Memory access patterns for malware detection

Sergii Banin, Andrii Shalaginov, Katrin Franke
sergii.banin@ntnu.no, andrii.shalaginov@ccis.no, kyfranke@ieee.org
Norwegian Information Security Laboratory
Center for Cyber- and Information Security
Norwegian University of Science and Technology

Abstract

Malware brings significant threats to modern digitized society. Malware developers put in significant efforts to evade detection and remain unnoticed on victims' computers despite a number of malware detection techniques. To eliminate known and noticeable traces in memory, network or disk activities, they use encryption and obfuscation. Because of this, there remains a strong need for new malware detection methods, especially ones based on Machine Learning models, because processing of large amounts of data is not a suitable task for a human. This paper presents a novel method that could potentially detect zero-day attacks and contribute to proactive malware detection. Our method is based on analysis of sequences of memory access operations produced by binary file during execution. In order to perform experiments, we utilized an automated virtualized environment with binary instrumentation tools to trace the memory access sequences. Unlike the other relevant papers, we focus only on analysis of basic (Read and Write) memory access operations and their n-grams rather than on the fact of a presence or an overall number of operations. Additionally, we performed a study of n-grams of memory accesses and tested it against real-world malware samples collected from open sources. Collected data and proposed feature construction methods resulted in accuracy of up to 98.92% using such Machine Learning methods as k-NN and ANN. Thus, we believe that our proposed method will serve as a stepping stone for better proactive malware detection techniques in the future.

1 Introduction

Malware is the malicious software designed to perform an illegal or unwanted activity on a victims system. The VirusShare database [1] contains 25,072,568 malware samples as of 9th May, 2016. When a new malware sample is detected there is a time gap between the moment antivirus vendors can analyze it and the moment they update their databases for their customers. To thwart detection,

This paper was presented at the NISK-2016 conference; see http://www.nik.no/.

malware developers develop additional techniques to evade detection by antimalware software through the use of different obfuscation techniques such as encryption, polymorphism, metamorphism, dead code insertions, and instruction substitution [2] in order to change the appearance of a file and its static characteristics. For example, it is possible to change hash sums used as file signatures (such that SHA-1 or md5) by simply changing different strings in the file. Further, dead code insertions can be used in executables to change opcode sequences, making detection troublesome.

There are two main approaches for malware analysis that can be found in the literature [3, 4]: *static* and *dynamic*. *Static analysis* is done on a malicious file without its execution and aimed at collecting various static characteristics such as bytes, opcodes and API n-grams frequencies, Portable Executable header features, strings and others [2, 5, 6]. *Dynamic analysis* is based on running a malicious executable in a controlled environment and tracking its activity within the system. Such activities include network, registry and disk usage patterns, API-calls monitoring, instruction tracing, memory layout investigation and others [7]. To collect such information one can use either specialized sandboxes like Cuckoo [8] or utilize any Virtual Machines such as VirtualBox accompanied by a debugger or other watchdog software. Despite the fact that some authors consider [9, 10] disk and network activities crucial for malware detection, few authors have outlined the utility of memory properties analysis [11, 12].

This paper presents a novel methodology for malware detection. It is based on the extraction of the memory access sequences (further called *memtraces*) for both benign and malicious executables using the dynamic binary instrumentation tool Intel Pin [13]. This dynamic binary instrumentation tool is used for live analysis of binary executables and allows for analyzing different properties of execution such as memory activity, opcodes, addressing space, etc. Our proposed methodology is based on the assumption that similar opcodes with similar arguments will result in nearly the same memtraces, which is explained later in the paper. Thus, we apply an n-gram technique to extract features from memtrace sequences in order to perform benign against malicious classification. Moreover, we used specially-tuned feature selection to be able to verify classification accuracy while adhering to a set of community-accepted Machine Learning (ML) methods. It will be shown that our method can find an application in proactive malware analysis with reliable results using only fraction of the execution records of the malware sample. Unlike the other related works, where authors worked on specific ML methods and other dynamic features, we focus primarily on memory access sequences and patterns within them. So, this paper contributes to a new malware detection methodology and test it against real-world samples.

The remainder of the paper is organized as following: 2 presents an overview of the dynamic malware analysis, including existing behavioural characteristics as well as how memory activity can be used for identification of malicious activities. Further, 3 presents our contribution towards the memtraces analysis for malware detection. Description of the collected malware samples and analysis of the results are given in the 4 section. Finally, 5 contains our final remarks and conclusion.

