Correlating High- and Low-Level Features:*
Increased Understanding of Malware Classification

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

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

Abstract. Malware brings constant threats to the services and facilities used by modern society. In order to perform and improve anti-malware defense, there is a need for methods that are capable of malware categorization. As malware grouped into categories according to its functionality, dynamic malware analysis is a reliable source of features that are useful for malware classification. Different types of dynamic features are described in literature[6][5][13]. These features can be divided into two main groups: high-level features (API calls, File activity, Network activity, etc.) and low-level features (memory access patterns, high-performance counters, etc). Low-level features bring special interest for malware analysts: regardless of the anti-detection mechanisms used by malware, it is impossible to avoid execution on hardware. As hardware-based security solutions are constantly developed by hardware manufacturers and prototyped by researchers, research on low-level features used for malware analysis is a promising topic. The biggest problem with low-level features is that they don't bring much information to a human analyst. In this paper, we analyze potential correlation between the low- and high-level features used for malware classification. In particular, we analyze n-grams of memory access operations found in [6] and try to find their relationship with n-grams of API calls. We also compare performance of API calls and memory access n-grams on the same dataset as used in [6]. In the end, we analyze their combined performance for malware classification and explain findings in the correlation between high- and low-level features.

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

1 Introduction

Malware, or malicious software, is one of the threats that modern digitized society faces every day. The use of malware ranges from showing ads to users, spreading spam and stealing of private data, to attacks on power grids, transportation and banking facilities[23][19]. The more severe consequences of malware use, the

* 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

more likely they are a part of malicious campaign performed by an APT: Advanced Persistent Threat[9], an organization or a human that performs stealthy, adaptive, targeted and data focused [8] attack. APTs utilize different methods, tools and techniques to achieve their goals. Malware can be used at the different steps of APT kill-chain[6]: from reconnaissance and denial-of-service attacks to data stealing and creation of backdoors (for remote access) in the victim system. Since malware can be used for the variety of purposes, it is not only important to detect it, but also to be able to categorize it into different categories based on certain properties.

Malware classification (categorization) is an important step for understanding goals and methods of adversaries[1], analyzing security of systems and operations as well as for improving defense and security mechanisms. Static malware detection may fail due to obfuscation and encryption techniques used by the creators of malware. Because of this dynamic, or behavior-based detection methods are used. Moreover, malware samples are categorized into *types* and *families* by anti-virus vendors based on their behavior[6]. Hence, it is possible to assume that the use of features derived from *malware behavior* for malware classification can outperform static methods due to the nature of categories. Both static and dynamic methods need predefined sets of *features*: properties derived from a malicious file itself or its behavior.

We can divide features for dynamic analysis into two main groups: high-level (API and system calls, network activity, etc.) and low-level (memory access operation, opcodes, operations on hard-drive, etc). Generally speaking, we consider low-level features as those that directly emerge from the system's hardware[6][14][18]. Malware authors can try to conceal their malware and its behavior from anti-malware solutions and malware analysts by utilizing different techniques such as obfuscation, encryption, polymorphism or anti-debug. Despite their attempts, they can not avoid execution on the systems hardware[17][6]. That's why hardware-based, or low-level, features (since they are behavior features) are a reliable source of information for malware detection[7][18] and classification [6]. Different low-level features have been used for malware detection and classification: Hardware Performance Counters[5], frequencies of memory reads and writes[17],memory access patterns[7][6], architectural and micro-architectural events[22]. To the author's knowledge, there are no attempts to explain how particular low-level features correspond to high-level activity. Hardware-based features describe behavior of an executable on a very fine-grained level, so it is hard, by looking at the low-level feature itself, to explain which role in the behavior of an executable it has. Therefore, in this paper, we made an attempt to explain how memory access patterns correlate with the behavior of malware described by high-level features. This will make it easier for a human analyst to understand what exactly makes malware samples to be distinctive.

In order to describe our problem more generally we use an approach pictured on the Figure 1. Assume we have a dataset that contains N samples and the task is to classify them into M classes. From the dataset we can extract features of types A (e.g. low-level features) and B (e.g. high-level features). Different

feature types are derived from different sources of information: different ways to describe properties of samples in the dataset. After feature selection, features of both types can be independently used for classification of samples from the dataset. Here we suggest a hypothesis that features from feature sets A and B can correlate with each other. In this paper, we focus on finding a correlation

