Detection of running malware before it becomes malicious *

Sergii Banin[1] and Geir Olav Dyrkolbotn[1]

Department of Information Security and Communication Technology, NTNU, Gjøvik, Norway

Abstract. As more vulnerabilities are being discovered every year[17], malware constantly evolves forcing improvements and updates of security and malware detection mechanisms. Malware is used directly on the attacked systems, thus anti-virus solutions tend to neutralize malware by not letting it launch or even being stored in the system. However, if malware is launched it is important to stop it as soon as the maliciousness of a new process has been detected. Following the results from [8] in this paper we show, that it is possible to detect running malware before it becomes malicious. We propose a novel malware detection approach that is capable of detecting Windows malware on the earliest stage of execution. The accuracy of more than 99% has been achieved by finding distinctive low-level behavior patterns generated before malware reaches it's entry point. We also study the ability of our approach to detect malware after it reaches it's entry point and to distinguish between benign executables and 10 malware families.

Keywords: Malware detection · Low-level features · Hardware-based features · Information security · Malware analysis · Malware classification

1 Introduction

Every year our society becomes more dependent on computers and computer systems, thus attacks on the personal, industry and infrastructure computers start having more severe consequences [23][26]. According to NIST the amount of vulnerabilities discovered every year has grown almost 3 times during the years 2015-2019 [17]. At the same time a number of vulnerabilities found on Windows platforms has shown 10% growth[18]. Furthermore, the amount of newly discovered Windows malware has grown 30% during the same period[4]. Such security landscape outlines the need for updates in existing and invention of new malware detection mechanisms.

Malware detection methods can be divided based on which features of malware they use for detection: static and dynamic. Static features emerge from the properties of an executable files themselves: file header, opcode and byte n-grams or hashes are known to be used for malware detection[24]. Dynamic

* The research leading to these results has received funding from the Center for Cyber and Information Security, under budget allocation from the Ministry of Justice and Public Security of Norway

features represent the behavior of malware when it runs and can be roughly divided into high- and low-level features[8]. API and system calls, network and file activity are some of the high-level features, while memory access operations, opcodes or hardware performance counters are the low-level features. Basically we perceive behavioral features that emerge from the system's hardware as the low-level ones[7][13][20]. Static features are easier to change for an attacker utilizing techniques such as obfuscation or encryption. However, malware becomes malicious only when it is executed and it is impossible to avoid a behavioral footprint[10]. Even though different techniques such as polymorphism, anti-VM or anti-debug might be used to change high-level behavioral patterns, the functionality of malware remains similar. Moreover, as soon as malware is launched - it is impossible to avoid execution on the system's hardware. That's why in this paper we use low-level features such as memory access patterns for malware detection[9] and classification[7].

Memory access patterns previously were proven to be effective features for malware detection[9] and classification[7]. A memory access pattern is a sequence of read and write operations performed by an executable and will be described in details in Section 3. The problem with low-level features is that it is hard for a human analyst to understand the context under which a certain pattern has occurred. A previous work [8] presented an attempt to fill the gap between low-level activity (memory access patterns) and its high-level (more human understandable API calls) representation. During the study it was also found, that under the experimental design used in [8] and [7] most of the recorded behavioral activity emerged not from the main module of an executable (after the Entry Point[1] - AEP) but prior to the moment when instruction pointer (IP) is set to the Entry Point (before the Entry Point - BEP). Without going into much details (see Section 2 for details) these findings showed, that it is potentially possible to detect running malicious executable before it starts executing the logic that was put into it by the creator.

To study these findings, in this paper we use a novel approach in behavioral malware analysis. This approach involves analysis of behavioral traces divided into those generated BEP and those generated AEP: *BEP-AEP approach*. More specifically, we show how memory access patterns can be used for malware detection based on the activity produced BEP. To be consistent in our studies we also compare these results to those achieved based on the activity produced AEP: by the malicious code itself. As paper [7] showed a possibility to classify malware into categories (families or types) using memory access patterns, further we investigate the usefulness of *BEP-AEP approach* for distinguishing between benign executables and different malware families. In order to formalize our future findings we propose the following hypotheses:

Hypothesis 1. *It is possible to detect (distinguish from benign) running malicious executable based on the memory access patterns it produces before it begins to execute malicious code (BEP).*

[1] In this paper, by Entry Point, we mean the first executed instruction from the main module of executable.

Hypothesis 2. *It is possible to detect (distinguish from benign) running malicious executable based on the memory access patterns it produces after its Entry Point (AEP).*

And as the logic put into the executable (and makes malware malicious) normally runs AEP we had another hypothesis:

Hypothesis 3. *If Hypotheses 1, 2 are true, then it should be easier (higher classification performance) to detect running malicious executable AEP than BEP.*

To test whether a *BEP-AEP approach* can be used to distinguish between benign and several different categories of malicious executables we had another three hypotheses (directly derived from Hypotheses 1, 2 and 3)

Hypothesis 4. *It is possible to distinguish between several malware categories and benign executables based on the memory access patterns they produce BEP.*

Hypothesis 5. *It is possible to distinguish between several malware categories and benign executables based on the memory access patterns they produce AEP.*

Hypothesis 6. *If Hypotheses 4, 5 are true then it should be easier (higher classification performance) to distinguish between several malware categories and benign executables based on the memory access patterns they produce AEP.*

In order to check the above mentioned hypotheses we decided to perform a series of experiments that consist of several parts. First, we record memory access patterns produced by executables before and after entry point with help of dynamic binary instrumentation framework Intel Pin[12]. Second, we perform feature construction and selection to create different feature vectors. Last, we train several machine learning (ML) algorithms to check our hypotheses by looking at classification performance of machine learning models.