2 Memory patterns in malware detection

In this section we provide a short overview of existent studies that are related to ours, because there dynamic binary instrumentation tools and memory activity analysis were also used.

Dynamic malware analysis involves malware execution in the controlled environment with further investigation of its activity and any possible traces that can be found in the system. In the Malware Analysis Cookbook, Ligh et al. [14] defined the following automated procedure for dynamic malware analysis covering a set of predefined operations ranging from VM start up to traces collection.

According to SANS [3] one may conclude that a number of behavioural characteristics can be used to identify whether or not an executable file has some malicious functionality. With a use of such dynamic malware analysis, it is possible to collect different types of features such as file system events, registry changes, API and DLL calls, network and memory activity [9]. In the paper [10], the authors claim that memory analysis without ground-truth cant be considered trustworthy, especially on proprietary operating systems (e.g. MS Windows). They investigate the accuracy and efficiency of traversal-based and signature-based memory analysis tools (Volatility [15] framework and its plug-ins), which are designed to gather information about processes, modules, files etc. Further work [10] also examined accuracy and efficiency of robust field- and graph- based signature schemes SigField [16] and SigGraph [17]. They compared results from binary analysis tool and Volatility over the Virtual Machine memory, claiming that traversal-based and signature-based methods tend to produce less accurate results.

A methodology for malware analysis with using Intel Pin was discussed earlier [11]. The model was first tested in a virtual environment and afterwards in a real environment with Windows XP or Xen Linux installed. They extracted the following characteristics: system or user API calls if any file or folder was modified, calls that create hard or symbolic links, calls or arguments of function *exec()* and instructions that performed memory operations *read* and *write*. Unlike in this paper, where we focus on single memory access operations generated by single opcodes, they utilized *basic blocks* of a program. The basic block is an instruction sequence which is executed between control flow transfer instructions. Among other features, the authors used the fact of presence, size of transferred data and memory range of memory access operations within the basic blocks. Recording the execution trace they generated regular expressions and security policies, which then were used for malware detection. As the result, they achieved 100% detection rate for original and obfuscated malware samples on both Windows and Linux. The authors claim that their system is capable of accurate malware detection with 93.68% code and path coverage of input-dependent executables. Finally, it is worth mentioning the work [12] that proposed the ensemble learning technique of malware detection based on a number of features extracted with Intel Pin [13]: frequency of opcode occurrence, presence of particular opcode, difference between frequency of opcode in malware and benign executables, distance and presence memory reference and total number of load and store memory operations as well as branches. For each executable they collected a feature vector for every 10,000 committed instructions and achieved classification accuracy up to 95.9% with a specialized ensemble classifier.

3 Memtraces for malware detection

The proposed method based on memtraces is described below. Steps from characteristics collection and feature construction for future use in ML are presented.

Collecting memory access

Opcode (API calls) n-grams have consistently been successfully utilized as reliable features for malware detection [18]. No matter the programming language or frameworks used for developed programs, a compiled PE32 executable can be represented as a sequence of opcode instructions. Opcodes (or assembly commands) are basic commands executed on the hardware level. Some operate only with CPU's registers, and as such $XOR\ EAX, EAX$ or $MOV\ ESI, EBX$ wont have any interaction with virtual memory, while others can generate sequences reading from and writing to virtual memory operations, for instance $MOV\ EBX,\ VAR\_NAME$ which reads from memory and $MOV\ [VAR\_NAME],\ 110$ which writes to it.

In this paper we analyse sequences of basic memory access operations which are R for Read and W for Write operations. Our goal is to record a sequence of memory access operations, or *memtraces*, and analyze this sequence. The majority of modern desktop computers utilize x86 compatible architectures that were introduced in order to implement pipelines and, as a result, increase execution speed. Modern x86 compatible CPUs translate opcodes into a sequence of micro-operations (or *uops*) responsible for loading and storing data, interacting with arithmetic logical units, branching, and so on, each *uop* executed on the specific port. Some authors collected information about number and types of micro-operations used by CPUs in order to execute certain opcodes, as in [19] where such information was collected for Intel architectures ranging from Pentium to the Skylake architecture. For example, in Sandy Bridge architecture *port p23* stands for memory read or address calculation, and *p4* for memory write.