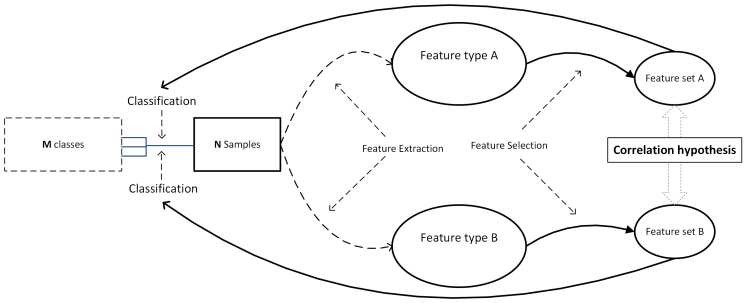

Fig. 1. Generalized problem description

between n-grams of memory access operations and API calls. To address this problem we take paper [6] as baseline. In their paper authors used patterns of memory access operations to classify malware into 10 malware families and 10 malware types. The best 29 n-grams of memory access operations are selected, and we reuse them in our case since our datasets are identical. As the high-level, or human understandable, features we decided to use n-grams of API calls since they are shown to be a reliable source of information for malware detection[4] and classification[13]. To get the most complete picture of possible correlations between memory access operations and API calls we need to search for *all-to-all* correlations. However, such an exhaustive search is computationally infeasible. In order to be able to carry out the search, we had to adjust the method, as described in Subsection 4.7. We record an execution flow of malware samples that contains memory access operations performed by single instructions as well as calls to the API functions (more details in Section 4). First, we perform classification and feature selection for n-grams of API calls. Our goal is not to study the performance of API calls for malware classification, but rather to find good and relatively short feature set of API calls n-grams as it will be more useful for research and analysis purposes. This feature set is later used in an attempt to find a possible relationship between memory access patterns and API calls, which existence or non-existence will help to reveal nature of memory access patterns that were successfully used for the same classification task.

The key findings of our paper are following. Our results show no significant correlation between information relevant for multinomial malware classification represented by best API-calls and best memory access patterns. This is important, as it shows that memory access patterns are not redundant to the higher level features such as API calls. As the result, feature set combined from memory access patterns and API calls show improved classification performance. This

contributes to better malware detection and classification as well as to the potential hardware-based security solutions.

This paper is a proof of concept, and our main goal is to address challenges and possibility of a high-level explanation of low-level events as well as creation of a stepping stone towards an explanation of a performance of low-level features in malware classification context. The remainder of the paper is arranged as follows. In Section 2 we provide an overview of the related studies and focus on the baseline paper [6]. In Section 3 we describe our problem more specifically and describe an approach for validating of our hypothesis. In Section 4 we describe our experimental design, analysis environment, methods used for feature extraction and selection, explain how we search for correlation between features of different types as well as provide terms, definitions and assumptions important for our study. Finally, in Section 5 we present results, analyze them and provide conclusions in Section 6.

2 Background

In this section, we present a short overview of articles that are related to features and methods we use in this paper. There are many papers that use hardware-based features for malware detection or categorization. For example in [5] a real-time dynamic malware detection with the use of special-purpose registers of modern CPUs as a source of features is proposed. Special-purpose registers, or hardware performance counters, are used for CPU scheduling, performance monitoring, integrity checking or workload pattern identification. In their paper, authors used four different events to construct features: retiring of a branch, load and store instructions as well as mispredicted branch instructions. With the use of various machine learning algorithms, they achieved up to 96% accuracy when classifying malicious and benign executables. Even though their dataset is small, consisting of only 20 benign and 11 malicious samples, their paper shows that hardware-based (or low-level) features can be used for malware detection.

In [17] Ozsoy et al. propose so-called malware-aware processors: processors that has a built-in hardware module that is capable of malware detection. In their work authors also mention hardware performance counters, but choose slightly different features to be used in malware detection: frequency of memory reads and writes, immediate and taken branches as well as unaligned memory accesses. They implemented a malware-aware processor in an FPGA emulator and state that their system is capable of malware detection with detection rates up to 94% and false positive rates of up to 7%. As they didn't achieve low-enough false positive rates, they propose to use malware-aware processor together with a software-based solution. They also emphasize the importance of malware-aware processor to be always on, so that it is hard to avoid detection from it.

Paper [7] is of particular interest for us, since it proposes a novel method for malware detection based on memory access pattern. In their work Banin et al. recorded sequences of memory access operations produced by malicious and benign executables. They didn't take into account addresses and values used by

these operations but utilized only a type of operation: read or write. Each sample in their dataset was launched under surveillance of specially crafted Intel Pin [12] tool and was made to produce up to 10 millions of memory access operations. Later, larger sequences of memory access operations were split into a set of overlapping sub-sequences - n-grams of a size from 16 to 96. With the use of a feature selection and various machine learning algorithms they achieved a classification accuracy of up to 98%. Results showed, that 800 memory access n-grams are enough to achieve the highest accuracy on their dataset of 455 benign and 759 malicious executables. They claimed, that n-grams of memory access operations of a size 96 extracted from only the first million of memory access operations performed by executables are reliable features for malware against benign classification. Later, in [6] they evaluated performance of 96-grams derived from the first million of memory access operations for the malware classification task. They used two different datasets, one consisted of 952 malware samples and was label according to malware types while the other had 983 malicious executables that were labeled according to malware families. With the use of feature selection, they compared results from feature sets of a size 50,000 and 29. Even though machine learning algorithms showed a decline in performance while given 29 features instead of 50000, this decline was only of a 5%. With only 29 features they achieved a classification accuracy of up to 78% for malware families and 66% for types. Even though it was far from the 98% from their previous paper they stated, that 78% can be considered good enough for 10-class classification problem. They also compared their results to the results from a paper [20], where authors used the same malware families and types but on the different dataset. In [20] Shalaginov et al. used static features, and achieved lower true positive and higher false positive rates. As we stated in the Section 1 we use paper [6] as a baseline: we use the same datasets, execution environment (Virtual Machine) and use their feature set as low-level features which origin we tend to explain. We will elaborate more on the similarities between our data collection processes in the Section 4.

