Detection of Previously Unseen Malware using Memory Access Patterns Recorded Before the Entry Point

Sergii Banin*, Geir Olav Dyrkolbotn†,
*†Department of Information Security and Communication Technology
Norwegian University of Science and Technology
* Email: sergii.banin@ntnu.no
† Email: geir.dyrkolbotn@ntnu.no

***Abstract*—Recently it has been shown, that it is possible to detect malware based on the memory access patterns produced before executions reaches its Entry Point. In this paper, we investigate the usefulness of memory access patterns over time, i.e to what extent can machine learning algorithm trained on "old" data, detect new malware samples, that was not part of the training set and how does this performance change over time. During our experiments, we found that machine learning models trained on memory access patterns of older samples can provide both high accuracy and a high true positive rate for the period from several months to almost a year from the update of the model. We also perform a substantial analysis of our findings that may aid researchers who work with malware and Big Data.**

***Index Terms*—information security, malware detection, low-level features, memory access patterns**

I. Introduction

Detection and analysis of malware is one of the important areas in the information security research [1]. Malware analysis can be divided into two categories: static and dynamic analysis. While static analysis uses features derived from the file itself, dynamic makes use of the behavioral traces generated when malware is launched. Behavioral or dynamic features can be categorized into high- and low-level features [2]. Low-level features emerge from the hardware of the system: hardware performance counters, opcodes, or memory access patterns are the low-level or hardware-based features. In our paper we utilize dynamic malware analysis using low-level features (memory access patterns). Malware analysis often involves dealing with Big Data: for example, the database table containing memory access patterns of 4000 executables can take several tens of gigabytes [1]. Thus, in this paper, we show how one can analyze the classification performance results obtained after processing the big amounts of data.

It has been recently shown, that it is possible to detect Windows malware based on the behavioral traces produced before the Entry Point (BEP) [3]. It is an important finding since malware malware stopped upon detection BEP can not harm the system where it was launched. However, the authors of [3] used k-fold cross-validation to assess the performance of the machine learning model used for malware detection. Such approach has some limitations: for example, the feature selection made on the full dataset may affect the validity of the classification performance results. As the new malware samples are detected every day, and the amount of newly discovered malware constantly grows [4] it is important to study, how a novel malware detection approach can handle samples that were not involved in training. Thus, we decided to test how memory access traces produced BEP can be used to detect newer, previously unseen malware. We split our dataset into the train set and several time-arranged test sets which will emulate "aging" of the model. We also update the train set with newer malware and observe changes in the performance of the model. To conduct this study we outlined the following research questions. **RQ1:** Is it possible to use memory access traces recorded BEP to distinguish between previously unseen malicious and benign executables? **RQ2:** How long, since the update, the ML model trained on memory access traces recorded BEP can provide a good detection rate? While answering the RQ1 we expect, that the detection of previously unseen malware *should* be possible with the use of BEP memory access traces. At the same time, while answering the RQ2 we expect, that the classification performance and detection rate should be worse the further away in time a train and test set are from each other. However, the 13 months time span of our dataset may affect the results. We assume a detection rate equal or more than 0.95 to be good. While conducting our studies, we found some unexpected trends in classification performance. Since the amounts of data we worked with were very big, we perform analysis which may pose an interest to other researchers working with malware and Big Data. The remainder of the paper is arranged as follows: in Section II we make a short literature overview; in Section III we describe our methods; in Section IV we describe our experimental setup; we provide the results in Section V; in Section VI we perform an analysis of the findings and outline a need for an additional evaluation which is present in Section VII; in the end, we discuss our findings and provide concluding remarks in Section VIII.

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.

II. Background

Here we present a brief overview of the related literature. Malware detection with the use of memory access patterns was first described in [5] by Banin et al. There it has been shown, that it is possible to distinguish between malicious and benign executables with the accuracy of up to 98%. It was [5] where the required amount of memory access operations (1M) and the size of n-grams (96) were shown to be sufficient for such tasks. The technique proposed in [5] was recently extended by Yucel et al. in [6], where authors used memory access patterns to explore the similarity between different malware categories. Later, Banin et al. in [7] showed, that memory access patterns can be successfully used to classify malware into 10 families and 10 types with an accuracy of 78% and 66% respectively. In that paper, it has also been shown, that one needs very few features to perform this type of classification. However, the memory access pattern by itself does not give any information regarding the functionality of the malware to the human analyst. Thus, in their next paper [2] authors performed an attempt to correlate memory access patterns (low-level features) with API calls (high-level features) to bring more context to the human analysts. During the analysis of findings made in [2] authors found, that most of the memory access patterns they recorded emerged from BEP. These findings lead to another work [3], where authors showed that memory access patterns from BEP can be used to detect malware with an accuracy similar to the one achieved with memory access patterns emerged from after the Entry Point (AEP). In particular, they achieved a classification accuracy of more than 99% when distinguishing between malicious and benign executables with a help of only 9 BEP memory access patterns. To the best of our knowledge, memory access patterns have not been tested against previously unseen malware that was not used to train the model. However, many works provide examples of splitting the malware dataset into train and test sets to emulate the detection of previously unseen malware. In [8] authors randomly selected 50% of the dataset to be used as a train set, while the remainder was used as a test set. On the test set, containing roughly 3K malicious and 2.2K benign executable, they managed to achieve accuracy of 100% with the Random Forest algorithm. Similarly, authors of [9] used around 10K malicious and 2.5K benign samples for training, 750 malicious and 610 benign executables for testing, and achieved up to 89% of accuracy. Authors of [10] split their dataset into train and test sets based on the year when samples were submitted to the VirusTotal [11]. With a train set containing samples from the year 2012, and a test set from the year 2013, they managed to achieve 72% detection rate without a human reviewer and 89% with. The different approach in training and testing was presented in [12]. The authors used a dataset of benign and malicious Android applications from the years 2010-2017. They performed consecutive training on a certain year and testing on the years newer than the one used for training. Their results show, that e.g. precision may both drop or rise as the test set becomes more distant in time from the train set. Similarly, authors of [13] test the performance of their Android malware detection approach on test sets arranged on the monthly basis. In their paper, it is possible to observe the decay of the performance of the model trained on samples from the year 2014 as test sets become more distant in time from train one. Authors of [14] elaborate on the good practices for building the relevant malware dataset and conducting time-aware malware studies. Among the other recommendations they give, there are several that we follow in our research: describe an experimental environment, OS, network connectivity, etc.; describe the dataset; provide family names of the malware samples in use. Authors of [15] state, that train and test sets combination that is built based on a certain time metric will generally yield the performance worse than the one of a random split (e.g. k-folds cross-validation).

III. Methodology

This section is dedicated to the description of the methods used in this paper. Our choice of methods is based on findings made in [5], [2] and [3]. We begin with the description of our data collection process. Then we explain the way we preprocess and select features. Later, we explain the way we split our data into train and test sets. In the end, we describe the evaluation metrics and machine learning algorithm used in this paper.