To be more specific, we focus on opcodes that allow interaction with memory such as MOV, AND, XOR, ADD etc. It was found that, regarding the number of *load* and *store* micro-operations, the opcodes were similar to those presented in the book [19]. Our scope was confined to solely memory-related activity, and we looked for a number of micro-operations going to a memory read port (e.g. *p2* or *p3* for Ivy Bridge and Skylake architectures) or a memory write port (e.g. *p4* for Ivy Bridge and Skylake architectures). No similar information was found for AMD CPUs, so the scope was also limited to Intel processors. With the help of Intel Pin, we checked data that are usually transferred by the detected read and write operations where we found that most of the memory operations involve transfers of 4 bytes. (1- 2- 8- and 10-byte memory accesses were also found.) This means that Intel Pin is capable of detecting memory operations on the level of separate opcodes and has the desired granularity. From the results of this study we concluded that opcode with similar parameters will generate similar memory access sequence regardless of the overall task of the executable and Intel CPU model. To verify this we tried, with help of Intel Pin, to make the output contain executed opcodes and its (if exists) memory operations. The examples of opcodes and memtraces captured from *calc.exe* benign executable taken from Windows 7 are given below.

```
[mov edi, dword ptr [ebp-0x20]]
R
[add dword ptr [eax], ecx]
RW
[mov dword ptr [ebp-0x8], edx]
W
```

This sequence can be explained as following:

- *mov edi, dword ptr [ebp-0x20] reads* from memory, and writes this information to the *edi* register. According to [19] instructions of type *MOV Register, Memory* for all addressing types involves 1 read operation (1 microoperation for the port p23).
- *add dword ptr [eax], ecx reads* data from address that is previously calculated (by the memory read port which has additional function of address calculation), then it calculates the sum and later data is *written* to the already calculated address. Instructions of type *ADD Memory, Register* involves 2 microoperations to the store port (one for address calculation and one for reading) and 1 microoperation for writing [19].
- Instruction *mov dword ptr [ebp-0x8], edx* generates *W* (*Write*) because *MOV Memory, Register* is a memory writing.

Based on what was said above, we highlight two hypotheses: (i) Opcode n-grams are reliable features for malware detection as described earlier, and (ii) Opcodes with similar arguments will produce similar memory activity. As results, memtraces can be used as robust features for identification of malware samples. It is hard to say how many memtraces are required for good detection rate; this will have to be studied later on. To start with, however, we decided to restrict length of recorded memtrace sequence to 10 millions of records. According to our measurements, in order to perform 10,000,000 memtraces, an executable (from our dataset) spends about 0.053 seconds on our hardware. Time was recorded with a use of *chrono* a C++ library. This makes our system potentially applicable for the systems which require near real time malware detection because classification of a software sample will take less than a second. Afterwards, the original memtrace sequence is pruned to get first 100,000 and then 1,000,000 memtraces in order to study influence of the memtraces sequence length on the accuracy.

N-gram as feature extraction

In order to detect malicious executable we record the sequence of memory access operations(memtraces). We define original memtrace sequence $S_{original}$ as a set of memory access operations: $S_{original} = (m_1, m_2, ...m_l)$ where l is the number of memtraces recorded during execution of a program and m_i is either Read or Write memory access operation. Memtrace sequence ms is defined as subgroup of original memtrace sequence where $ms \subseteq S_{original}$. Here ms is constructed from memtraces: $ms = (m_{k+0}, m_{k+1}...m_{k+p-1})$ where $k \in [1, l-p+1]$ is the starting position of certain memtrace sequence, and $p \in [1, l]$ is the length of memtrace sequence. So, the memtrace sequence of length p=n is called n-gram. We use only R for read and W for write operation regardless to size of the transmitted data. Then we extract n-grams of preferred size from the original memtrace sequence. While authors who apply opcode n-grams for virus detection usually use *n=1, n=2* [20] *n=3, n=4, n=5* [21], we could not use *n* of such small sizes. Here are the reasons:

- Executable may contain hundreds of different opcodes. Thus, sequences of *OR,OR,OR* and *ADD,ADD,ADD* represent different features for opcode based methods.
- We trace only memory accesses. Thus, both *OR,OR,OR* and *ADD,ADD,ADD* (*Memory, Register* operands) can be recorded as *RWRWRW* and *RWRWRW.*

- Memtraces sequence is a binary sequence, because it contains only two symbols. For the length of 1 or 10 million, class-wise frequency for memtrace n-gram of size 6 will be close to uniform, and there is a high likelihood, that among benign and malicious classes there will be no a single unique n-gram for particular class.