The remainder of the paper is arranged as following: Section 2 provides a literature overview, Section 3 describes our choice of methods, Section 4 explains our experimental setup, in Section 5 we provide results and analyze them, in Section 6 we discuss our findings and in the Section 7 we provide conclusions.

2 Related works

In this section we provide an overview of papers that are related to this article in terms of features used for malware detection as well as methods to extract those features. The first paper we would like to mention is [3] where authors suggested to use Intel Pin based tool to detect malicious behavior by matching it against predefined security policies. Authors record execution flow of executables and describe it by splitting into basic blocks with additional information about each basic block. Among the different sources of information of the basic blocks they used: file modification system calls, fact of presence of *exec* function call and the fact of presence of memory read and write operations. During the testing phase they managed to achieve average path coverage of more than 93%

which later helped them to get as much as 100% detection rate on Windows and Linux systems. Even though their datasets were relatively small this work showed promising capabilities of Intel Pin in the malware research.

The next paper [5] focuses more on the low-level features and their use in malware detection. As the features they used retired and mispredicted branch instructions as well as retired load and stored instructions derived from hardware performance counters. Authors achieved classification precision of more then 90%. Their dataset was also relatively small, but they pointed to the effectiveness of low-level feature in malware detection. Later, the same authors expanded their approach by using additional low-level features (near calls, near branches, cache misses etc.) in the paper [6]. They have also expanded their task to multinomial classification of benign and malicious samples divided into several families. This time they achieved 95% precision on a bigger dataset, what, once again, showed capabilities of low-level features use in malware detection and classification.

In [14] another example of application of hardware-based features is proposed. Extending their work from [19], authors propose hardware malware detector that uses several low-level features such as: frequencies and presence of opcodes from different categories, memory reference distance, presence of a load and store operations, amount of memory reads and writes, unaligned memory accesses as well as taken and immediate branches. Using ensemble specialized and ensemble classifiers authors achieved classification and detection accuracy of around 90% and 96% respectively.

Papers [3][5][14] used information about memory access operations but they didn't use sequences, or patterns, of memory access operations. The first paper where memory access patterns were used for malware detection was [9]. There authors explored a possibility of malware detection based on n-grams of memory access operations. They recorded sequences of memory access operations from malicious and benign executables. After the experiments authors found, that with n-gram size of 96 it is possible to achieve malicious against benign classification accuracy of up to 98%. Later, the same authors explored possibility of a use of memory access n-grams for malware classification [7]. They tested their approach on two datasets label into malware *types* and *families* respectively. After the feature selection they went down to as low as 29 features which allowed them to classify malware types with accuracy of 66% and families with accuracy of 78%. This performance was not as good as pure malicious against benign classification. However, for 10-class classification problem such accuracy showed that this methods (with certain limitations) can be used for malware classification as well. During their studies authors discovered a following problem: memory access patterns provide little context to a human analyst as it is almost impossible to understand which part of the execution flow created a distinctive memory access pattern. To eliminate this knowledge gap, in their next paper [8] they performed an attempt to "correlate" memory access patterns (as low-level features) with API calls (as high-level features). Together with memory access operations they recorded API calls performed by malicious executables.

In the end their attempt was not successful: with their methodology they were not able to find any significant "correlation" between memory access patterns and API calls. However, as those events were proven to be independent they showed, that combining API calls and memory access patterns into integrated feature vector results into increased classification performance. On the dataset from [7] they managed to show increased classification accuracy of 70% and of 86% for malware types and families respectively. It was in this paper where they discovered, that most of the behavioral activity they recorded originated from BEP and outlined a need for additional study of such finding.

To the best of our knowledge no one has analyzed the possibility of malware detection and classification based on activity generated BEP. Therefore we think that our paper provides a novel contribution and grounds for further research.

3 Methodology

This section describes the methods used in our work. We begin with a description of the process creation flow on Windows. It has multiple stages and it is important to show where we begin to record a behavioral trace: a set of opcodes with their memory access operations, current function and module name. Second, we explain the way we transform a behavioral trace into the memory access patterns that are later used as features for training the ML models. We also describe how we perform a feature selection. Last, we provide a description of ML methods and evaluation metrics.

3.1 General overview

As we present BEP-AEP approach in this paper, we have to provide a brief description of a process creation flow the way it is implemented in Windows. The flow of process creation consists of several stages (as described in Windows Internals [28]) and is depicted on the Fig. 1. First, the process and thread objects are created. Then a Windows Subsystem Specific process initialization is performed. Lastly, the execution of the new process begins from the Final Process Initialization (Stage 7 on Fig. 1). During these stages OS initializes a virtual address space that is later used by a process. Virtual address space is divided into private process memory and protected OS memory. The size of virtual address space depends on the OS type. Normally 32-bit Windows will have up to 4GB while 64-bit - up to 512GB of virtual address space. The virtual address space contains heap, stack, loaded DLLs, kernel and code of the executable (main module). CPU executes instructions (opcodes) from main module or one of the loaded libraries. Each opcode can be divided into several microoperations. Some microoperations are used for arithmetical-logical operations while some are responsible for memory read and write operations. Whenever execution of an opcode requires a memory related microoperation to be executed, Intel Pin tool will record this into the behavioral trace. Intel Pin tool begins to record the behavioral trace at Stage 7, when a new process is started. In the context of a

newly created process, Stage 7 generates a BEP activity and includes (but is not limited to) the following actions: installing of exception chains; checking if the process is debugee and whether prefetching is enabled; initialization of image loader, heap manager; loading of all the necessary DLLs. When it is finished, AEP activity begins from execution of Entry Point in the main module. Some malware samples might use packing, thus will unpack itself in the beginning of its execution. However, it is important to understand, that unpacking will be done with instructions from the main module of executable. Thus, with our approach, unpacking will happen AEP.