Finally, we will look at some articles that make use of API-calls performed by malware during its execution for malware detection and classification. In their paper [13] Islam et al. used frequencies of occurrence of API calls during the execution of malware to detect malware and categorize into one of the 9 malware families. They also carve several static-based features such as lengths of functions or printable strings. Combining dynamic and static features they created so-called *integrated feature vector* and evaluated the classification performance of different features separately and together. They achieved a classification accuracy of up to 97% and showed that integrated feature vector can outperform other feature vectors. On its turn, Lim et al. in [16] proposed to use *k-grams* (special modification of n-grams derived from behavior automatons) of API-calls for malware detection. Even though authors didn't clearly picture the performance of their algorithm, they explained how small sequences of API calls can be used to measure the similarity between the behavior of different malware samples.

Shijo et al. [21] (similarly to [13]) utilized integrated feature vector con-

structed from dynamic and static features. As dynamic features, they used API calls n-grams of a size 3 and 4. With the use of only dynamic features they achieved a classification accuracy of up to 97% for malware against benign classification. Integrated feature vector allowed them to gain an increase in classification accuracy of up to 1%. The last paper we want to mention is [4] where Alazab et al. used API calls n-grams of a size 1 to 5 for malware detection. With the use of Support Vector Machines they achieved a classification accuracy of up to 96% and concluded, that for their dataset the best features were actually 1-grams or unigrams: n-grams of a size 1.

As we have seen, different high- and low-level features are used for malware detection and classification. Our goal in this paper is to find possible correlation between memory access patterns (low-level features) and API calls (high-level features).

3 Problem description

From the literature overview, we can state that low-level features (despite difficulties with their extraction) can be a reliable source of information for malware detection and classification. However, system counters, opcodes and memory access patterns don't give much information about malware functionality to the security analyst. An n-gram of opcodes of a size 4, when given out of context, does not reveal what it was used for by itself. The same can be said about sequence of memory access operations: it is challenging to grasp which goals were achieved by malware when a certain sequence of memory access operations was performed. For example, a typical n-gram of a size 96 of memory access operations found in [6] looks like this: $WRWRRRRR...WWWRRRRRW$. It is obvious, that such features, even if they can be effectively used for malware classification, do not bring much useful information about malware's behavior. As different papers describe the use of low-level features for malware detection and classification, to the author's knowledge there have been no attempts to find a relationship between low-level activity and high-level events such as API-calls. Because of everything said above, first, we propose two following statements:

1. N-grams of memory access operations can be used for malware classification (shown in [6]).
2. N-grams of API calls can be used for malware classification (shown in e.g. [21]).

Based on statements 1 and 2 we propose the following hypothesis: if statements 1 and 2 are true, then it should be possible to find a correlation between some of the features from both feature spaces. An approach for validating this hypothesis is described in Subsection 4.7. For example, we assume that some memory access n-grams might originate in API call n-grams. If we are able to validate this hypothesis then we will find a way to correlate sequences of memory access operations to the events of higher level which are more human understandable. If our hypothesis is rejected, then API calls and memory access n-grams are

independent features, thus combining them into an integrated feature vector should increase overall classification accuracy. Generally speaking, our goal is to check whether sequences of memory access operations that were successfully used for multinomial malware classification can be attributed to certain sequences of API calls, thus can be explained with high-level events and become more human understandable.

4 Experimental design

In this section, we present terms and definitions, provide the assumptions used and describe experimental setup and properties of datasets. Later on, we explain methods used for data collection and processing, list the machine learning and feature selection algorithms and describe the way we were searching for correlation between high- and low-level event.

4.1 Terms, definitions and assumptions

In this subsection, we provide terms and definitions and assumptions used during this study. We begin with the definitions:

- **N-gram**. An n-gram is a sub-sequence of length n of an original sequence of length L. For example if an original sequence of length L=6 [RRWRWW] is split into n-grams of length n=4(4-grams) then our n-grams set will be: RRWR,RWRW,WRWW[6]. In this example, similarly to baseline paper [6], and our paper we use overlapping n-grams: the next n-gram begins from the second element of the preceding one.
- **Memory access operations**: when an executable is *reading* from virtual memory, *read* (or R) memory operation is recorded. When *writing* to virtual memory performed by an executable, *write* (or W) memory operation is recorded.
- **API call**: or Application Programming Interface call is a call to a function provided by the operating system (Windows 7 in our case). API calls are usually used by malware and goodware to perform network, file, process and other kinds of activity.
- **Malware types and families**. Malware *types* and *families* are different ways to divide malware into categories. Malware *types* describe *general* functionality of malware: *what* it does, which *goals* it pursues. Malware *families* describe *particular* functionality of malware: which *methods* it use and *how* it pursues its goals [6]. For example, *virus, worm* and *backdoor* are malware types, while *hupigon, vundo* and *zlob* are malware families.

We continue with the following assumptions:

1. We assume that for the research and analytic purposes it is better to use smaller feature sets even if their performance in terms of classification accuracy is slightly lower [6]. For example, it is way easier to understand feature set of a size 33 that brings classification accuracy of around 70% than the one of a size 20000 with classification accuracy 73%.

2. We assume that if features from different sources (memory access operations and API calls) are related to each other, then this relationship can be found among small sets of the best features.