During initial experiments, we also found that there is no class-unique n-grams up to the size of 12 for the dataset used for experiment. It can be explained by the fact that not all the opcodes generate memtrace activity. From one perspective it makes less data to work with, but from another, it could probably result in lower classification accuracy. So, we decided to start from n-gram size of *16*, and proceed with *20,24,36,48,72,96* to cover different n-grams and have feasible processing and analysis overhead. Another reason of increasing n-gram size is that probability of particular n-gram to occur in a sequence of memtraces is higher for smaller n values, thus utilizing small n values can result in impossibility of finding unique features and low classification accuracy. We limit n-gram size to 96 because our scripts were not able to finish bitmap construction for 10,000,000 memtraces due to low memory error.

Feature Selection

To use extracted memtraces for training ML models, we need to extract features that will provide the best description of classes [22]. A perfect feature for classification task is the feature, that exists only in particular class and is present in all instances of this class. However, in real-world problems, it is usually very hard or even impossible to find such features. So, the task is to find features that fit previously stated request better than others using the following steps:

1. For each class (benign and malicious) construct vector of n-grams (which are unique within the class) and their class-wise frequencies, e.g.: $[[WWR, 1.0], [WRW, 0.87], [RRR, 0.66], ...]$

2. Having two vectors, one for benign and one for malicious executables, we delete those n-grams that are present in both vectors, regardless to their class-wise frequencies. In other words, we subtract intersection of two sets from each of them and get two *clean* vectors.

3. From each of clean vectors we select a particular amount (e.g. 100,200,400) of n-grams with highest class-wise frequency and combine them into the final feature vector of length 200, 400 or 800. The numbers 100, 200, 400 were chosen, because many researchers used to utilize feature numbers from 100 to 1,000. So, this a common baseline for similar researches, yet we need to take into account ML software performance.

However, to be able to apply ML classification we need to build a bitmap (matrix) of presence, where "1" is placed if particular instance contains particular feature and "0" if not. Such bitmap is used later to train ML methods. In order to assess model quality, we will use 5-fold cross validation. For results assessment, we will use classification accuracy because it shows how well model performs on the whole dataset. Finally, newly built feature vector is then used for building the bitmap that is later used in Machine Learning algorithms.

4 Experiments & Results

This section is devoted to experiments design and analysis of achieved results of the proposed method.

Computing Environment

All our experiments were performed on Virtual Dedicated Server (VDS) with Intel(R) Core(TM) CPU @ 3.60GHz, 4 cores, SSD RAID and 48GB RAM. Ubuntu 14.04 64 was installed with MySQL 5.5, PHP 5.5.9 and VirtualBox 5.0.16. Windows 7 32-bit was used as guest OS, because of its wide spread [23] and the fact that malware written for 32-bit OS's will run on 64-bit as well. Another reason to use 32-bit version of Windows 7 is that our VDS was not capable of running newer or 64-bit versions of Windows due to virtualization issues.

Malware & data collection

Benign files were collected from clean installations of 32 bit versions of Microsoft Windows OS (XP, 7, 8, 10). The reason of such choice is that there is no publicly available datasets with big amount of benign files. Malicious files were taken from *VirusShare* repository [1] (*VirusShare_00207.zip*). This archive contains collection of PE32 files, which is a very popular executable format due to strong legacy of 32-bit operational systems and compatibility issues. Both benign and malicious files were unsorted and uncategorised. In order to avoid duplicates, files were renamed with their *md5* sums. Having information gained from *peframe* [24] those files, that contain *GUI* field in the peframe output, were selected. The reason of this kind of filtering is that many malicious executables, that don't have GUI, can switch to idle mode soon after start, so, this will bring significant problems to automated dynamic analysis since it could produce very small amounts of data or it will require an unreasonably long waiting time. After files were filtered by presence of GUI, the smallest 1,000 files from each of the datasets were selected. File sizes of selected benign and malicious executables vary from 115.4 KB to 152.1 KB and from 976 bytes to 28.7 KB respectively. The experiments setup included automated execution of malware as specified in the Figure 1 according to [14].

