Total by its hash value and get it running. It is a tool designed to hook other processes, which confirms our findings. Memory forensic can never be an easy task and Detective could not get the evidence for investigators directly. There are many other ways to solve the problem in this case if you are quite experienced. However, suggestions from the clue strategy module often becomes the key to success and accelerates the forensic process.

V. Related work

A. Memory acquisition and recovery.

In the area of memory acquisition, techniques are divided into hardware-based and software-based methods. Memory can be obtained by a dedicated hardware card or via a special hardware bus. Virtualization software can also be helpful as the memory of a virtual machine can be saved into a file. Besides, Windows provides software crash dumps and hibernation files, which are also valuable sources. All these techniques can be evaluated by two factors namely atomicity and availability. Atomicity reflects the demand to produce an accurate and consistent image and availability refers to the applicability of a certain technique on arbitrary system platforms for any given scenario. As for the recovery, most of researches concentrate on how to restore processes, system registry, network information and files completely and correctly from raw memory image as malwares often try their best to hide their traces from being restored. The forensic framework Volatility [1] implements most of the methods in this area.

B. Malware detection: static and dynamic analysis.

As many commercial-off-the-shelf malware detection tools still use signature-based techniques[2] that are not reliable, researchers nowadays in this area mainly focus on behavior based detection techniques that are rust to code obfuscation and polymorphism. They can be classified into the static and the dynamic ways. Static analysis techniques try to analyze the information and contents of a given file. Often they scan the header and the payload of the file to get necessary information to build models to detect malwares [3]. While a number of static analysis techniques show promising results, they are usually extremely slow [4] and thus unsuitable for real-world forensic scenarios. Dynamic analysis techniques try to model run-time behavior of process. Often they rely on system call sequences [5] or flow graphs [6]. They analyze the execution of a malware or the effects that the malware has on the operating systems. Those methods using system call sequences build the behavioral model of malwares on the basis of the sequence of invoked system calls and their arguments. And the idea behind flow graph is to employ taint technique by analyzing the data flow model.

VI. Conclusion

We have presented the Detective framework that automatically locates and analyzes malware processes in forensic scenarios via DLLs. Detective mainly includes three components: identifier, analyzer and extractor. Identifier could classify benign and malware processes based on models built on their DLLs at a reasonable accuracy. Analyzer then clusters malware processes into different groups formed by training data with DBSCAN algorithm. Next, extractor describes the group with frequent item sets and tags, characterizes the unknown malware process with known functions or features, and sheds light on evidence hunting task. As detective has been a proof-of-concept prototype so far, many functions are implemented as separate tools and some functions are still manual, more work will be done regarding to the integration to make detective more automatic. Besides, more versions of Windows will be covered and more customized classifying and clustering algorithms for this purpose will be developed. Our experiments prove it is practical to create a unified interface to make the whole process automatic and effective as long as DLL data is provided, no matter offline or online. Meanwhile, both accuracy rate and time cost are reasonable.

References

[1] The Volatility Framework, https://code.google.com/p/Volatility/; 2014.

[2] P. Szor, The Art of Computer Virus Research and Defense, Addison-Wesley Professional, 2005.

[3] M.Z. Shafiq, S.M. Tabish, F. Mirza, M. Farooq, PE-Miner: mining structural information to detect malicious executables in realtime, in: Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection, 2009.

[4] M. Christodorescu, S. Jha, S. Seshia, D. Song, R. Bryant, Semantics-aware malware detection, in: IEEE Symposium on Security and Privacy, 2005.

[5] F. Ahmed, H. Hameed, M.Z. Shafiq, M. Farooq, Using spatio-temporal information in API calls with machine learning algorithms for malware detection, in: Proc. of the 2nd ACM Workshop on Security and Artificial Intelligence, 2009.

[6] H. Yin, D. Song, M. Egele, C. Kruegel, E. Kirda, Panorama: capturing system-wide information flow for malware detection and analysis, in: Proc. of the 14th ACM Conference on Computer and Communications Security, 2007.

[7] A. Schuster. Searching for processes and threads in Windows memory dumps. Digital Investigation July 2006d;3(1):10e6, http://computer.forensikblog.de/en/2006/03/dmp_file_structure.html.

[8] B. Dolan-Gavitt, A. Srivastava, P. Traynor, J. Giffin. Robust signatures for kernel data structures. In: Proc. of the 16th ACM conference on computer and communications security; 2009.

[9] ProcessExplorer, http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx; 2014.