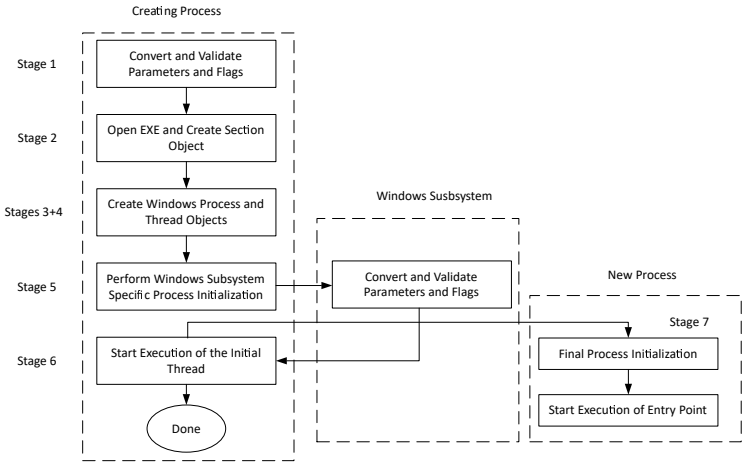

Fig. 1. Process creation flow [28]

3.2 Data collection

In this subsection we describe the way we record behavioral traces. In order to record memory access traces we wrote a custom tool that was based on the Intel Pin framework[12]. Intel Pin is a binary instrumentation framework that allows to intercept execution flow of a process and extract much of the information related to this process such as: memory access operations, opcode, name of a module from which an opcode is being executed and name of a current routine (if possible to derive). Every executable from our dataset (Section 4.1) was launched together with the Intel Pin tool. The tool records all the data mentioned above into the behavioral trace. The process of each executable was observed from the beginning of its execution. We recorded the behavioral trace until we gathered 1,000,000 (1M) of memory access operations (similar to [9] and [7]) BEP, and then we continued recording AEP - again until we reached 1M of memory access operations. As we worked with real-life malicious and benign executables, we were not always able to record the desired amount of memory access operations. Some samples reached main module before producing the

desired 1M of memory access operations BEP, while some finished their work before producing 1M of memory access operations AEP. It is worth mentioning, that some samples didn't produce any traces AEP. All data collection was done in the Virtual Box virtual machine (VM) in order to protect the host system, allow automation and ensure equal launch conditions for all executables.

3.3 Feature construction and selection

Before using our data for training the ML models we have to construct and select features. Memory access operations (BEP or AEP respectively) are concatenated into memory access sequence. Based on the methods used in [8] we split memory access sequence produced by an executable into a set of subsequences: n-grams of the length 96. These n-grams are overlapping, so every next n-gram begins from the second element of the previous one. A typical memory access n-gram looks the following way: *RRRWWWWR...WRRRRRRRW*. If we treat *R* as *0* and *W* as *1* n-gram of a size *n=96* becomes binary sequence with potential feature space of 2^{96}. Even though we do not get this amount of distinctive features, our samples still produce millions of features (see Section 5). So we need to perform feature selection in order to reduce feature space, reject uninformative features and be able to train ML models in a feasible time. Smaller feature set also contributes for better understanding of the findings and allows "manual" analysis if necessary[7][8].

The feature selection is performed in two steps. On the first step we go down from millions of features to 50,000 by using Information Gain feature selection method. Information Gain (IG) is an attribute quality measure that reflects "the amount of information, obtained from the attribute A, for determining the class C"[15] and is calculated as following:

$$Gain(A) = -\sum_{k} p_k \log p_k + \sum_{j} p_j \sum_{k} p_{k|j} \log p_{k|j}$$

where p_k is the probability of the class k, p_j is the probability of an attribute to take j_{th} value and $p_{k|j}$ is the conditional probability of class k given j_{th} value of an attribute. On the second step we use Correlation-based feature selection (CFS)[11] from Weka[2] package (CfsSubsetEval). This method selects a subset of features based on the maximum-Relevance-Minimum-Redundancy principle by selecting features that have maximal relevance for representing the target class and minimal mutual correlation[21]. The reason we did not apply this method to the full feature set is computational complexity. In order to perform CFS feature selection one needs to calculate correlation matrix between all features which would require infeasible amount of computational resources and time. We also select 5,10,15 and 30 thousands of features with IG. It is important to know, that CFS adds features to the feature set until further increase of its merit is no longer possible. Thereby, in the end we use IG to select the same amount of features as was selected by CFS. By doing so we can directly compare performance of two feature selection methods. After the feature selection process

we create data that is later used to train ML models. Basically we generate a table, where each row represents values that features from the feature set take for a certain sample. In our paper similarly to [9] we use *1* if feature (memory access n-gram) is generated by sample and *0* if not.