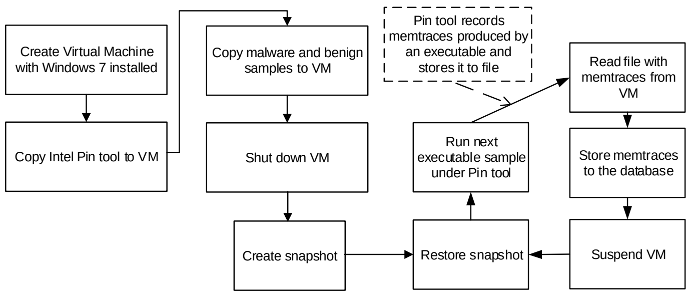

Create Virtual Machine with Windows 7 installed
Copy Intel Pin tool to VM
Copy malware and benign samples to VM
Shut down VM
Create snapshot
Pin tool records memtraces produced by an executable and stores it to file
Run next executable sample under Pin tool
Restore snapshot
Read file with memtraces from VM
Store memtraces to the database
Suspend VM

Figure 1: Automated malware analysis using Intel Pin for metraces sequences extraction

During the experiments only 445 benign and 759 malicious files managed to start since some had *anti-debug* or *anti-VM* features. Final dataset contains 1,204 files and MySQL table with raw memtraces occupies 6.9 GB of storage space and table of 96-grams for memtrace sequence length of 10,000,000 takes 32.2 GB.

Results

We used memory access sequence of lengths 100,000; 1,000,000 and 10,000,000, n-gram sizes of 16,20,24,36,48,72,96 and feature set sizes of 200,400, and 800. The following ML methods were trained: NB (Naive Bayes), BN (Bayesian Network), J48 (C4.5), k-NN (k-Nearest Neighbours), ANN (Artificial Neural Network) and SVM (Support Vector Machine). In this paper we present only the most outstanding achieved results.

Influence of the memtrace sequence length and n-gram on classification accuracy
Considering three lengths mentioned before, we can say, that usage of the first 1,000,000 memtraces is enough to achieve good malware detection rate together with the number of selected features equal to 800. The results of experiments are given in the Table 1. Other configurations gave lower or equal results.

n-gram size	**Machine Learning method**					
	NB	BN	J48	k-NN	ANN	SVM
100,000 memtraces						
16	60.22	60.47	**64.29**	64.20	63.54	63.54
20	60.88	62.71	65.03	**65.61**	63.54	63.54
24	62.46	63.95	67.19	**67.28**	62.13	63.54
36	59.72	60.63	67.61	**67.77**	62.54	63.62
48	59.72	59.72	68.77	**68.94**	62.96	64.45
72	64.12	64.12	**71.26**	**71.26**	64.70	67.28
96	67.03	67.03	70.76	**71.01**	63.87	68.77
1,000,000 memtraces						
16	60.71	61.30	82.97	**83.80**	**83.80**	70.02
20	55.32	55.32	83.89	**84.88**	61.38	77.74
24	56.15	56.81	79.49	**79.57**	61.38	76.50
36	57.64	57.64	78.16	**78.32**	65.70	76.16
48	73.34	73.34	85.71	**85.88**	65.86	85.05
72	**92.11**	**92.11**	90.95	91.20	92.03	92.03
96	94.44	94.44	98.84	**98.92**	**98.92**	98.51

Table 1: Accuracy %, for 800 features

We can see that k-NN and ANN provide best classification accuracy of **98.92%** for 1,000,000 memtraces, 800 features and 96-grams. k-NN also shows most of the row-wise best results in the tables. However, the point where k-NN and ANN reached best accuracy could not be stated as optimal condition for malware-benign classification task. Several results were not gained due to out-of-memory problem, but better results could probably be achieved in future work. From one point of view k-NN is a good algorithm, because it doesn't require actual training phase. However, it makes no generalization about processed data. This can result in overfitting and classification time growth. Among others, J48 and ANN showed good results and could be considered as reliable candidates for malware-benign classification task if k-NN is not suitable for some reason. As we can see from the Table 1, the more memtraces we have in the original sequence - the higher accuracy we gain. This is natural dependency because bigger length of memtrace sequence gives us more information about an executable. As we can see, n-grams (48, 72, 96) of bigger size gives us better accuracy. Finally, we have also studied influence of the number of features on classification accuracy. Similarly to memory access sequence length, overall dependency shows growth of accuracy with growth of feature number as shown in the Figure 2.