A. Data collection

Our data collection is based on the BEP-AEP approach that was first presented and described in [3] and analysis of memory access patterns first used in [5]. The key concept involves splitting the behavioral trace of the process into two main parts: the one that occurs before the Entry Point (BEP) and after (AEP). In this paper, we focus only on the trace produced BEP. With a help of Intel Pin binary instrumentation framework [16] we record the memory access operations produced by the process from the moment it starts. We record only the type of memory access operation: *R* for read and *W* for write. We record the sequence of the first 1M of memory access operations. However, if the sample does not produce 1M million memory access operations BEP we still keep its data, thereby making our experiments more realistic. Similarly to [3] we stop recording the trace as soon as the execution flow reaches the first instruction from the main module of executable.

B. Data preprocessing and feature selection

The sequence of up to 1M of memory access operations is recorded for each sample in the dataset. Each sequence is later split into the set of overlapping n-grams of the size *n=96*: memory access patterns. Each next n-gram overlaps the previous one on *n-1* operations. These 96-grams later serve as features for ML algorithms. For classification purposes, each feature describes the presence or absence of a certain pattern in a trace of a sample: it takes value *1* if a pattern is present in a trace of a sample and *0* if not. When working with memory access traces the amount of features (unique

patterns) is always big and can reach millions of features [1] [2] [3] [5] [7]. It is unlikely, that all features contain valuable information. Moreover, regular machine learning packages are not suitable to work with data of such a high dimensionality. So it is important to perform feature selection before feeding the data into ML algorithm. To reduce the feature space, we perform a two-step feature selection process that was described in more detail in [3]. First, we select 50K best features from the training set based on their Information Gain (IG) [17]. Later, we use these 50K features to select the best feature subset using Correlation-based feature selection (CFS) [18] from ML package Weka [19]. CFS searches for the best subset of the given feature space and selects features that have a high correlation with classes in the dataset but low correlation between each other. We use CFS with the default for Weka parameters. With current implementations, it is challenging to use CFS on the full feature set, since performing the CFS requires a calculation of correlation matrix between all features, the process that requires an infeasible amount of time and computational resources when we are talking about millions of features. It is important to note, that CFS adds features to the feature set until the merit of the feature set stops growing more than a certain threshold [18]. Thus, it is challenging to choose the desired amount of features to be selected by CFS.

C. Splitting the dataset

Different authors utilize different approaches to test their malware detection method on previously unseen malware. Some simply split the dataset into train and test sets. While others make their dataset time coherent: samples arranged based on a certain time property. This allows emulating the updates of the models with time. In this paper, we arrange our dataset based on the first seen time from the VirusTotal (VT) [11]. There are not many other sources of time-related information when talking about the Windows executables, as compilation time available in PE header can be forged [20]. We split our dataset into bins based on the month the malware samples were first seen on VT. Note, that finding enough benign samples from a certain period of time is a quite challenging task. Thus, even though we also arrange benign samples based on VT data, we add them into bins based on their position on the benign timeline and the number of malicious samples in the same bin (see Section IV-B). This approach allows us to make training and test sets to have almost equal amounts of malicious and benign executables. We consider samples that are present in a certain bin to be *unseen* to those present in the older bins. Thus, newer benign and malicious samples do not contribute to the model and do not affect the feature selection process. We try to keep the amount of malicious and benign executables in bins equal. We also keep all malware samples in the bins regardless of the malware family they belong to. We decided to use our dataset as is since samples from the same family evolve over time and the distribution of families across the bins is not uniform what adds more realism to our experiments.

D. Evaluation

To check the applicability of memory access traces recorded BEP for the detection of previously unseen malware we train the ML model on the training set that consists of one or several bins and separately test it on the bins that were not used for training. We iteratively increase the training set by adding newer bins into it. As an ML algorithm we have chosen Random Forest (RF) algorithm from Weka [19] package, since it has shown one of the best results in [3]. RF constructs a number of decision trees, which are used for the classification. We use RF with default, for Weka, parameters where the number of trees is 100. To evaluate the quality of the models we use several metrics. Accuracy, as the amount of correctly classified samples. True positive rate (TPR), as the amount of actual malware that is detected as malware (detection rate). False positive rate (FPR), as the amount of actual benign executables classified as malware (potential false alarms in the system). In this paper, we show, how these metrics change with the increased amount of time passed since the ”last update” of the model (latest bin added to the training set).

IV. Experimental setup

In this section, we describe our experimental environment, provide details about our dataset, and explain our experimental flow.

A. Experimental environment

When using dynamic malware analysis, it is important to avoid the influence of changes in the experimental environment and ensure equal launching conditions for all samples. It is also important to isolate malware so that the host system or network are not affected by the malicious behavior. To ensure security and repeatability we use Virtual Box virtual machine (VM) with Windows 10 guest operating system. The VM was isolated from the internet. All our VMs were launched on the Virtual Dedicated Server (VDS) with 4-cores Intel Xeon CPU E5-2630 CPU running at 2.4GHz and 32GB of RAM with Ubuntu 18.04 as a host operating system.

B. Dataset

In this subsection, we describe the content of our dataset and explain how the dataset is split into bins which are later used to construct different train- and test-set combinations. This dataset was previously used in [3] and [1]. Our dataset can be divided into two main parts: benign and malicious executables. Malware samples were obtained from *VirusShare_00360* collection from VirusShare [21]. *VirusShare_00360* contained 65518 samples, out of which 2973 were PE executables. For each malicious sample, we got a report from VirusTotal(VT) [11]: an online malware analysis tool that also allows seeing how different Anti-Virus engines react to a certain sample. We left only samples that were recognized as malicious by at least 20 engines. In the final dataset, we included samples that belonged to the 10 most common families: Fareit, Occamy, Emotet, VBInject, Ursnif, Prepscram, CeeInject, Tiggre, Skeeyah, GandCrab. According to the VT reports, resulted

TABLE I
DISTRIBUTION OF MALWARE FAMILIES IN THE DATASET

Total	Fareit	Occamy	Emotet	VBInject	Ursnif	Prepscram	CeeInject	Tiggre	Skeeyah	GandCrab
2005	573	307	196	164	162	143	127	117	115	101

samples were first seen (first submission date) between March 2018 and March 2019. Not all the samples were launched successfully, so the amount of malware samples that generated traces is 2005. Benign samples were downloaded from Portable Apps [22] in September 2019 and is a set of free Portable applications. It contains various software such as graphical, text, and database editors; games; browsers; office, music, audio, and other types of Windows software. According to the VT, benign samples were first seen between December 2006 and December 2019. Some benign samples were first seen on VT after their download date because it was we who first uploaded them to the VT to check whether they are truly benign. We left only samples that were not recognized as malicious by any of the AV engines available on VT. After running the samples 2098 of them produced traces.

As it is outlined in the literature [14], it is important to present the distribution of malware categories in the dataset. In Table I we show the number of samples that belong to each of the families. As it is possible to see, the dataset is not balanced in terms of malware families. However, we did not polish this aspect of our dataset since we only cared about samples being benign and malicious.