4.2 Experimental flow

In this subsection, we will describe our experimental flow. On the Figure 2 we picture a schematic view of our experiment. By running malware samples from two datasets (see Subsection 4.3), we collect data (memory access operations and API calls, Subsection 4.5) and perform feature construction (n-grams of API calls), later on, we use feature selection to reduce feature space and train machine learning algorithms in order to assess quality of a newly built feature vectors (Subsection 4.6). For the consistency (to the baseline paper) reasons, we run malware samples until they generate 1,000,000 of memory access operations. Some samples stop execution before they generate the desired amount of memory access operations, but we keep such data as is since this is a real-world scenario where one can't expect malware to produce as much traces as needed. While running malware, we record memory access operations and API calls (if present) for every executed instruction. From the literature review we understood, that API call n-grams of a size 3 and 4 are the most promising features. However, we also decided to use n-grams of length 8 in order to get a slightly more complete picture of API calls n-grams capabilities for malware classification. This also gives us more data to use in the search for correlation between memory access patterns and API calls. The number of n-grams is quite big, so in order to pursue one of our goals (shorter and more understandable feature set) we perform **feature selection** to reduce the dataset. As well as authors of [6] we used Correlation Based feature selection [11] from machine learning tool Weka [3] as it showed quite good performance while reducing the size of a feature set in several times of magnitude. After getting a reduced feature set, we store data in the format that can be used for training of machine learning models. In our case, similarly to the baseline paper, as feature values, we store only the fact of presence (1 or 0) of a certain feature in the behavior of a malicious sample. The logic here is similar to [6]: in contrast to other articles, where authors rely on frequencies of appearance of certain features, we want to find features that work regardless the time malicious executable has run. Our data looks like a **bitmap** of presence, where each row represents a single malicious sample, first column represents a category of a sample (family or type) and the rest of the columns represent features. Cells contain *1* if a certain feature is present in the behavior of malware and *0* if not. The bitmap of presence is later used for training the **machine learning** models (see Subsection 4.6), which classification performance (classification accuracy) is compared with the one from baseline paper. Having API call n-grams as features, we later search through the entire records or behavior data from each malware sample in order to find whether these n-grams are related to the 29 memory access n-grams derived by authors of [6]. We elaborate on the search technique in the Subsection 4.7.

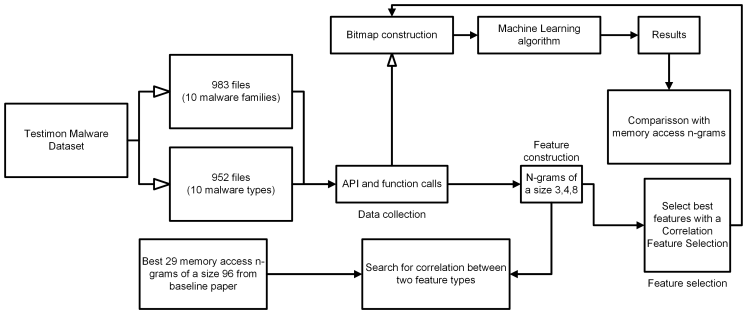

Fig. 2. Detailed experimental flow

4.3 Dataset

Similarly to [6], our two datasets are derived from the original dataset collected under the initiative of Testimon [10] research group. It consists of 400k malware samples: malicious PE32 executables gathered from VirusShare[2]. Initial dataset was used for research purposes and is described in [20]. Both our datasets are the same as in baseline paper [6]. The authors of a baseline paper provide a detailed description of their datasets, while we focus only on the most important properties of these datasets. First of all, one dataset (952 files) has malware samples that are labeled according to ten types: *backdoor, pws, rogue, trojan, trojandownloader, trojandropper, trojanspy, virtool, virus, worm*. Secondly, another dataset (983 files) has its malware samples label according to ten families: *agent, hupigon, obfuscator, onlinegames, renos, small, vb, vbinject, vundo, zlob*. The choice of categories was made by the simple rule: 10 most prevalent categories in the original dataset were chosen. To simplify automated malware analysis (see Section 4.4) sample were chosen to be without anti-VM and anti-Debug features. As described in [6], dealing with anti-analysis functionality of malware is out of scope in such research, since their goal was to study a possibility of malware classification with memory access patterns as features. The distributions of categories within datasets are almost uniform, so we assume that datasets are nearly balanced, so there is no need to study the influence of categories distribution on the results of an assessment of machine learning models.

4.4 Analysis environment

Our analysis environment was almost identical to the one in [6], apart from different versions of host OS and VirtualBox. We assume that these changes will not influence the results of the experiments since hardware and guest OS are identical. We run our experiments on Virtual Dedicated Server (VDS) with Intel Core CPU running at 3.60GHz, 4 cores, SSD RAID storage and 32GB of virtual memory. As a main operating system, Ubuntu 18.04 64bit was used. Additionally, Intel Pin 3.6[12], Python 2.7 and VirtualBox 5.2.22 were used.

Windows 7 32-bit was installed on the VirtualBox virtual machine as a guest OS. We used a virtual machine as an isolated environment to run malware together with a specially crafted Intel Pin tool. The virtual machine is reverted to the same snapshot before each run, so we avoid the influence of the environment on the results of data acquisition. To be consistent with a baseline paper, we choose the 32-bit version of Windows 7.