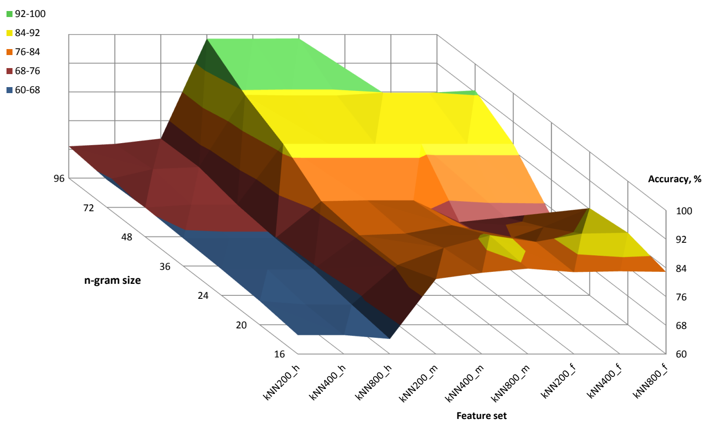


Figure 2: Accuracy depending on n-gram size and number of features (200-800) for all memtrace sequences lengths

Analysis of the classification accuracy

The non-trivial factors that influence classification accuracy are given in the Table 2 first. During the feature selection process we used to calculate intersection of unique n-grams sets from benign and malicious executables. This intersection then was subtracted from both benign and malicious unique n-gram sets. From the Table we can see that characteristics of intersection under different memtrace sequence lengths and n-gram sizes. It contains the following columns: *n-gram size* - the size of n-gram, *Isec* - the number of unique n-grams shared between benign and malicious executables (intersection size), *B_unique* - the number of unique n-grams found in benign executables, *M_unique* - the number of unique n-grams found in malicious executables, *Ratio* - intersection ratio, shows similarity between malicious and benign n-grams sets. Calculated as $Ratio = \frac{Isec}{(B\_unique + M\_unique)}$ The Table 2 contain intersection ratios calculated for the 1,000,000 memtraces.

n-gram size	**isec**	**B_unique**	**M_unique**	**Ratio**
16	50,751	53,587	56,722	0.46008
20	156,553	209,037	240,098	0.34857
24	234,725	364,926	440,593	0.29140
36	372,336	670,314	1,009,935	0.22160
48	481,347	910,659	1,655,659	0.18756
72	694,570	1,384,718	2,944,111	0.16045
96	918,076	1,884,036	4,201,454	0.15086

Table 2: Intersection size and ratio for unique benign and malicious n-grams for 1,000,000 memtraces

Interpretation of achieved results and findings

In the Figure 3 we show dependency between accuracy rate and intersection ratio for 1M of memtraces and all lengths of feature vector. Accuracy rates were taken from k-NN column of corresponding result table because k-NN has shown most of the best results for particular n-gram size. Logarithmic trendlines were added to every series for easier understanding. We will use k-NN's accuracy to illustrate other findings and tendencies for the same reason.

The results achieved by k-NN under all conditions are shown in the Figure 2: x-axis is for feature set, y-axis is for n-gram size and z-axis is for accuracy. Labels

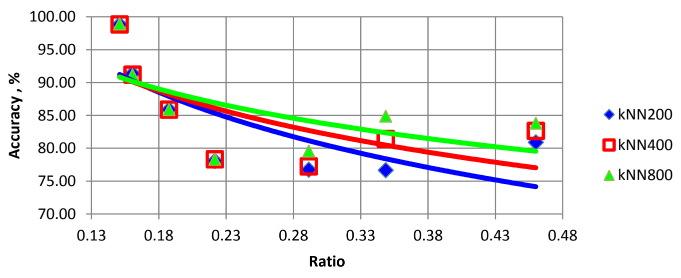


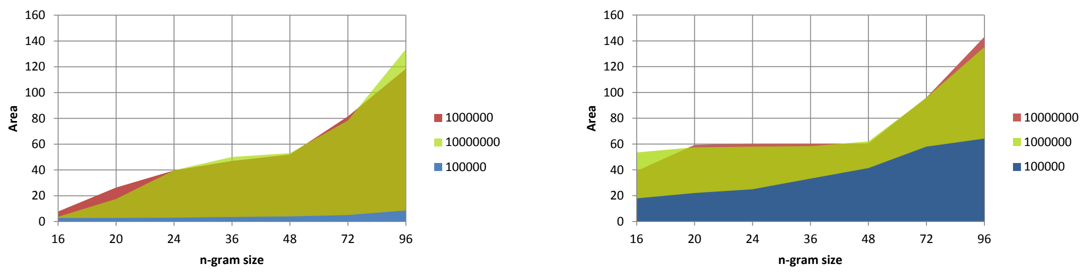
Figure 3: Accuracy vs intersection ratio for 10^6 memtraces