3.4 Machine Learning methods and evaluation metrics

We use Weka[2] machine learning toolkit to build and evaluate our models. Similarly to [8] we choose the following ML methods to build our models: k-Nearest Neighbors (kNN), RandomForest (RF), Decision Trees (J48), Support Vector Machines (SVM), Naive Bayes (NB) and Artificial Neural Network (ANN) with the default for Weka[2] package parameters. To evaluate quality of models we use 5-fold cross validation[15] and choose the following evaluation metrics for models assessment: accuracy (ACC) as number of correctly identified samples and F1-measure (F1M) which takes into account precision and recall. We omit using False Positives measure as it is not representative for multinomial classification. The F1M values presented in Section 5 are average weighted. For the benign against malicious classification our dataset is nearly balanced (see Subsection 4.1), however while doing multinomial classification we had to deal with imbalanced classes. The problem with imbalanced classes is that evaluation metrics does not reflect real quality of models, since simple guessing on the majority class will give high accuracy. To deal with this problem we apply weights to the samples, so that sums of the weights of samples within each class would be equal.

4 Experimental setup

In this section we describe our dataset, experimental environment and experimental flow.

4.1 Dataset

As Windows is the most popular desktop platform [16] we focused on analyzing Windows malware. Our dataset consists of two parts: malware samples and benign samples. Benign samples were collected from Portable Apps [22] in September 2019. It is a collection of free Portable software that includes various types of software such as graphical, text and database editors; games; browsers; office, music, audio and other types of Windows software. In total we obtained 2669 PE executables. Malicious samples were taken from *VirusShare_00360* pack downloaded from VirusShare[1]. *VirusShare_00360* contained 65518 samples, out of which 2973 were PE executables. For each sample we downloaded a report from VirusTotal[27] and left samples that belonged to the 10 most common families. Those families are: Fareit, Occamy, Emotet, VBInject, Ursnif, Prepscram, CeeInject, Tiggre, Skeeyah, GandCrab. According to the VirusTotal reports, resulted samples were first seen (first submission date) between March 2018 and

March 2019. Not all the samples were launched successfully, and from those that launched not all the samples produced traces AEP (most likely executables lacked some resources, e.g. certain libraries). So the amounts of samples that generated traces BEP and AEP are different. In the Table 1 we present amount of samples of each category that produced traces BEP and AEP.

Table 1. Amount of samples that generated traces BEP and AEP.

	Benign	Malicious	Fareit	Occamy	Emotet	VBInject	Ursnif	Prepscram	CeeInject	Tiggre	Skeeyah	GandCrab
BEP	2098	2005	573	307	196	164	162	143	127	117	115	101
AEP	1717	1755	573	174	188	162	161	143	115	69	73	97

4.2 Experimental environment

For our experiments we used Virtual Dedicated Server with 4-cores Intel Xeon CPU E5-2630 CPU running at 2.4GHz and 32GB of RAM with Ubuntu 18.04 as a main operating system. As a virtualization software we used VirtualBox 6.0.14. We created a Windows 10 VM and disconnected it from the Internet. We have uploaded Intel Pin togther with our custom tool into the VM. We also disabled all built-in anti-virus features to make malware run properly and also because they kept interrupting the work of Intel Pin and created a base snapshot which was used for all experiments. We controlled the VM and data collection process with Python 3.7 scripts.

4.3 Experimental flow

During the data collection phase we begin with starting up a VM. Then we upload an executable to the VM and launch it together with Intel Pin tool. When a behavioral trace is ready we download it from the VM and begin a new experiment with reverting a VM to the base snapshot. It is important to notice that benign executables were uploaded together with their folder. This allowed more of the benign applications to run properly and helped to emulate a more real-life scenario, where benign applications often come with various additional resources they need for normal operations.

5 Results and Analysis

In this section we provide the classification performance of ML models performed under different conditions. We also analyze the results and show how they align with the Hypotheses from Section 1.

5.1 Classification performance

Each table contains performance metrics of ML methods (Subsection 3.4) achieved with a feature sets (Subsection 3.3) of a different length (*FSL* stands for feature set length). Some of the cells contain missing values: due to processing limitations of Weka we were not able to obtain all of the results.

In the Tables 2 and 3 the results of malicious against benign classification BEP and AEP are presented. As we can see, under our experimental design it is possible to achieve classification accuracy of 0.999 for BEP and 0.992 for AEP with 10000 features. CFS selected 9 features for BEP and 39 for AEP. Classification performance with use of CFS-selected features is slightly lower than the best result achieved with those selected by IG. At the same time, it is often higher for the same amount of features selected by IG.

Table 2. Malicious vs Benign BEP classification performance.

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	0.996	0.996	0.996	0.996	0.997	0.997	0.983	0.983	0.693	0.671	-	-
	30K	0.996	0.996	0.997	0.997	0.998	0.998	0.986	0.986	0.983	0.983	-	-
	15K	0.996	0.996	0.998	0.998	0.998	0.998	0.991	0.990	0.983	0.983	-	-
	10K	0.998	0.998	**0.999**	**0.999**	0.998	0.998	0.992	0.991	0.983	0.983	-	-
	5K	0.995	0.995	0.997	0.997	0.997	0.997	0.988	0.988	0.983	0.983	-	-
	9	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988
CFS	9	0.997	0.997	0.997	0.997	0.996	0.996	0.997	0.997	0.988	0.988	0.997	0.997

Table 3. Malicious vs Benign AEP classification performance.