We looked at two possible splitting approaches to create time-ordered subsets from the original dataset. In the first approach, we split malware into 13 bins based on the month they were first seen on VT. We also included benign samples into the monthly bins based on their VT first seen date. However, this approach resulted in highly imbalanced subsets, where the malware to goodware ratio sometimes was as high as 50 to 1. It is quite problematic to find the desired amount of benign samples from the desired time period. So we decided to discard the first approach due to this imbalance. Instead, we decided to split benign samples into 13 time-ordered bins and align the number of samples in them according to monthly bins created from malware samples. Each bin would contain as many benign samples as the corresponding malicious bin. Only the last bin would contain more benign samples since the amount of benign samples is bigger in the original dataset. This way, every next bin will have benign samples that are newer than those in the previous one. In Table II we present the amount of benign and malicious samples that were put into each of the 13 resulting bins. As we can see, the bins have different amounts of samples in them. To describe our bins in an even more detailed way, in Fig. 1 we present the distribution of the above-mentioned malware families among the malware samples in each of the 13 bins. As we can see, malware families are not evenly distributed among the bins.

Having 13 bins with equal malware to goodware ratio except for the bin #13 we use them to construct train and test sets that are used to training and test ML models. The training set

TABLE II
AMOUNT OF BENIGN AND MALICIOUS SAMPLES IN BINS.

Bin #	0	1	2	3	4	5	6	7	8	9	10	11	12
Benign	64	119	174	88	167	66	39	130	149	264	214	307	317
Malicious	64	119	174	88	167	66	39	130	149	264	214	307	224

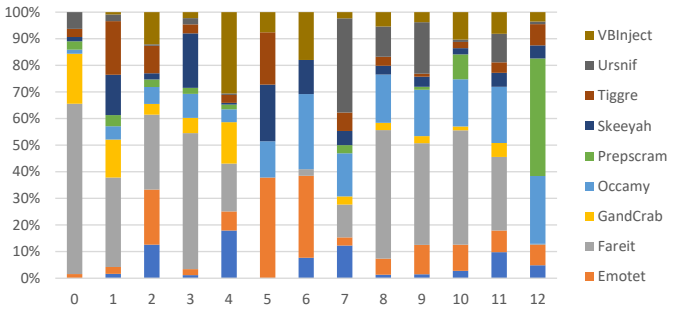


Fig. 1. Distribution of malware families among the malware samples within each of the 13 bins

is a combination of one or several consecutive bins. From here the training sets will be named in the following way: training set based on the bin #0 is called *T0* while training set built from bins #0 to 7 is called *T0_7* and so on. For each training set, we have one or more test sets, made of the remaining bins that were not used in the construction of the train set. For example, for the training set *T0_10* we will have two test sets consist of the bin *11* and bin *12* respectively. With such approach, we obtain 12 combinations of train and test sets. As we previously described the distribution of malware families within the bins (which now also represent test sets) we now use Fig. 2 to show the distribution of the malware families within train sets. As we can see, after the train set *T0_1* the distribution of families within the train sets begins to stabilize itself and becomes quite similar between the train sets closer to the last one.

C. Experimental flow

Every sample from our dataset is first copied to the clean snapshot of the VM. Then, we launch it together with a customized Intel Pin tool. The Intel Pin tool records memory access operations and stores them into a trace file. The trace

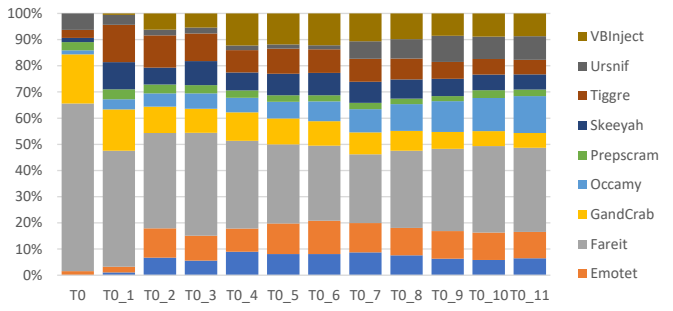
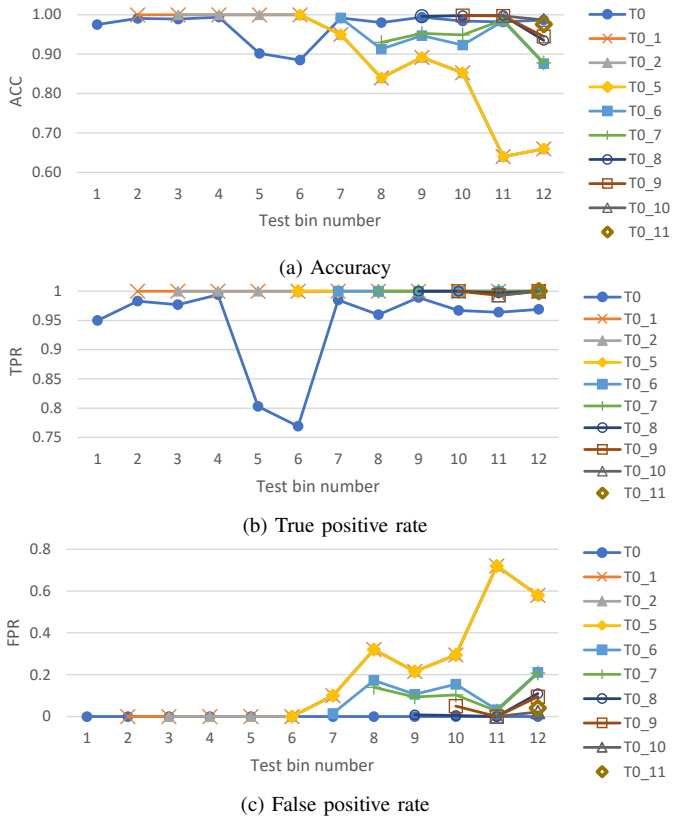


Fig. 2. Distribution of malware families among the malware samples within train sets

is later copied to the host system, and the VM is reverted to the previous state. It is important to note, that the benign executables from PortableApps were copied to the VM together with the content of their folder. This approach allowed us to provide more realistic results since benign executables often require additional resources to be launched properly.

V. Results

In this section, we provide the detection performance achieved with our approach by RF algorithm. As we outlined in Section III we decided to test our malware detection approach with help of the RF ML algorithm because it has previously shown good classification performance with a similar type of features. In Fig. 3 we show the accuracy, true positive rate, and false positive rate of RF algorithm. For table data see Appendix A. Each line on the chart represents a certain evaluation metric of the ML model trained on a certain train set. Each point of the line is the value of the metric obtained while attempting to classify samples from one of the test bins. Before looking into the achieved results it is important to mention, that accuracy, TPR, and FPR achieved by models trained on the sets *T0_1* - *T0_5* match for all the corresponding test sets. Thus, corresponding points and lines on the charts merge. To simplify our charts we omit results achieved with *T0_3* - *T0_4* since they are the same as those achieved with *T0_1, T0_2* and *T0_5*. First, let's take a look at the accuracy achieved by the RF algorithm. As it is possible to see from Fig. 3a we can outline two main trends in the classification performance. The first trend shows, that the further in time a test bin from the train set - the lower the classification accuracy. This trend, however, has several exceptions. First of all, the model trained on *T0* shows a drop in accuracy for test bins 5 and 6. For other test bins, it has a quite stable accuracy while showing minor improvements (e.g. accuracy on bin 7 is higher than the one on the bin 0). Lastly, models trained on *T0_6* and *T0_7* show a significant spike of accuracy on the *T0_11* and drop on the last test bin. The second trend shows, that the closer train set to the test set (the more up-to-date it is) the higher classification accuracy on the test set becomes. For example, accuracy on the test set 12 improves when the train set is updated with newer bins. Now let's look at the TPR and FPR showed by the RF algorithm. As we can see, most of the test bins are classified with TPR that is equal to or higher than 0.95. Moreover, models trained on sets *T0_1* - *T0_7* always show TPR of 1 for all test bins. It means, that such a model will not miss any of the previously unseen malicious samples. However, from the FPR chart, we can also see that some models (especially *T0_1* - *T0_5*) show an increasing amount of false positives for test bins that are further away from the training set. FPR can become as high as 0.72 which in reality will result in a significant amount of false alarms and may seriously affect the operations of the system that uses such models in AV solutions. It is also important to mention, that model trained on *T0* shows 0 FPR for all of the test bins while missing some of the malware samples. The overall trend of FPR and TPR is the following. Most of the models, while keeping high TPR (detection rate) over time develop higher FPR which clearly shows that even the models with good detection capabilities have to be regularly updated.