on x-axis are named as *XXX_type* where *XXX* stands for feature number and *type* for memtrace sequence length: **h** for 100,000, **m** for 1,000,000 and **f** for 10,000,000. It is easy to see that growth of feature vector and memtrace sequence length (x-axis) generally results in accuracy growth. Increase of N-gram size and feature number result in accuracy growth, but now it is easier to see that area around n-gram size of 24 contains descending of accuracy (as well as weak matrix sparseness fading and area under class-wise frequency chart growth).

Worth to mention that there is visible correlation between intersection ratio and accuracy rate. The smaller ratio implies the bigger accuracy. This can be explained in the following way. Bigger intersection ratio means fewer features for feature selection. This results in smaller class-wise frequency of a particular feature, hence bigger sparseness of presence matrix. Sparse matrix can worsen generalization of dataset and even result in zero-filled rows, which definitely will decrease accuracy. As we utilize bitmap of presence, and our feature selection method is aimed at selecting only class-unique features, zero-filled row means that particular sample could be difficult to correspond to one of the classes. High matrix sparseness means that many features are not very efficient. So, it will lower ability of ML methods to generalize through the data. We conducted the study of class-wise frequencies of features selected in feature vectors and found that there are no n-grams with frequency *1.0*. This means that either there does not exist a single n-gram that describes just malicious or just benign executables or n-grams with class-wise frequency *1.0* were rejected during feature selection as those present in both classes.

In the Figures 4, *a* and *b* Areas Under Feature Class-wise frequency charts (AUFC) are visualized for both malicious and benign executables. Having charts of this kind built, we can claim that there is a positive correlation between n-gram size and area. Another natural finding is that the more features we have - the more samples we can cover with them, this brings us better accuracy. Also, we found, that AUFC for memtrace sequence length of 100,000 keeps growing on the n-gram size from 24 to 48, while for other lengths growth is almost stopped. We should notice that AUFC for malicious features is bigger than similar areas for benign features. This means that benign files are more different from each other in terms of n-grams than malicious ones. It can be explained as benign executables were made for bigger variety of purposes, while malicious are aimed at performing malicious *activity*.

From one point of view AUFC is a good measure for estimation of feature selection efficiency, but since ML algorithms work with presence bitmap, it is better to use another measure, which will directly show quality of extracted data. As it was said earlier, presence bitmap is a matrix of zeros and ones, so, if it has many zeros it will be hard to generalize data, hence ML algorithms will produce lower accuracy.



(a) Benign class (b) Malicious class

Figure 4: Area under class-wise frequency chart for 200 features

Let's use *sparseness* as a measure of zeros percentage in matrix. Sparseness is a ratio between numbers of zeros in matrix to number of cells and could be expressed as $sparseness = \frac{number\ \ of\ \ zeros}{width\ \ of\ \ matrix\ \cdot\ length\ \ of\ \ matrix}$.

Moreover, we also performed sparseness calculations for all bitmaps. Using sum of benign and malicious AUFC and sparseness measures for all bitmaps, we built charts in Figure 5 for 800 features. As it can be seen from this chart, sparseness of presence bitmap is descending, while overall AUFC grows, proving earlier statement.