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	0.974	0.974	0.985	0.985	0.991	0.991	0.948	0.948	0.735	0.720	-	-
	30K	0.982	0.982	0.990	0.990	0.989	0.989	0.949	0.949	0.795	0.787	-	-
	15K	0.990	0.990	**0.992**	**0.992**	0.988	0.988	0.947	0.947	0.795	0.787	-	-
	10K	0.990	0.990	**0.992**	**0.992**	0.989	0.989	0.955	0.955	0.795	0.787	-	-
	5K	0.989	0.989	0.991	0.991	0.988	0.988	0.960	0.960	0.795	0.787	-	-
	39	0.910	0.909	0.910	0.909	0.908	0.908	0.907	0.906	0.844	0.840	0.910	0.909
CFS	39	0.990	0.990	0.990	0.990	0.989	0.989	0.987	0.987	0.982	0.982	0.991	0.991

In the Tables 4 and 5 we present performance of ML models in classifying benign and 10 malicious families using features generated BEP and AEP. In these tables we show classification performance for the imbalanced (*Imb*) and balanced datasets (*Bal*) (Subsection 3.4). As we can see, performance of multinomial classification is lower than the benign against malicious classification. By using BEP and AEP features we achieved 0.605 and 0.749 classification accuracy

respectively. The main observation that can be derived from these tables is that it is easier to distinguish between benign executables and 10 malware families using features generated AEP than BEP. As the number of samples that pro-

Table 4. 10 Malicious families vs Benign BEP classification performance.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.812	0.776	0.812	0.775	0.811	0.771	-	-	0.429	0.462	-	-
		Bal	0.601	0.546	**0.605**	0.549	0.596	0.541	-	-	0.403	0.326	-	-
	30K	Imb	0.812	0.777	0.813	0.776	0.808	0.769	0.789	0.740	0.687	0.667	-	-
		Bal	0.598	0.542	0.600	0.543	0.594	0.538	0.549	0.477	0.433	0.355	-	-
	15K	Imb	0.813	0.777	0.815	0.777	0.811	0.772	0.792	0.745	0.689	0.668	-	-
		Bal	0.594	0.538	0.596	0.540	0.594	0.539	0.565	0.501	0.435	0.355	-	-
	10K	Imb	0.813	0.775	0.814	0.776	0.809	0.770	0.798	0.753	0.689	0.668	-	-
		Bal	0.589	0.531	0.593	0.535	0.590	0.531	0.569	0.502	0.435	0.356	-	-
	5K	Imb	0.789	0.745	0.790	0.745	0.789	0.743	0.782	0.728	0.633	0.591	-	-
		Bal	0.508	0.446	0.513	0.452	0.512	0.446	0.492	0.413	0.382	0.301	-	-
	92	Imb	0.653	0.575	0.653	0.575	0.653	0.575	0.652	0.571	0.651	0.567	0.652	0.573
		Bal	0.184	0.140	0.185	0.140	0.183	0.137	0.182	0.135	0.180	0.128	0.170	0.136
CFS	92	Imb	0.813	0.775	0.813	0.775	0.810	0.769	0.805	0.760	0.740	0.725	0.810	0.771
		Bal	0.585	0.526	0.585	0.527	0.578	0.529	0.572	0.512	0.521	0.467	0.576	0.540

duced traces BEP and AEP is different we have also tested the performance of features from BEP on the normalized dataset, when we only take into account samples that produced traces AEP. These results are present in the Appendix Appendix A. We also combined features produced BEP and AEP and tested classification performance of the combined feature set. These results presented in the Appendix Appendix B.

5.2 Analysis

From the results presented in Tables 2 and 3 we can conclude that both Hypotheses 1 and 2 are supported: we can distinguish between malicious and benign behavior BEP and AEP. However, even if there is a visible decline in accuracy when switching from AEP behavior to BEP it is relatively low. Thus, we are not able to conclude that our approach allows to detect malware BEP better then AEP or vise versa. Thereby, we were not able to support or reject Hypothesis 3. This may be a reflection of property of our dataset or a limitation of our approach, and therefore needs further investigation in the future work.

By looking at the numbers of features selected by CFS we can see, that it selects more features for AEP data than for BEP data. And it's not surprising, since the behavior of executables become more diverse AEP: this is where their internal logic starts being executed. It is also confirmed by the amount of unique features produced by the samples BEP and AEP. Malicious samples produced more than 1M features BEP, and more than 7M features AEP. On the

Table 5. 10 Malicious families vs Benign AEP classification performance.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.890	0.883	0.902	0.890	0.897	0.888	-	-	0.433	0.420	-	-
		Bal	0.715	0.694	0.724	0.714	**0.749**	0.737	-	-	0.503	0.418	-	-
	30K	Imb	0.891	0.883	0.891	0.883	0.898	0.888	0.727	0.635	0.505	0.527	-	-
		Bal	0.725	0.705	0.732	0.714	0.729	0.708	0.539	0.498	0.493	0.414	-	-
	15K	Imb	0.889	0.881	0.900	0.891	0.898	0.887	0.780	0.712	0.508	0.530	-	-
		Bal	0.724	0.704	0.731	0.714	0.723	0.702	0.625	0.589	0.499	0.420	-	-
	10K	Imb	0.887	0.878	0.900	0.891	0.897	0.886	0.805	0.756	0.509	0.530	-	-
		Bal	0.717	0.695	0.725	0.706	0.729	0.711	0.645	0.606	0.497	0.418	-	-
	5K	Imb	0.866	0.851	0.872	0.854	0.865	0.848	0.747	0.669	0.384	0.384	-	-
		Bal	0.660	0.618	0.661	0.619	0.653	0.605	0.504	0.442	0.433	0.342	-	-
	40	Imb	0.694	0.615	0.694	0.616	0.693	0.613	0.688	0.604	0.670	0.597	0.693	0.617
		Bal	0.306	0.212	0.307	0.206	0.302	0.204	0.299	0.217	0.298	0.193	0.283	0.225
CFS	40	Imb	0.902	0.896	0.903	0.892	0.891	0.880	0.889	0.873	0.872	0.864	0.900	0.890
		Bal	0.725	0.701	0.726	0.704	0.717	0.695	0.706	0.667	0.695	0.653	0.722	0.692