4.5 Data collection

We focus on "correlating" the n-grams of memory access operations with n-grams of API calls. We need to: a) record memory access operations produced by malware b) record calls to API functions. The first task is the easiest one. With the use of dynamic binary instrumentation framework Intel Pin, one can put instrumentation on each executed instruction and record memory access operations performed by it. For the consistency reasons, we chose the same amount of memory access operations to record as was used in [6]. A malicious executable run until it produces 1 million of memory access operations. As it was shown in the previously published papers, this is not only enough to reveal maliciousness of an executable [7] but also to perform multinomial classification of malware into categories and types[6]. The second task is more difficult. When a *call* instruction is performed it only contains an address of a function. In order to get its name from a library, one should find which one of the export symbols correspond to a certain address. Moreover, some native Windows libraries perform inter- and intra-modular calls not to the functions themselves (a call to a first instruction of a function) but to the subroutines within these functions. Most of the papers that use dynamic API call sequences do not describe how they treat such calls: it is not clear whether they record or just ignore them. In this paper, we treat a call to a first instruction of an API function and a call to a subroutine in an API function equally. Our reason for this is that if a logic of an executable requires such calls to be done and we can collect this information, it may improve the understanding of malware's current execution goals and context.

The call instruction can be used to invoke internal (to an executable itself) function. It is usually impossible to derive a name of an internal function of an executable (unless you have debug file, which is not the case in malware analysis), so we store a name of a section where a function of interest is placed. We also keep this information and treat such calls equally to the API calls. Having raw data recorded, we split a sequence of API calls generated by each malicious sample into **n-grams**.

For better analysis capabilities as well as future work we record additional information for *each* instruction executed after launching a malware sample. A real example of raw data is present in the Listing 2 in Appendix A. In order to record this data, we created an Intel Pin based tool that is launched together with each sample from a dataset. A tool records all data into a file and stops if an executable generated 1 million of memory access operations. Some samples generate less memory activity than others, but we consider it a real-world scenario where one can't rely on malware to generate a particular amount of data.

From the raw data we extract names of the called functions, store them into the sequence according to their execution order and split the sequence into n-grams of a different size. For example, one of the API call n-grams of a size 4 derived from malware families dataset looks as following: *memset,GetModuleHandleW,ferror,_freea*. From the raw data, we extract API calls and memory access operations, that are later used in training the machine learning models and searching for mutual correlation.

4.6 Machine learning algorithms and feature selection

For the consistency reasons, we chose the same machine learning (ML) algorithms as in [6]: k-Nearest Neighbors (kNN), RandomForest (RF), Decision Trees (J48), Support Vector Machines (SVM), Naive Bayes (NB) and Artificial Neural Network (ANN). The following parameters (default for Weka[3] package) were used for ML algorithms: kNN used *k=1*; RF had 100 random trees; J48 used pruning confidence of 0.25 and a minimum split number of 2; SVM used radial basis as function of kernel; NB used 100 instances as the preferred batch size; ANN used 500 epochs, learning rate 0.3 and a number of hidden neurons equal to half of the sum of a number of classes and a number of attributes. In order to assess the quality of machine leraning models we used 5-fold cross validation, and chose accuracy (number of correclty classified instances) as the measure of evaluation. To reduce the feature set we used Correlation Based feature selection from Weka. Correlation-based feature selection [11] is an algorithm that chooses a subset of features that have the highest correlation with classes, lowest correlation with each other and give the best merit among other possible subsets. First reason to choose this feature selection method as it helped authors of a baseline paper to go from 50 thousands of features to just 29, so we wanted to get a number of features of nearly the same magnitude. Second reason is that one of our goals is to have relatively short feature set that can be easily analyzed by a human analyst.

4.7 Correlating features derived from different sources

In this section, we present a method to validate our hypothesis presented in Section 3. There are several approaches that can be used to validate our hypothesis. The first one is the most obvious: create the *entire* feature sets for memory access operations and API calls n-grams and find correlations between them (*all-to-all* approach). This approach will reveal the full picture of correlation between the two feature types. But it also has one major drawback, that makes its use almost impossible. The entire feature space of memory access n-grams in [6] consists of 15 millions distinctive features for malware families dataset. Finding their correlation with around 12 thousands of API calls 3-grams (see Subsection 5.1) can not be finished in feasible time. Slightly less time consuming variant is to search for correlation between the best memory access operations features and the entire feature space of API calls n-grams (*best-to-all* approach). This method would provide a less complete overview over the possible correlations, but would

still be very time consuming, and is left for the future work. To some initial results we used a *best-to-best* approach: instead of taking the entire feature sets of memory access operations and API calls, we use only the best features out of both feature spaces. This approach allowed us to finish the experiments in feasible time, but also has some limitations that will be discussed in the following sections. As this paper is aiming to provide a proof of concept for searching for correlations, we believe that this approach properly fits our purposes.

One of the challenges we met during this research is how to correlate a certain n-gram of memory access operations to an n-gram (n-grams) of API calls. First of all, we need to locate a place in a raw data, where a certain n-gram of memory access operations is found. To do this, we iterate over the raw data, collect memory access operation into a buffer of a size 96 (see Section 2) and check if the pattern in the buffer is found among one of the features taken from the baseline paper. If match occurs - we save the position where memory n-grams starts and begin the search for API call n-gram. There can be various approaches to this and we selected the following one, as it brings wider coverage of execution flow. To state that a certain memory access n-gram is related to an API call n-gram we use the following criteria:

1. If the beginning of memory access n-gram lays after first call in API calls n-gram and before the call that follows current n-gram - these memory and API call n-grams correlate. In this case we assume that memory access n-gram is correlated to an API calls n-gram.
2. For any other case we state, that memory access and API call n-grams are not correlated.