(a) Accuracy

(b) True positive rate

(c) False positive rate

Fig. 3. Performance of RF algorithm

While acquiring the data present on the Fig. 3a, 3b and 3c we derived several interesting findings. The performance of the models does not change when trained on sets *T0_1* - *T0_5* as they show very low accuracy towards the last test bins. But when the model is trained on the sets *T0_6* and beyond, the accuracy rapidly improves. We also noticed, that when a train set is changed from *T0* to *T0_1* the model starts performing worse for many of the test bins. It is quite counter-intuitive since normally we would expect a better performance of the model that used more recent samples (bigger and updated training set) to train. The rapid improvement of accuracy and a counter-intuitive difference in performance between models trained on *T0* and *T0_1* raised our attention and we analyze these findings in Section VI.

As we can see, the performance of most of the models built with the use of memory access patterns recorded BEP degrade over time. Some models degrade more than the others, acquiring high FPR. But at the same time, the TPR of most of the models remains relatively high, thus even an outdated model built with BEP memory access traces will protect the potential system over a long period of time (while producing a high amount of false alarms).

TABLE III
AMOUNT OF FEATURES SELECTED BY CFS FEATURE SELECTION METHOD FOR ALL TRAIN SETS

Train Set	T0	T0_1	T0_2	T0_3	T0_4	T0_5	T0_6	T0_7	T0_8	T0_9	T0_10	T0_11
Features	555	133	135	136	134	134	134	8	10	11	11	10

VI. ANALYSIS

When we discovered a rapid improvement in the model's performance with the change of train set from *T0_5* to *T0_6* and significant difference between models trained on *T0* and *T0_1* we had several hypotheses about the reason for these changes.

A. Influence of families

The simplest idea was the influence of the malware families' distribution in training and test sets. For example, different distribution of families in train sets could lead to models biased towards a certain category of malware. However, if looking into Fig. 2 and 1 from Section IV we may see, that the distribution of families in training sets *T0_5* and *T0_6* are almost identical. The family distribution does not also explain the difference in performance between models trained on *T0* and *T0_1*: it is easy to see, that family distribution in *T0_1* is closer to e.g. test bin 9 than the one in *T0*. Thus we rejected this hypothesis.

B. Influence of features

The other potential reason for the model performance changes could be the features. As we are not using the entire feature set and using a two-step feature selection process it could be, that the features we select as well as their amount can affect the potential performance of the model trained on data built with such features. First of all, let's take a look in Table III where the amounts of features selected by CFS for each of the train sets are present.

First of all, it is easy to see that train sets can be grouped into three categories based on the number of features selected on them by CFS. In the first group, it will be a single set *T0*: 555 features were selected from it. In the second group there will be sets *T0_1 - T0_6* with the number of features ranging from 133 to 136. And in the third group there will be sets *T0_7 - T0_11* with the number of features ranging from 8 to 11.

The amounts of features selected from the sets in the third group did not surprise us, since they are quite similar to what can be seen in [3]. It was already known, that in some cases we need very few of the BEP memory access patterns to achieve 0.99 classification accuracy.

When talking about *T0*, 555 is a quite high amount of features, since *T0* has only 128 samples. It is generally considered in the literature, that fewer features in the dataset improves the performance of ML algorithms [5] [7] [3] [17]. In some articles authors suggest using *the rule of 10*: to train a good performing ML model, it is advised to have ten times more training samples than features [23]. However, in our case, a bigger amount of features allowed to eliminate false positives and contributed towards relatively stable TPR and accuracy along the test sets. Based on the experience from [3] we were surprised by the fact that CFS has selected so many features for a relatively small dataset. So we calculated IG for all of the 555 features selected from *T0* and found, that 531 of them had IG of *1*. For a two-class dataset, it means that each of these features can be solely used to correctly classify all samples of the train set. As CFS stops adding features to the feature set when the merit of the set stops to increase, it becomes clear that the high amount of selected features is due to their high quality. Even if they correlate with each other, there is no way to distinguish between features with exact same values for all samples if they carry a lot of information.

TABLE IV
AMOUNT OF COMMON FEATURES BETWEEN THE FEATURE SETS.

	T0_555	T0_1_133	T0_2_135	T0_3_136	T0_4_134	T0_5_134	T0_6_134	T0_7_8	T0_8_10	T0_9_11	T0_10_11	T0_11_10
T0_555	555	126	126	127	126	126	126	2	2	3	3	3
T0_1_133		133	128	126	126	126	126	1	1	1	1	1
T0_2_135			135	127	127	127	127	1	1	1	1	1
T0_3_136				136	127	127	127	1	1	1	1	1
T0_4_134					134	128	128	1	1	1	1	1
T0_5_134						134	128	1	1	1	1	1
T0_6_134							134	1	1	2	1	1
T0_7_8								8	0	0	0	0
T0_8_10									10	4	2	0
T0_9_11										11	1	1
T0_10_11											11	5
T0_11_10												10

While evaluating the second group, we found, that the number of features selected for the sets *T0_5* and *T0_6* is the same. So the number of features has no influence over the rapid increase in model performance. Thus we decided to check how feature sets change with the change of the train sets. To observe changes in feature sets we built a Table IV. In this table, each row and column is named *T0_X_N* where *T0_X* represents one of the training sets, while *N* is the number of features selected from this training set. Each cell of the table shows the amount of features common between the feature sets whose row and column cross in this cell.

It is easy to see, that feature sets from *T0* to *T0_6* share many common features between them. However, as it was shown in Section V, model trained on *T0* show better performance on the last 6 test bins than models trained on *T0_1-T0_6*. This can be a sign of the drawback in our feature selection approach, as it can not select the same features from e.g. train sets *T0* and *T0_1*. Let's now look into the feature sets *T0_5_134* and *T0_6_134*, a place where the RF model gets rapid improvement in classification accuracy. These feature sets share 128 of 134 features. However, the latter allows for higher classification accuracy. We examined 6 "old" unique features from *T0_5_134* and 6 "new" from *T0_6_134* in terms of the information they carry. We found, that 5 out of 6 of old and new features have the same IG in their respective train sets. The remaining features have their IG different in the 5th digit after the decimal point. We believe, that such an

TABLE V
PROPORTION OF FEATURES THAT REPRESENT ONE CLASS MORE THAN ANOTHER.