other hand, benign applications produced more than 4.5M of features BEP and almost 20.5M AEP. This resulted in more then 5M unique features to choose from for BEP classification, and 25M for AEP classification. This also shows, that benign applications are more diverse and produce more distinctive memory access patterns as a result of a more distinctive behavior. And it makes sense, since malware samples belong to 10 malware families, thus should share more common properties according to the definition of malware family from [7].

The results of multinomial classification (Tables 4 and 5) are more diverse then those for malicious against benign classification. This time, it is clearly easier to distinguish between 11 classes AEP than BEP. Even though multinomial classification accuracy BEP is not that impressive it is still significantly better than potential accuracy of 0.09(09) that can be achieved by random guessing. Thus we can conclude, that Hypothesis 4 is supported. Multinomial classification accuracy AEP was significantly better. So we can conclude that Hypothesis 5 is also supported, thereby Hypothesis 6 as well.

This time CFS has chosen less features for the AEP classification than for the BEP classification. As we mentioned above, malware assigned to one of the families based on its particular functionality. And this functionality becomes revealed AEP. Thereby it is logical to say, that classification of 11 classes is more accurate based on the behavior generated AEP. Table 6 present combined results of the Hypotheses evaluation.

Table 6. Evaluation of Hypotheses after analyzing the results

	H 1	H 2	H 3	H 4	H 5	H 6
Supported	**Yes**	**Yes**	-	**Yes**	**Yes**	**Yes**

6 Discussion

In this section we present an attempt to interpret our findings. Earlier, we showed the possibility of malware detection based on the memory access patterns generated BEP. So, we wanted to find an explanation of why the BEP activity of malicious and benign executables is so different. More specifically we wanted to see which high-level activity is responsible for generating specific memory access patterns. As it was written in Subsection 3.2, we recorded not only memory access operations, but also routine names for each executed opcode. Since BEP activity happens in the Windows libraries (Subsection 3.1) we are always able to derive a name of a current routine. Thereby, a memory access pattern can be represented as a sequence of routine names. However, our memory access patterns are of a length 96, so having 96 routine names (many of which are repetitive) makes analysis harder and adds redundant information. Thus, we decided to represent each memory access pattern as a sequence of unique routine names. For example, if memory access pattern begins in a routine RTN_1, proceeds into the RTN_2 and finishes in the RTN_1 we store the following sequence: {*RTN_1,RTN_2,RTN_1*}. After performing this search on the 9 features selected by CFS (Subsection 5.1) we made a surprising discovery: most of these features originated in *RtlAllocateHeap* routine from the *ntdll.dll* Windows library. Some memory access patterns were completely generated by *RtlAllocateHeap*, while others involved other routines as well. The same memory access pattern can be found in different routine sequences. However, similar to [8], this is the result of our patterns structure and feature construction method (e.g. they can start and end with a sequence of repetetive *W's* or *R's*) that allow similar pattern to appear multiple times in a row. For example, one feature can be found in the following sequences: {*RtlAllocateHeap*}, {*bsearch,RtlAllocateHeap*}, {*LdrGetProcedureAddressForCaller,RtlAllocateHeap*}, {*RtlEqualUnicodeString, RtlAllocateHeap*}. The *RtlAllocateHeap* routine is responsible for allocating a memory block of a certain size from a heap. Thus, when the Final Process Initialization phase of process creating flow needs to allocate a memory block it produces a distinctive activity that allows to distinguish between malicious and benign processes on the stage of initialization. Unfortunately, we were not able to explain why this memory allocation activity can be so distinctive. Neither the official Microsoft documentation on *RtlAllocateHeap*, nor the Windows Internals book[28] gives enough details about memory allocation routines. To answer this question, one may need to revers engineer *ntdll.dll* library and perform a Kernel-level[25] debugging. And we leave it for the future work, as this is out of scope of this paper.

7 Conclusions

In this paper we presented a novel dynamical malware analysis approach, where we distinguish between activity produced before and after Entry Point. As we were able to show, it is possible to distinguish between malicious and benign

executables BEP with accuracy of up to *0.999* with 10000 features, and up to *0.997* with just 9 features. It means, that it is possible to detect malicious executables on the stage of their launch: before they become malicious. We also found, that distinguishing between benign samples and samples from 10 malware families is also possible using BEP activity. We have also made an interesting discovery: many of the memory access patterns used for malware detection BEP are generated by the *RtlAllocateHeap* routine. This paper shows a need for further research of the low-level activity use in malware analysis. First of all, we need to make a complete explanation of why the BEP activity of malicious and benign executables are that different. Second, we have to check the robustness of this approach against the previously unknown malware. Lastly, to fully utilize the capabilities of BEP-AEP approach we need to study the possibility of building the real-time system that uses our approach. This will involve assessment of computational overhead and potential impact on the user experience.