The above mentioned criteria works as shown in Figure 3 where we present a simplified version of our data. On this Figure, memory access n-gram of a size 96 correlates with API calls 3-grams [APIcall_1, APIcall_2, APIcall_3], [APIcall_2, APIcall_3, APIcall_4] and [APIcall_3, APIcall_4, APIcall_5] but does not correlate with [APIcall_4, APIcall_5, APIcall_6]

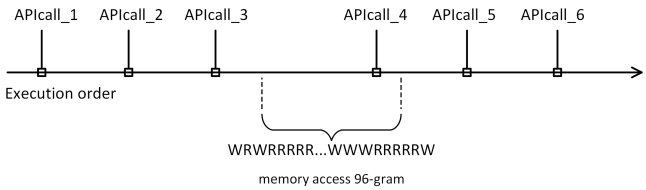

Fig. 3. Correlation between API calls and memory access n-grams

5 Results and analysis

In this section we provide the results of feature selection and classification for API calls n-grams, compare them to the results achieved with memory access

n-grams from [6] and evaluate our findings in correlating these two types of features.

5.1 API call n-grams for malware classification

From the raw captured data we extracted 12818 3-grams, 17407 4-grams and 33900 8-grams in the malware family dataset and 17252 3-grams, 24054 4-grams and 49513 8-grams in the malware types dataset. Using correlation based feature selection allowed us to reduce number of features to the following: 23 3-grams, 33 4-grams and 47 8-grams in the malware family dataset and 52 3-grams, 62 4-grams and 76 8-grams in the malware types dataset. The reduction of feature vectors worked similarly to the baseline paper: we went down from *tens of thousands* to *less than hundred* features. As assessment of classification performance of API call n-grams is not the main goal of this paper, we provide only the results for reduced feature sets. However, we performed classification on the full feature sets and their classification accuracy was only a few percents higher then in reduced feature sets. It is again similar to [6], so we assume that it is possible to compare newly acquired feature set with the one from [6]. In the Table 1 the classification accuracy achieved by different machine learning algorithms is presented. On the left and right sides of the table we present the results achieved on malware families and malware types datasets respectively. First row represent results achieved with n-grams of memory access operations of a size 96 from [6]. We name this feature type *Mem96*. Rows from 2 to 4 represent results achieved with API calls n-grams of sizes 3,4, and 8. We name them *API3,API4* and *API8* respectively. As we can see, most of the time API calls n-grams performed on the same or even higher level then memory acess n-grams for the malware families dataset. In contrast, performance of API calls n-grams for malware types dataset most of the time was lower then the one by memory access n-grams. These results help us to prove Statement 2 from Section 3. In the Table 1 we use bold font in order to underline best classification accuracy for a certain type of features. It is also worth mentioning, that in general API calls n-grams of a size 4 performed better then other types of n-grams. We have to draw an important conclusion from the results we achieved with API calls n-grams. Classification performance of less then a hundred API calls n-grams are comparable to those achieved with tens of thousands of memory access n-grams in [6].

5.2 Correlating memory access and API call n-grams

The results we got were quite surprising. With the feature selection, we used and feature correlation search method we described in Subsection 4.7 we found no correlation between memory access n-grams and API call n-grams for malware *types* dataset. For malware *types* dataset our hypothesis about the correlation between features derived from different sources was rejected. Results for malware *families* dataset was not much different. One memory access n-gram was found to be related to a certain API calls 3-gram in different malicious samples, and

Table 1. Classification accuracy for baseline feature set, API call n-grams feature sets and combined feature sets.

		Feature set size		Families						Types					
#	Feature type	Fam.	Typ.	kNN	RF	J48	SVM	NB	ANN	kNN	RF	J48	SVM	NB	ANN
1	Mem96	29	29	**0.784**	0.781	0.769	0.740	0.724	**0.784**	**0.668**	**0.668**	0.626	0.584	0.498	0.617
2	API3	23	36	0.775	0.780	0.746	0.709	0.652	0.774	0.616	0.631	0.587	0.533	0.521	0.607
3	API4	33	46	**0.813**	0.810	0.792	0.765	0.677	0.805	0.636	0.636	0.604	0.541	0.566	0.616
4	API8	47	67	0.799	0.801	0.784	0.751	0.694	0.797	0.643	**0.660**	0.605	0.537	0.562	0.615
5	API3+Mem96	52	65	0.834	0.856	0.817	0.781	0.711	0.845	0.680	**0.700**	0.641	0.573	0.556	0.682
6	API4+Mem96	62	75	0.838	**0.859**	0.824	0.786	0.716	0.842	0.680	0.694	0.662	0.580	0.566	0.676
7	API8+Mem96	76	96	0.832	0.845	0.801	0.773	0.717	0.835	0.667	0.687	0.649	0.586	0.575	0.686

the other was found to be related to two API calls 4-grams in different malicious samples as shown in Listing 1.1. So our initial hypothesis was mostly rejected for malware *families* dataset as well. Having this information we decided to create integrated feature sets by combining memory n-grams feature set with API call n-grams feature sets. We analyze the performance of an integrated feature set in the next subsection.

5.3 Performance of integrated feature sets

We found an idea about combining features of different types into an integrated feature vector from [16]. In the Table 1 we present classification accuracy achieved with integrated feature vectors. In the rows 5 to 7 results of combining memory n-grams feature vector with all API call n-gram feature vectors are present. As we can see, with several exceptions, most of the time integrated feature vector outperform separate feature vectors. Moreover, with an integrated feature vector (which size didn't exceed 100) we achieved a classification accuracy of 85.9% for families and 70% for types, which are higher than respective 84.5% and 66.8% achieved in [6] with use of 50,000 memory access n-grams. This indicates that combining API call and memory access n-grams does not bring redundant information which often results in lower classification accuracy[15]. Even though our hypothesis was rejected, the increased classification accuracy of an integrated feature set show that our correlation search method (Subsection 4.7) was correct.