Feature Set	T0_555	T0_1_133	T0_2_135	T0_3_136	T0_4_134	T0_5_134	T0_6_134	T0_7_8	T0_8_10	T0_9_11	T0_10_11	T0_11_10
Mal	0.771	0.030	0.037	0.037	0.022	0.022	0.022	0.125	0.2	0.273	0.364	0.5
Ben	0.229	0.970	0.963	0.963	0.978	0.978	0.978	0.875	0.8	0.727	0.636	0.5

insignificant difference in feature set quality could not result in the improvement of classification accuracy that we've seen in Section V. It is also worth to mention, that feature sets from *T0_7* to *T0_11* share more common features with *T0* then with *T0_1-T0_6*. And the RF models trained on them generally perform better than those trained on *T0_1-T0_6*.

Another thing we could check emerges from the nature of our features and the way we process experimental data. As we explained in Section III the feature takes value *1* when a certain memory access pattern is generated by a sample and *0* otherwise. What can happen, that the majority of selected features take value *1* for more samples of one class that of another. This means, that the feature set *represents* behavior of a certain class. In other words, one class can be described by the presence of certain memory access patterns while another by *absence* of such. And since malware and benign software evolve over the time it might happen, that newer samples of a certain class will start generating patterns that were not generated by samples of this class at the time of training (in the training set) and vice versa. Thus, we decided to explore how selected features represent classes in train sets. To do this we counted the proportion of features that represent more malicious or more benign samples. Basically, we found the number of features that take value *1* in more samples of one class than in another. In Table V, values in column Mal reflect the portion of the entire feature set features that take value *1* more often in malicious samples, while column Ben reflect similar values for the benign class. As we can see from this table, feature sets that contribute to the good performance of RF models (*T0,T0_7-T0_11*) have a smaller imbalance between the number of features that represent malicious and benign classes than other feature sets. However, the feature set *T0_6* stands out. It has one of the highest imbalances but allows for better performance than feature sets with similar feature balance. Thus, we can not conclude that the way features represent classes affects the performance of models. We also trained RF models with 50K best features (see Section III) but the results were the same as with CFS-selected features.

C. Influence of feature space

As we found, that feature qualities have no direct influence on the classification performance we decided to check whether the entire feature sets can be a reason for the classification performance we observed in Section V. Samples in a dataset can be considered as points in the multidimensional space, where dimensionality is defined by the number of features and coordinates of the point are the values of features for the particular sample. It is known, that in general the further away in given feature space samples of a different class from each other - the easier it is to distinguish between them [17] [24]. Some ML algorithms, e.g. k-Nearest Neighbors or Support Vector Machines, use distances between samples directly. But even if the distance measure between samples is not used in the ML algorithm, two samples of two different classes that have the same coordinates (are in the same point of feature space) are impossible to distinguish from each other. So we decided to visualize how selected features allow to separate samples of different classes. It is impossible to draw a space which dimensionality exceeds 3. However, there is a way to reduce the dimensionality of a dataset and draw it on the 2D plane while keeping relative distances between point intact: multidimensional scaling [25] (MDS). With a help of MDS, it is possible to visualize how samples from the multidimensional dataset are located relative to each other on the two-dimensional plane. Using MDS implementation from *scikit-learn* [26] Python package we built Fig. 4. In this Figure, each subfigure is an illustration of the location of the train and test samples in the feature space of a certain training set. For example, in Fig. 4c we can see how samples are located in the feature space of the *T0_2*. On each of the subfigures we depicted the following elements (for colors and shapes see the legend on Subfigure 4m):

- Train samples of benign and malicious classes. Train samples of a malicious class have different shapes according to their family (see Legend) but use the same dark-red color.
- Test samples of benign and malicious classes. Test samples of a malicious class have different shapes and colors according to their family (see Legend).
- We have also marked centers of malicious and benign parts of the train and each of the test sets. A *center* here is a point that has coordinates equal to the mean of the samples in the group: it can be considered as a centroid of the corresponding cluster. For example in Fig. 4l a point Named "Ben Test 12" is a center of benign samples for test set 12.

From Fig. 4 it is possible to understand some of the classification results. For example, in Fig. 4a we can see, that benign parts of train and test samples lay relatively close to each other, while several groups of malicious test samples are located closer to benign samples than to the malicious train set. This explains 0 FPR achieved by ML model on this train and test sets combination and non-ideal TPR since some malicious samples are closer to the benign part of the train set than to the malicious one. On its turn, Fig. 4b shows why FPR grows and accuracy drops for a model trained on *T0_1*: both benign samples and centers of benign parts of test sets become closer to the malicious train part over the time. We may also observe in Fig. 4h- 4l that malicious and benign sets slightly overlap, but still quite distinguishable. On the other hand, a comparison of Fig. 4f and 4g does not explain the rapid improvement