Appendix A Classification results: normalized dataset

Here we present classification results for the normalized dataset using features from BEP.

Table 7. Malicious vs Benign BEP classification performance on the normalized dataset.

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50000	0.997	0.997	0.996	0.996	0.998	0.998	0.981	0.980	0.750	0.738	-	-
	30K	0.997	0.997	0.997	0.997	0.998	0.998	0.983	0.983	0.981	0.980	-	-
	15K	0.997	0.997	**0.999**	**0.999**	0.998	0.998	0.990	0.990	0.981	0.980	-	-
	10K	0.997	0.997	**0.999**	**0.999**	0.998	0.998	0.990	0.990	0.981	0.980	-	-
	5K	0.995	0.994	0.996	0.996	0.997	0.997	0.988	0.988	0.981	0.981	-	-
	10	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988
CFS	10	0.998	0.998	0.998	0.998	0.997	0.997	0.998	0.998	0.988	0.988	0.997	0.997

Table 8. 10 Malicious families vs Benign BEP classification performance on the normalized dataset.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.819	0.779	0.818	0.776	0.817	0.774	-	-	0.478	0.500	-	-
		Bal	0.586	0.528	0.588	0.528	0.590	0.532	-	-	0.386	0.292	-	-
	30K	Imb	0.817	0.776	0.819	0.777	0.816	0.774	0.798	0.744	0.685	0.659	-	-
		Bal	0.579	0.517	0.587	0.525	**0.591**	**0.536**	0.555	0.491	0.423	0.337	-	-
	15K	Imb	0.815	0.774	0.821	0.779	0.815	0.770	0.799	0.747	0.686	0.662	-	-
		Bal	0.574	0.511	0.587	0.525	0.584	0.522	0.587	0.525	0.428	0.345	-	-
	10K	Imb	0.817	0.776	0.819	0.777	0.817	0.772	0.800	0.749	0.685	0.660	-	-
		Bal	0.576	0.513	0.580	0.517	0.578	0.518	0.569	0.505	0.422	0.335	-	-
	5K	Imb	0.812	0.769	0.815	0.771	0.814	0.770	0.803	0.750	0.631	0.572	-	-
		Bal	0.571	0.505	0.570	0.505	0.570	0.509	0.564	0.502	0.419	0.313	-	-
	52	Imb	0.663	0.579	0.663	0.579	0.663	0.579	0.661	0.575	0.661	0.575	0.661	0.575
		Bal	0.190	0.135	0.189	0.155	0.189	0.133	0.189	0.144	0.188	0.131	0.190	0.146
CFS	52	Imb	0.823	0.782	0.822	0.781	0.817	0.772	0.809	0.760	0.739	0.721	0.823	0.781
		Bal	0.588	0.531	0.584	0.525	0.584	0.525	0.571	0.510	0.507	0.444	0.567	0.529

Appendix B Classification results: combined feature set

Here we present classification results achieved with combined feature set.

Table 9. Malicious vs Benign classification performance on the normalized dataset using combined feature set

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50000	0.996	0.996	0.998	0.998	0.999	0.999	0.981	0.981	0.981	0.980	-	-
	30K	0.996	0.996	**0.999**	**0.999**	0.998	0.998	0.982	0.982	0.981	0.980	-	-
	15K	0.997	0.997	**0.999**	**0.999**	0.998	0.998	0.987	0.987	0.981	0.980	-	-
	10K	0.998	0.998	**0.999**	**0.999**	0.998	0.998	0.989	0.989	0.981	0.980	-	-
	5K	0.999	0.999	**0.999**	**0.999**	0.998	0.998	0.995	0.995	0.981	0.980	-	-
	13	0.988	0.987	0.988	0.987	0.998	0.998	0.988	0.987	0.988	0.987	0.988	0.987
CFS	13	0.997	0.997	0.998	0.998	0.996	0.996	0.996	0.996	0.988	0.988	0.997	0.997

Table 10. 10 Malicious families vs Benign classification performance on the normalized dataset using combined feature set.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.910	0.905	0.917	0.910	0.910	0.906	-	-	0.802	0.771	-	-
		Bal	0.740	0.718	**0.749**	0.726	0.746	**0.736**	-	-	0.508	0.419	-	-
	30K	Imb	0.906	0.900	0.918	0.910	0.910	0.904	0.787	0.750	0.795	0.761	-	-
		Bal	0.744	0.744	0.743	0.722	0.734	0.718	0.518	0.465	0.495	0.403	-	-
	15K	Imb	0.904	0.898	0.917	0.909	0.908	0.902	0.806	0.771	0.908	0.902	-	-
		Bal	0.744	0.723	0.740	0.720	0.737	0.723	0.608	0.570	0.493	0.400	-	-
	10K	Imb	0.903	0.896	0.909	0.901	0.908	0.898	0.799	0.765	0.790	0.753	-	-
		Bal	0.735	0.708	0.729	0.705	0.728	0.707	0.593	0.550	0.486	0.388	-	-
	5K	Imb	0.792	0.763	0.789	0.759	0.790	0.757	0.754	0.710	0.679	0.647	-	-
		Bal	0.535	0.499	0.534	0.498	0.535	0.499	0.440	0.382	0.408	0.311	-	-
	62	Imb	0.663	0.579	0.662	0.577	0.662	0.577	0.662	0.577	0.660	0.569	0.663	0.579
		Bal	0.190	0.134	0.191	0.156	0.190	0.134	0.189	0.133	0.181	0.418	0.185	0.140
CFS	62	Imb	0.915	0.909	0.916	0.909	0.909	0.903	0.896	0.879	0.876	0.862	0.908	0.903
		Bal	0.744	0.722	0.745	0.724	0.739	0.721	0.723	0.691	0.669	0.625	0.743	0.729