5.4 Discussion and analysis of correlation findings

As we already said, for the two entire datasets, we found only two memory access n-grams that we found to be related to the API call n-grams from a reduced feature set. In the Listing 1.1 we show found relationships of memory access n-grams and API call n-grams. As we can see, for our family dataset, a memory access n-gram is related to only one API call 3-gram. However, the relationship between memory access n-gram and API 4-grams can look a little bit more complicated. We found that the same memory access n-gram can originate from different API call n-grams. But this can be easily explained after analysis of the API 4-grams themselves. As we can see, these two API call 4-grams can easily

overlap: last three API calls of the first 4-gram can be the first three API calls in the second 4-gram. And as we described in Subsection 4.7 if the beginning of memory access n-gram lays between first and last call in the API call n-gram - these memory and API call n-grams are related. So it is easy to understand now, that if selected API call n-grams are overlapping - the same memory access n-gram can originate from both of them.

```
Memory ngram WWWRRRRRRRWRRWRRWWWWRRWWWWRWRRRRRWRRWWRWWR
    ↪ RRWWWRRRWWRRRRWRWRRRRRRWRRRRRRRRRRRWRWWRWWWWWRRRWRRRWW
is related to the following API call 3-gram: RtlTryEnterCriticalSection,
    ↪ RtlLeaveCriticalSection,memset

Memory ngram WRWRRRRRRWRRWRWRRRWRRRRRRWRRRWRWRRRRRRWRRR
    ↪ RRRRRWWRWRRRWWWRWWRWRRRRRRWRWWRWRWWRRWWRWWWWWRRRRRW
is related to the following API call 4-grams: RtlEnterCriticalSection,
    ↪ RtlEnterCriticalSection,RtlEnterCriticalSection,
    ↪ RtlCompareUnicodeStrings
and RtlEnterCriticalSection,RtlEnterCriticalSection,RtlCompareUnicodeStrings,
    ↪ RtlCompareUnicodeStrings
```

Listing 1.1. Memory access n-grams and correlated API call n-grams from malware families dataset

As a way to improve our search technique it is possible to split a sequence of API calls into *non-overlapping* n-grams. However, in some rare cases, it might result in several memory trace n-grams to be related to a single API call n-gram. Another reason of small correlation findings can be a *best-to-best* approach that we chose for correlation search. Utilizing a *best-to-all* approach together with an in-depth explanation of correlated API calls n-grams is one of the priority goals for the future work.

There is one thing that is important to look at after presenting relatively poor correlation findings. As we have written above, we trace the execution of malware samples until they generate 1 million of memory access operations. Some samples produce less than the expected number of memory access operations. It is important to understand, that the execution of PE file does not start from the main module of a file. Instead, different API calls are invoked by an operating system (they still executed under the process of malware, so we trace them anyway) in order to prepare an execution environment. The amount and type of calls performed before execution of main logic (main module) of a malware depends on the way an executable was compiled and the resources it needs for execution. It is important to notice, that even if an API call is made from the main module of an executable, its instructions will be corresponded to the external module(e.g. ntdll.dll). To go deeper into this problem first we counted the number of instructions executed by malware from its main module and divided it by the total number of instructions in the trace. Amount of instructions performed from the main module (defined by the malware directly) ranges from 0% to 99.9% with an average of around 20%. It means that some samples didn't even reach to the execution of their main module. From first glance, it should have led to the sample being indistinguishable from each other. Nevertheless, as we already said, this platform-specific (PE is an executable format used in Windows) preamble depends on the properties of the file. Another thing that we checked was the percentage of *call* instructions executed from the main module. These numbers range from 0% to 8% with an average of up to 1.5%. From what

was said above, and from additional data analysis, it is possible to draw the following conclusion: most of the API calls in our experiments didn't originate from the main modules of executables. Moreover, as the number of instructions performed from the main modules is relatively low, the memory access n-grams from [6] also did not originate from main modules either. The first conclusion that can be drawn from this is that some malicious executables can be categorized into families and types (with an accuracy we achieved) based on the activity they produce before executing their main logic. On the first hand, these are very promising results since detection mechanisms based on the features used in this paper can potentially detect malware before anything malicious is done. However, we didn't study what changes to our victim system our malicious samples did. So this is clearly a question for future research. On the other hand we might have actually detected malicious behavior by itself: there are known malware samples that achieve its goals from TLS callbacks or by inserting malicious code into legitimate DLLs or executables (other than malware's main modules) and performing direct jumps or calls to the infected parts of legitimate DLL's or executables.

As a final remark to this subsection we suggest the following solution to the questions we outlined in the beginning. To understand if API calls that actually produce memory access patterns from [6] can be useful for malware classification we have to use only a certain amount of API calls made around a place from where memory access n-gram is originated from. Based on these API call sequences we may try to find features that are relevant for malware classification. This is planned to be done in the future work, as the amount of "API calls made around a place from where memory access n-gram is originated from" has to be found after a number of experiments. Also, the type of features in this future case has to be discussed as well.