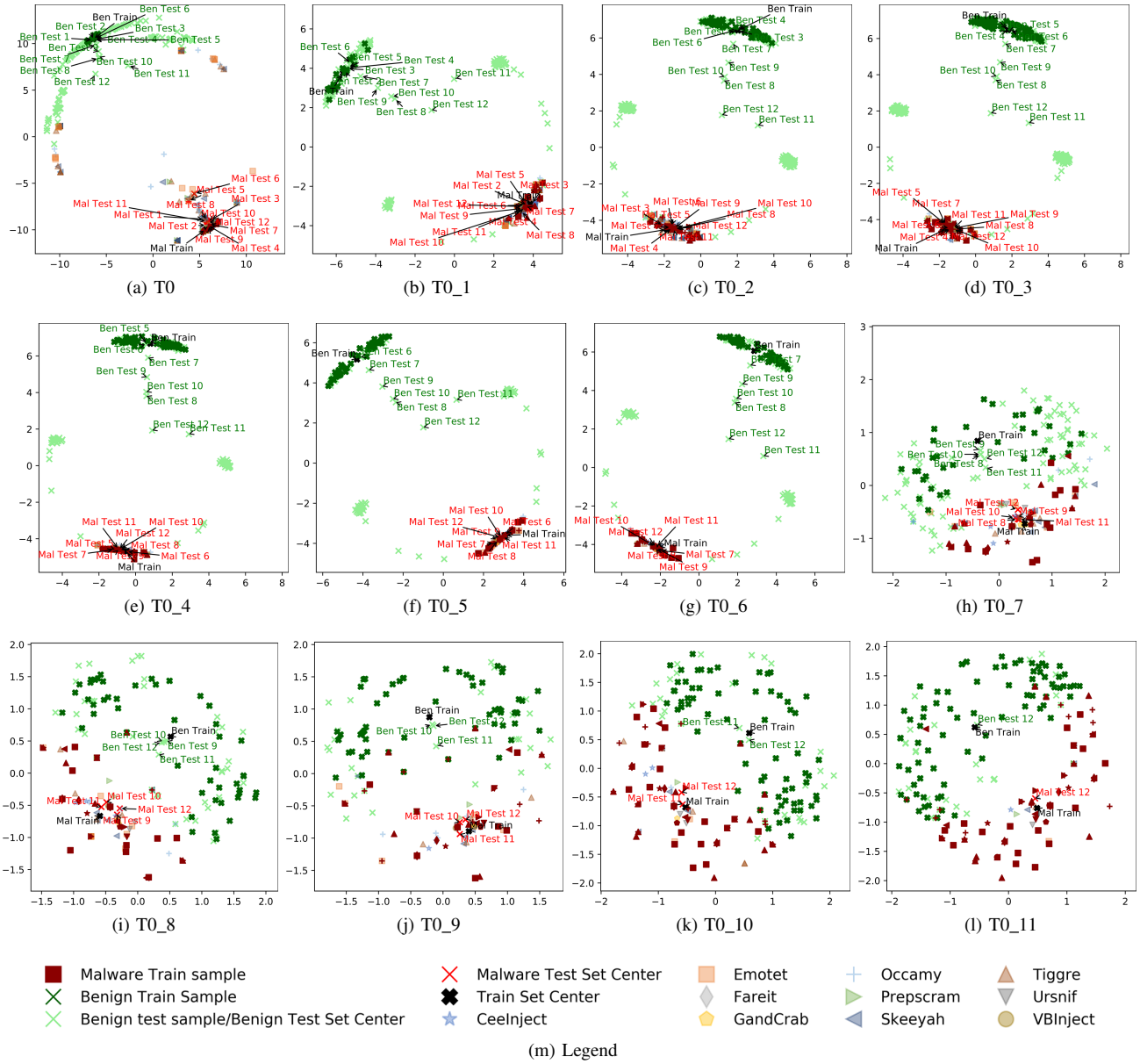


Fig. 4. Distance-preserving projection of train and test samples from multidimensional feature spaces into the two dimensional plane.

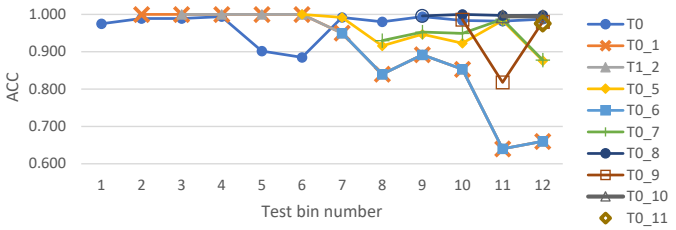
in the classification accuracy of the models. Moreover, the relative positioning of the benign and malicious sets almost doesn't differ[1]. At this point, we had to conclude, that feature spaces analysis does not help to understand the classification accuracy difference between *T0_5* and *T0_6* models. We must also admit, that the counter-intuitive difference between the performance of models trained on *T0* and *T0_1* can not be explained with this approach. Our next hypothesis about the unexpected classification performance was about the potential limitations of the RF algorithm. Thus, we decided to train models with several different ML algorithms and compare their performance to the RF algorithm. We decided to find out whether it is possible to obtain models that can detect malware with the use of BEP memory access traces better.

VII. Additional Evaluation

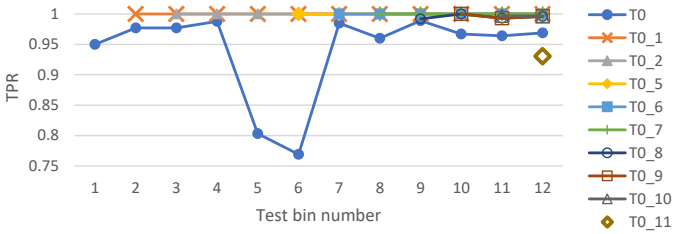
This section is dedicated to answering the question "Is it possible to classify malicious and benign samples better than with use of RF algorithm?[2]". When we decided to test classification performance of other algorithms we first checked the k-Nearest Neighbors(kNN) algorithm as it performed quite well in [3], [7] or [5]. However, it showed very similar to RF performance so we do not present the results achieved

[1]It is important to note, that Fig. 4f and 4g look rotated against each other only because we had no control over how exactly the points from multidimensional space are placed on the two-dimensional plane by the MDS algorithm.

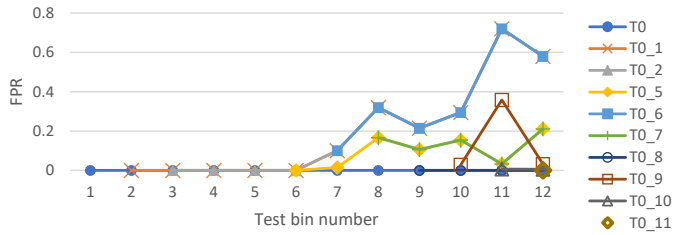
[2]Under the current conditions: same dataset and features.



(a) Accuracy



(b) True positive rate



(c) False positive rate

Fig. 5. Performance of J48 algorithm

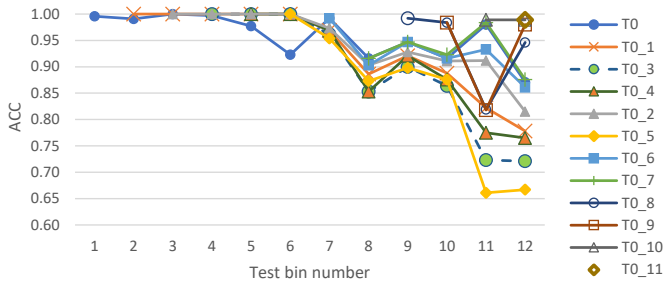
by kNN. The next algorithm we decided to check was the J48 (Decision Trees) algorithm from Weka [19]. The main difference between RF and J48 is the *number* of trees that are used. By default, RF from Weka uses 100 trees while J48 builds a single tree. In Fig. 5 we present accuracy, TPR, and FPR of J48. For table data see Appendix B. First of all, J48 trained on sets *T0-T0_4* show exactly the same performance as RF. On the Fig. 5, similarly to Fig. 3, we omit results on *T0_3* and *T0_4* because the results on the remaining test sets do not change. However, when we look at the performance of J48 trained on *T0_5* we can easily see, that J48 performs better with this train set than RF. Moreover, J48 trained on *T0_5* classifies test sets 7-11 with exactly the same accuracy as RF trained on *T0_6*. At the same time, J48 trained on *T0_6* classifies test sets 7-12 with exactly the same accuracy as RF trained on *T0_5*. For the rest of the training sets, J48 and RF behave mostly similarly but there are some visible differences. E.g. J48 trained on *T0_8* performs better than RF, while RF trained on *T0_9* performs better than J48. As RF is a set of many decision trees, we explored how the number of trees affects the performance of RF. It was found, that the number of trees of 4 or less RF performs the same way J48 does. This means that we could have observed a case of overfitting of the Random Forest algorithm when trained on *T0_5*. So far we showed, that there can be specific cases when one tree-based algorithm outperforms another. But in an attempt to improve the classification accuracy of models trained on *T0_1-T0_5* we decided to utilize Locally-Weighted Learning (LWL) algorithm from Weka. LWL is a combination of kNN and any other ML algorithm that supports weighted learning and has relatively low training time. The basic principle of LWL is the following: to classify a test sample, at the time of classification LWL trains an ML model with a use of k train samples that are close to the test sample. The k samples are weighted according to their distance. This way every test sample is classified by a separate ML model. The default ML algorithm that is used in LWL in Weka is Decision Stump (DS). Decision stump is a simple decision tree that consists of only one node. In the Fig. 6 we present the performance of the LWL algorithm. For table data see Appendix C What is easy to see in Fig. 6b is the improvement of TPR if compared to RF and J48. We can also observe the FPR values (Fig. 6c) became more diverse between the models trained on different train sets. From the accuracy chart (Fig. 6a) we can see, that the model trained on *T0* performs not as well as similar models of RF and J48. Its performance almost matches the one of a model trained on *T0_7* on the test sets 8-12. We also observe a decline in the performance of the models *T0_1* - *T0_5* if compared to *T0*. However, the LWL models *T0_1* - *T0_4* perform better than those of RF and J48: accuracy is higher, while FPR is lower. So we were able to improve the performance of some of the low-performing models by changing the ML algorithm. But the LWL trained on *T0_5* performs almost as bad as the RF and significantly worse than J48. When switching to the *T0_6* model we see the rapid improvement of the performance that is similar to the one we have seen with RF.