References

1. Virusshare.com. http://virusshare.com/, accessed: 09.03.2020
2. Weka: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/ (2019), accessed: 2019-03-312
3. Aaraj, N., Raghunathan, A., Jha, N.K.: Dynamic binary instrumentation-based framework for malware defense. In: Detection of Intrusions and Malware, and Vulnerability Assessment. https://doi.org/10.1007/978-3-540-70542-0_4
4. AVTEST. The independent IT-Security Institute: Malware. https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all (2020)
5. Bahador, M.B., Abadi, M., Tajoddin, A.: Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In: Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on. pp. 703–708. IEEE (2014). https://doi.org/10.1109/iccke.2014.6993402
6. Bahador, M.B., Abadi, M., Tajoddin, A.: Hlmd: a signature-based approach to hardware-level behavioral malware detection and classification. The Journal of Supercomputing **75**(8), 5551–5582 (2019). https://doi.org/10.1007/s11227-019-02810-z
7. Banin, S., Dyrkolbotn, G.O.: Multinomial malware classification via low-level features. Digital Investigation **26**, S107–S117 (2018). https://doi.org/10.1016/j.diin.2018.04.019
8. Banin, S., Dyrkolbotn, G.O.: Correlating high-and low-level features. In: International Workshop on Security. pp. 149–167. Springer (2019). https://doi.org/10.1007/978-3-030-26834-3_9
9. Banin, S., Shalaginov, A., Franke, K.: Memory access patterns for malware detection. Norsk informasjonssikkerhetskonferanse (NISK) pp. 96–107 (2016)
10. Burnap, P., French, R., Turner, F., Jones, K.: Malware classification using self organising feature maps and machine activity data. computers & security **73**, 399–410 (2018). https://doi.org/10.1016/j.cose.2017.11.016
11. Hall, M.A.: Correlation-based Feature Subset Selection for Machine Learning. Ph.D. thesis, University of Waikato, Hamilton, New Zealand (1998)
12. IntelPin: A dynamic binary instrumentation tool (2020)
13. Khasawneh, K.N., Ozsoy, M., Donovick, C., Abu-Ghazaleh, N., Ponomarev, D.: Ensemble learning for low-level hardware-supported malware detection. In: Research in Attacks, Intrusions, and Defenses, pp. 3–25. Springer (2015). https://doi.org/10.1007/978-3-319-26362-5_1
14. Khasawneh, K.N., Ozsoy, M., Donovick, C., Ghazaleh, N.A., Ponomarev, D.V.: Ensemblehmd: Accurate hardware malware detectors with specialized ensemble classifiers. IEEE Transactions on Dependable and Secure Computing (2018). https://doi.org/10.1109/tdsc.2018.2801858
15. Kononenko, I., Kukar, M.: Machine learning and data mining: introduction to principles and algorithms. Horwood Publishing (2007)
16. NetMarkeshare: Operating system market share. https://netmarketshare.com/operating-system-market-share.aspx (2020)
17. NIST: National vulnerability database. https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all (2020)
18. NIST: National vulnerability database: Windows. https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&query=Windows&search_type=all (2020)

19. Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., Ponomarev, D.: Malware-aware processors: A framework for efficient online malware detection. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). pp. 651–661. IEEE (2015). https://doi.org/10.1109/hpca.2015.7056070
20. Ozsoy, M., Khasawneh, K.N., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., Ponomarev, D.: Hardware-based malware detection using low-level architectural features. IEEE Transactions on Computers **65**(11), 3332–3344 (2016). https://doi.org/10.1109/tc.2016.2540634
21. Peng, H., Long, F., Ding, C.: Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. IEEE Transactions on pattern analysis and machine intelligence **27**(8), 1226–1238 (2005)
22. PortableApps.com: Portableapps.com. `https://portableapps.com/apps` (2020)
23. Reuters: Ukraine's power outage was a cyber attack: Ukrenergo. `https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA` (2017)
24. Shalaginov, A., Banin, S., Dehghantanha, A., Franke, K.: Machine learning aided static malware analysis: A survey and tutorial. In: Cyber Threat Intelligence, pp. 7–45. Springer (2018). https://doi.org/10.1007/978-3-319-73951-9_2
25. Sikorski, M., Honig, A.: Practical malware analysis: the hands-on guide to dissecting malicious software. no starch press (2012)
26. The Verge: The petya ransomware is starting to look like a cyber-attack in disguise. `https://www.theverge.com/2017/6/28/15888632/petya-goldeneye-ransomware-cyberattack-ukraine-russia` (2017)
27. Total, V.: Virustotal-free online virus, malware and url scanner. Online: https://www. virustotal. com/en (2012)
28. Yosifovich, P.: Windows Internals, Part 1 (Developer Reference). Microsoft Press (may 2017)