6 Conclusions

In this paper, we examined the nature of memory access n-grams that were successfully used for malware classification by authors of [6]. We also attempted to understand the relationship between those low-level features and high-level activity patterns such as API call n-grams. Our findings showed no significant correlation between the best n-grams of memory access operations and the best n-grams of API calls (at least under our experimental design). We also showed that API calls n-grams can be used for malware classification on the dataset from [6] and found that combining features derived from different sources (low- and high-level activity) can bring improvement in classification accuracy. While analyzing our data we concluded, that both low- and high-level features used in our experiments often have their origin outside of the main module of an executable. This paper brings important findings and outlines the direction of future research about the use of low-level features in malware analysis.

Appendix A Raw data sample

In this Appendix we present a sample of a raw data gather during our experiments. We also explain each field included in the data.

1. Opcode id: each opcode is given a unique identifier. If this opcode is executed again (e.g. in a loop), it will receive the same id.
2. Module name: a name of a module where current instruction is executed, It can be a name of a library or a name of an executable itself.
3. Section name: a name of a section in executable file or library where current instruction is executed. Often it will be *.text* or *CODE*, however it some cases (especially with malware) a name of an executable section can be different from standard.
4. Current function name: if a function name of a current instruction can be found we record it to understand *which* function performed a certain part of logic.
5. Opcode: text representation of an assembly instruction together with arguments but without arguments values.
6. Type of module: whether an instruction is executed from the main module of executable under analysis or from the external library.
7. Memory operations: memory operations performed by an instruction. Only *read* or *write* without addresses and values.
8. Name of a function being called: if a current instruction is *call* - a name of a function is being stored.

A real example of raw data is present in the Listing 2. The first line represents header: names of fields are in the same order as in the list above.

```
OPID;MODULE;SECTION;ROUTINE;OPCODE;MODULETYPE;MEMOPS;ROUTINETOCALL
6712;C:\Windows\SYSTEM32\ntdll.dll;.text;RtlInitializeExceptionChain;xor ecx,
    ↪  ecx;isNotMainModule;;
6713;C:\Windows\SYSTEM32\ntdll.dll;.text;RtlInitializeExceptionChain;call eax
    ↪ ;isNotMainModule;W;BaseThreadInitThunk
6369;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;mov edi, edi;
    ↪ isNotMainModule;;
6370;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;push ebp;
    ↪ isNotMainModule;W;
6371;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;mov ebp, esp;
    ↪ isNotMainModule;;
6372;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;test ecx, ecx
    ↪ ;isNotMainModule;;
6373;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;jnz 0
    ↪ x76f4853d;isNotMainModule;;
6374;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;push dword
    ↪ ptr [ebp+0x8];isNotMainModule;RW;
6375;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;call edx;
    ↪ isNotMainModule;W;unnamedImageEntryPoint
6714;C:\Users\win7\Documents\malware_PE32\1b6142e3c80362a3f49666856f330510;.
    ↪ duciuni;unnamedImageEntryPoint;inc ebx;isMainModule;;
6715;C:\Users\win7\Documents\malware_PE32\1b6142e3c80362a3f49666856f330510;.
    ↪ duciuni;unnamedImageEntryPoint;pushad ;isMainModule;W;
```

Listing 2. Raw data sample

References

1. Types of malware. `https://usa.kaspersky.com/resource-center/threats/types-of-malware`, accessed: 17.03.2019

2. Virusshare.com. http://virusshare.com/, accessed: 12.03.2019
3. Weka: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/ (2019), accessed: 2019-03-312
4. Alazab, M., Layton, R., Venkataraman, S., Watters, P.: Malware detection based on structural and behavioural features of api calls (2010)
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. 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
7. Banin, S., Shalaginov, A., Franke, K.: Memory access patterns for malware detection. Norsk informasjonssikkerhetskonferanse (NISK) pp. 96–107 (2016)
8. Cole, E.: Advanced persistent threat: understanding the danger and how to protect your organization. Newnes (2012)
9. Enterprise, W.A.M.T.Y., Hoglund, G.: Advanced persistent threat
10. Group, T.R.: Testimon research group. https://testimon.ccis.no/ (2017)
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 (2019)
13. Islam, R., Tian, R., Batten, L.M., Versteeg, S.: Classification of malware based on integrated static and dynamic features. Journal of Network and Computer Applications **36**(2), 646–656 (2013). https://doi.org/10.1016/j.jnca.2012.10.004
14. 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
15. Kononenko, I., Kukar, M.: Machine learning and data mining: introduction to principles and algorithms. Horwood Publishing (2007)
16. Lim, H.i.: Detecting malicious behaviors of software through analysis of api sequence k-grams i (2016). https://doi.org/10.13189/csit.2016.040301
17. 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
18. 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
19. 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)
20. Shalaginov, A., Grini, L.S., Franke, K.: Understanding neuro-fuzzy on a class of multinomial malware detection problems. In: Neural Networks (IJCNN), 2016 International Joint Conference on. pp. 684–691. IEEE (2016). https://doi.org/10.1109/ijcnn.2016.7727266
21. Shijo, P., Salim, A.: Integrated static and dynamic analysis for malware detection. Procedia Computer Science **46**, 804–811 (2015). https://doi.org/10.1016/j.procs.2015.02.149

22. Tang, A., Sethumadhavan, S., Stolfo, S.J.: Unsupervised anomaly-based malware detection using hardware features. In: International Workshop on Recent Advances in Intrusion Detection. pp. 109–129. Springer (2014). https://doi.org/10.1007/978-3-319-11379-1_6
23. 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)