As we were able to see, for some combinations of train and test sets it is possible to improve classification performance by choosing the different ML algorithms. Thus, we can answer positively to the question outlined at the beginning of this section. Some algorithms will perform better under certain conditions while worse under the other. But the final choice of the ML algorithm is always up to the developers of the potential AV system and should be based on the requirements of the system in interest.

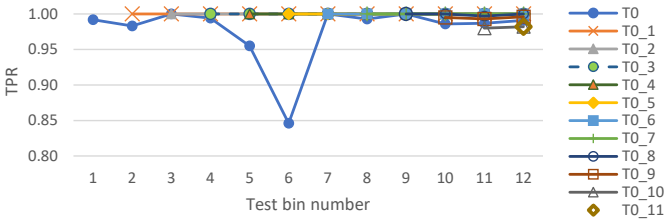
VIII. Discussion and Conclusions

In this section, we discuss our findings, present the conclusions, and outline possible improvements that can be implemented in future work.

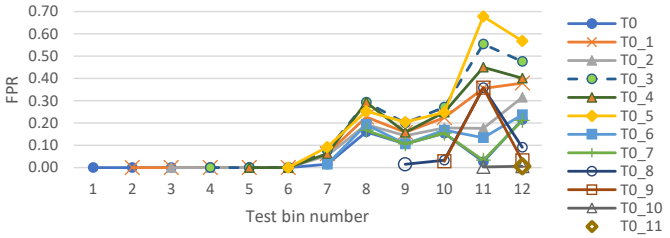
In this paper, we have shown, that behavioral traces recorded before the Entry point have the potential to be used for the detection of previously unseen malware. It is an important finding since malware detected BEP has no chance to harm the system even though it was launched. The results presented in Sections V and VII show, that we can answer *yes* to the *RQ1* outlined in Section I. The memory access traces recorded before the Entry Point can be successfully used to distinguish between previously unseen malicious and benign executables. To answer the *RQ2* we have also shown, that most of the ML models trained on the BEP memory access traces can provide a good detection rate ($TPR > 0.95$) for the significant periods of time since the update of the model. Some models provide high TPR for a period of at least 11 months. But it is important



(a) Accuracy



(b) True positive rate



(c) False positive rate

Fig. 6. Performance of LWL algorithm

to remember, that they also tend to develop high FPR which is a clear sign of the need for regular updates of the model since high FPR will disrupt operations of the system by raising a lot of false alarms. Thus, we have to conclude that memory access traces recorded BEP can be used for malware detection, but such an approach has its limitations that should be taken into account.

We have also performed an attempt to explain some of the cases of difference in the performance of ML models. Under our approach, we were not able to show that features or feature spaces influence the classification performance of the models. But we were able to show, that in some cases (for certain combinations of train and test sets) some ML algorithms perform better than the others. But it is still important to pay attention to the TPR and FPR when making a choice of the ML algorithm to be used in a real system. Some trends of the classification performance results remain similar between different ML algorithms: e.g. models trained on *T0* outperform those trained on *T0_1 - T0_6* on the last 7 test bins (true for RF, J48, LWL); models trained on *T0_5* are among the worst-performing models (RF, LWL); model trained on *T0* shows a drop of performance for the test bins 5 and 6 (RF, J48, LWL). Based on these findings we have to conclude, that such performance of models can be a sign of a potential weakness of our BEP memory access patterns approach. In future work, one may try to understand, whether this is a weakness in the use of memory access patterns or the fact that we are focusing on the BEP activity. To do this, different types of features and features recorded AEP may be used on the same dataset. It may also be the result of some special properties of our dataset, that we had no control over since we used all of the available samples. So in future research, one may use our approach on the different, potentially larger, and more diverse dataset. It may also be useful to perform the analysis of the misclassified samples [14], as it may help to understand the classification performance as it was shown in [7]. During the analysis phase we also observed a case of overfitting of Random Forest algorithm. We believe, that this finding has a potential to be investigated by machine learning researchers working on improving of understanding the performance of common ML algorithms.

We also believe, that our paper provides an important example of the analysis of the classification performance. Analysis of subcategories, the influence of the amount and quality of features in the updated feature set, and graphical analysis of feature space can help other researchers to understand their results. We think, that such approach can be used not only in malware analysis but in many areas where ML is used. Together with the feature selection approach, where we can reduce the feature space from hundreds of thousands and millions of features to hundreds and even fewer features, our paper provides valuable solutions for those working with Big Data and high-dimensional datasets.

REFERENCES

[1] S. Banin, *Malware Analysis using Artificial Intelligence and Deep Learning: Fast and straightforward feature selectionmethod: A case of high dimensional low sample size dataset inmalware analysis*. Springer, 2020.

[2] S. Banin and G. O. Dyrkolbotn, "Correlating high-and low-level features," in *International Workshop on Security*. Springer, 2019, pp. 149–167.

[3] S. Banin and G. O. Dyrkolbotn, "Detection of running malware before it becomes malicious," in *International Workshop on Security*. Springer, 2020, pp. 57–73.

[4] AVTEST. The independent IT-Security Institute, "Malware," https://www.av-test.org/en/statistics/malware/, 2020.

[5] S. Banin, A. Shalaginov, and K. Franke, "Memory access patterns for malware detection," *Norsk informasjonssikkerhetskonferanse (NISK)*, pp. 96–107, 2016.

[6] Ç. Yücel and A. Koltuksuz, "Imaging and evaluating the memory access for malware," *Forensic Science International: Digital Investigation*, vol. 32, p. 200903, 2020.

[7] S. Banin and G. O. Dyrkolbotn, "Multinomial malware classification via low-level features," *Digital Investigation*, vol. 26, pp. S107–S117, 2018.

[8] S. Sharma, C. R. Krishna, and S. K. Sahay, "Detection of advanced malware by machine learning techniques," in *Soft Computing: Theories and Applications*. Springer, 2019, pp. 333–342.

[9] A. Sharma, S. K. Sahay, and A. Kumar, "Improving the detection accuracy of unknown malware by partitioning the executables in groups," in *Advanced computing and communication technologies*. Springer, 2016, pp. 421–431.

[10] B. A. Miller, "Scalable platform for malicious content detection integrating machine learning and manual review," Ph.D. dissertation, UC Berkeley, 2015.