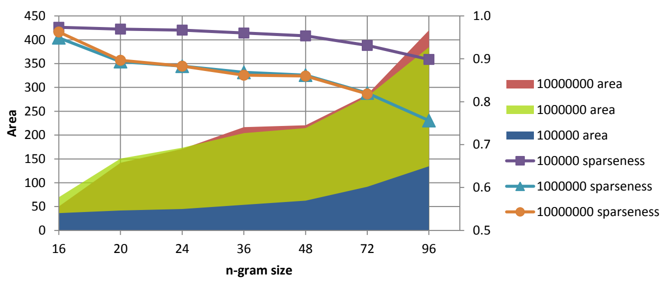


Figure 5: Area under class-wise frequency chart for 800 features

5 Discussions & Conclusion

This work targets malware detection using memory access patterns based on a sequence of read and write operations, also called memtraces. From the literature, we can see that many authors target dynamic malware analysis due to comprehensiveness of the collected behavioural features, including and not limited to disk, network and memory patterns. Yet, only a few consider memory access operations as reliable sources for malicious activities identifiers. We believe that memtraces can be highly relevant for proactive malware analysis. This is because it is the result of opcode execution, which produces consistent operations, yet may slightly vary for different arguments. We proposed a method for fast malware identification according to a presence or non-presence of a specific read and write pattern in its memory access sequence. For our experiments we used $10^5, \ldots, 10^7$ memtraces and 200-800 features extracted using $12, \ldots, 96$ n-gram size. It was found that 10^6 memtraces with 800 features and 96-grams give a robust classification accuracy up to 98.92% using ML methods. In addition to this, we studied a range of aspects and found that such method reveals a set of useful statistical properties that can be further applied for threat identification and ML-based malware detection. We believe that our work will contribute to proactive malware analysis in future.

References

[1] “Virusshare.com,” http://virusshare.com/, accessed: 28.02.2016.

[2] M. Schiffman, “A brief history of malware obfuscation,” 2010.

[3] D. Distler and C. Hornat, “Malware analysis: An introduction,” *Sans Reading Room*, 2007.

[4] K. Kendall and C. McMillan, “Practical malware analysis,” in *Black Hat Conference, USA*, 2007.

[5] D. Uppal, R. Sinha, V. Mehra, and V. Jain, “Malware detection and classification based on extraction of api sequences,” in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on.* IEEE, 2014, pp. 2337–2342.

[6] L. S. Grini, A. Shalaginov, and K. Franke, “Study of soft computing methods for large-scale multinomial malware types and families detection,” in *Proceedings of the The 6th World Conference on Soft Computing*, 2016.

[7] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[8] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, “The cuckoo sandbox,” 2012.

[9] J. Yang, J. Deng, B. Cui, and H. Jin, “Research on the performance of mining packets of educational network for malware detection between pm and vm,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2015 9th International Conference on*, July 2015, pp. 296–300.

[10] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, “On the trustworthiness of memory analysis #x2014;an empirical study from the perspective of binary execution,” *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 557–570, Sept 2015.

[11] Y. Kawakoya, M. Iwamura, E. Shioji, and T. Hariu, *Research in Attacks, Intrusions, and Defenses: 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. API Chaser: Anti-analysis Resistant Malware Analyzer, pp. 123–143.

[12] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, “Ensemble learning for low-level hardware-supported malware detection,” in *Research in Attacks, Intrusions, and Defenses.* Springer, 2015, pp. 3–25.

[13] “Pin a dynamic binary instrumentation tool intel developer zone 2012,” https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2012, accessed:2016-4-14.

[14] M. Ligh, S. Adair, B. Hartstein, and M. Richard, *Malware analyst’s cookbook and DVD: tools and techniques for fighting malicious code.* Wiley Publishing, 2010.

[15] V. FOUNDATION, “Volatility,” http://www.volatilityfoundation.org/, 2015, accessed:2016-4-15.

[16] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, “Robust signatures for kernel data structures,” in *Proceedings of the 16th ACM conference on Computer and communications security.* ACM, 2009, pp. 566–577.

[17] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, “Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures.” in *NDSS*, 2011.

[18] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, “A software birthmark based on dynamic opcode n-gram,” in *Semantic Computing, 2007. ICSC 2007. International Conference on.* IEEE, 2007, pp. 37–44.

[19] A. Fog, “Technical university of denmark.” *Instruction tables Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2016.

[20] 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.

[21] P. Vinod, V. Laxmi, and M. S. Gaur, “Reform: Relevant features for malware analysis,” in *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on.* IEEE, 2012, pp. 738–744.

[22] I. Kononenko and M. Kukar, *Machine learning and data mining: introduction to principles and algorithms.* Horwood Publishing, 2007.

[23] Netmarketshare, “Desktop operating system market share,” https://www.netmarketshare.com/operating-system-market-share.aspx, 2016, accessed: 2016-5-10.

[24] G. Amato, “Peframe,” https://github.com/guelfoweb/peframe, accessed: 20.10.2015.