[11] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www. virustotal. com/en*, 2012.

[12] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 1–28, 2020.

[13] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 729–746.

[14] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen, "Prudent practices for designing malware experiments: Status quo and outlook," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 65–79.

[15] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Are your training datasets yet relevant?" in *International Symposium on Engineering Secure Software and Systems*. Springer, 2015, pp. 51–67.

[16] IntelPin, "A dynamic binary instrumentation tool," 2020.

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

[18] M. A. Hall, "Correlation-based feature subset selection for machine learning," Ph.D. dissertation, University of Waikato, Hamilton, New Zealand, 1998.

[19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[20] A. Shalaginov, S. Banin, A. Dehghantanha, and K. Franke, "Machine learning aided static malware analysis: A survey and tutorial," in *Cyber Threat Intelligence*. Springer, 2018, pp. 7–45.

[21] VirusShare, "Virusshare.com," http://virusshare.com/, accessed: 12.10.2020.

[22] PortableApps.com, "Portableapps.com," https://portableapps.com/apps, 2020.

[23] Malay Haldar, "How much training data do you need?" https://medium.com/@malay.haldar/how-much-training-data-do-you-need-da8ec091e956, 2015.

[24] S. Sathe and C. C. Aggarwal, "Nearest neighbor classifiers versus random forests and support vector machines," in *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2019, pp. 1300–1305.

[25] A. Mead, "Review of the development of multidimensional scaling methods," *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 41, no. 1, pp. 27–39, 1992.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

APPENDIX A
CLASSIFICATION RESULTS ACHIEVED BY RF

TABLE VI
RF ACCURACY

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.975	0.991	0.989	0.994	0.902	0.885	0.992	0.980	0.994	0.984	0.982	0.987
T0_1		1.000	1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_2			1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_5						1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_6							0.992	0.913	0.947	0.923	0.984	0.876
T0_7								0.930	0.953	0.949	0.987	0.878
T0_8									0.996	0.998	0.998	0.935
T0_9										0.998	0.997	0.945
T0_10											0.997	0.987
T0_11												0.976

TABLE VII
RF TPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.95	0.983	0.977	0.994	0.803	0.769	0.985	0.96	0.989	0.967	0.964	0.969
T0_1		1	1	1	1	1	1	1	1	1	1	1
T0_2			1	1	1	1	1	1	1	1	1	1
T0_5						1	1	1	1	1	1	1
T0_6							1	1	1	1	1	1
T0_7								1	1	1	1	1
T0_8									1	1	0.997	1
T0_9										1	0.993	1
T0_10											0.993	1
T0_11												1

TABLE VIII
RF FPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0	0	0	0	0	0	0	0	0	0	0	0
T0_1		0	0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_2			0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_5						0	0.1	0.32	0.215	0.294	0.72	0.58
T0_6							0.015	0.173	0.106	0.154	0.033	0.211
T0_7								0.14	0.094	0.103	0.026	0.208
T0_8									0.008	0.005	0	0.11
T0_9										0.05	0	0.095
T0_10											0	0.022
T0_11												0.041

APPENDIX B
CLASSIFICATION RESULTS ACHIEVED BY J48

TABLE IX
J48 ACCURACY

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.975	0.989	0.989	0.994	0.902	0.885	0.992	0.980	0.994	0.984	0.982	0.987
T0_1		1.000	1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_2			1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_5						1.000	0.992	0.916	0.947	0.923	0.984	0.876
T0_6							0.950	0.840	0.892	0.853	0.640	0.660
T0_7								0.916	0.947	0.923	0.984	0.876
T0_8									0.996	1.000	0.997	0.998
T0_9										0.986	0.818	0.980
T0_10											0.995	0.995
T0_11												0.959

TABLE X
J48 TPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.95	0.977	0.977	0.988	0.803	0.769	0.985	0.96	0.989	0.967	0.964	0.969
T0_1		1	1	1	1	1	1	1	1	1	1	1
T0_2			1	1	1	1	1	1	1	1	1	1
T0_5						1	1	1	1	1	1	1
T0_6							1	1	1	1	1	1
T0_7								1	1	1	1	1
T0_8									0.992	1	0.993	0.996
T0_9										1	0.993	0.996
T0_10											0.997	0.997
T0_11												0.931

TABLE XI
J48 FPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0	0	0	0	0	0	0	0	0	0	0	0
T0_1		0	0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_2			0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_5						0	0.015	0.167	0.106	0.154	0.033	0.211
T0_6							0.1	0.32	0.215	0.294	0.72	0.58
T0_7								0.167	0.106	0.154	0.033	0.211
T0_8									0	0	0	0
T0_9										0.028	0.358	0.032
T0_10											0.007	0.006
T0_11												0

APPENDIX C
CLASSIFICATION RESULTS ACHIEVED BY LWL

TABLE XII
LWL ACCURACY

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.996	0.991	1.000	0.997	0.977	0.923	0.992	0.916	0.947	0.916	0.980	0.871
T0_1		1.000	1.000	1.000	1.000	1.000	0.969	0.886	0.921	0.888	0.822	0.778
T0_2			1.000	1.000	1.000	1.000	0.973	0.903	0.928	0.911	0.912	0.815
T0_3				1.000	1.000	1.000	0.962	0.853	0.900	0.864	0.723	0.721
T0_4					1.000	1.000	0.969	0.853	0.921	0.876	0.775	0.765
T0_5						1.000	0.954	0.873	0.898	0.876	0.661	0.667
T0_6							0.992	0.903	0.947	0.916	0.933	0.861
T0_7								0.916	0.947	0.923	0.984	0.876
T0_8									0.992	0.984	0.819	0.946
T0_9										0.984	0.818	0.980
T0_10											0.989	0.989
T0_11												0.989

TABLE XIII
LWL TPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.992	0.98	1	0.994	0.96	0.85	1	0.99	1	0.986	0.987	0.991
T0_1		1	1	1	1	1	1	1	1	1	1	1
T0_2			1	1	1	1	1	1	1	1	1	1
T0_3				1	1	1	1	1	1	1	1	1
T0_4					1	1	1	1	1	1	1	1
T0_5						1	1	1	1	1	1	1
T0_6							1	1	1	1	1	1
T0_7								1	1	1	1	1
T0_8									1	1	0.997	1
T0_9										0.995	0.993	0.996
T0_10											0.98	0.982
T0_11												0.982

TABLE XIV
LWL FPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0	0	0	0	0	0	0.015	0.16	0.106	0.154	0.026	0.215
T0_1		0	0	0	0	0	0.062	0.23	0.158	0.224	0.355	0.379
T0_2			0	0	0	0	0.054	0.19	0.143	0.178	0.176	0.315
T0_3				0	0	0	0.077	0.29	0.2	0.271	0.554	0.476
T0_4					0	0	0.062	0.29	0.158	0.248	0.45	0.401
T0_5						0	0.092	0.25	0.204	0.248	0.678	0.568
T0_6							0.015	0.19	0.106	0.168	0.134	0.237
T0_7								0.17	0.106	0.154	0.033	0.211
T0_8									0.015	0.033	0.358	0.091
T0_9										0.028	0.358	0.032
T0_10											0.003	0.006
T0_11												0.006