Discovering Similar Malware Samples Using API Call Topics

Akinori Fujino
Dept. of Communication Engineering,
Waseda University
3-4-1 Okubo Shinuku, Tokyo, Japan
Email: fujino@nsl.cs.waseda.ac.jp

Junichi Murakami
FFRI, Inc.
1-18-18 Ebisu Shibuya, Tokyo, Japan
Email: murakami@ffri.jp

Tatsuya Mori
Dept. of Communication Engineering,
Waseda University
3-4-1 Okubo Shinuku, Tokyo, Japan
Email: mori@nsl.cs.waseda.ac.jp

***Abstract*—To automate malware analysis, dynamic malware analysis systems have attracted increasing attention from both the industry and research communities. Of the various logs collected by such systems, the API call is a very promising source of information for characterizing malware behavior. This work aims to extract similar malware samples automatically using the concept of "API call topics," which represents a set of API calls that are intrinsic to a specific group of malware samples. We first convert Win32 API calls into "API words." We then apply non-negative matrix factorization (NMF) clustering analysis to the corpus of the extracted API words. NMF automatically generates the API call topics from the API words. The contributions of this work can be summarized as follows. We present an *unsupervised* approach to extract API call topics from a large corpus of API calls. Through analysis of the API call logs collected from thousands of malware samples, we demonstrate that the extracted API call topics can detect similar malware samples. The proposed approach is expected to be useful for automating the process of analyzing a huge volume of logs collected from dynamic malware analysis systems.**

I. Introduction

Understanding malware behavior is a crucial first step toward building a system that can detect newly generated malware samples. To this end, dynamic analysis of malware has attracted significant attentions because it enables malware analysts to capture malwares' behaviors without employing the analysis of highly complex obfuscation techniques [10]; i.e., dynamic analysis extracts actual executing instructions of obfuscated codes, which require time-consuming efforts to analyze with static malware analysis. By carefully examining logs collected from a dynamic malware analysis system, an analyst may be able to identify new patterns that can be used to detect *unknown* malware samples.

The number of malware samples is becoming increasingly large. The Kaspersky Lab has reported that more than 315,000 new malicious files are detected every day [15]. Therefore, automating the process for dynamic malware analysis is essential [10]. Cuckoo Sandbox [2] is one of the automated dynamic malware analysis systems. In general, such dynamic malware analysis systems automate a series of malware analysis processes, such as tracing Windows API calls, recording created files, dumping full memory, and monitoring network activities. In addition, some systems use an external virus checker, such as VirusTotal [4], to test the input samples. Numerous previous studies have revealed that, of the various outputs from automated dynamic malware analysis systems, API call logs are the most promising sources of information for characterizing malware behaviors [6], [7], [27]. In fact, as will be discussed in Section II, API call logs are significantly larger than other logs. We note the limitation of analyzing API calls collected from a malware sandbox. It is well known that some malware samples are protected with anti-VM techniques to evade retro-engineering [9]. In such cases, a malware sandbox may fail to extract API calls. Clearly, this limitation affects our approach. However, we believe that the recent advances in the dynamic malware analysis, e.g., anti-anti-VM techniques, will be useful to mitigate the limitation [11], [14].

The aim of this work is to automatically compile signatures that can be used to detect *unknown* malware samples from a huge volume of API call logs collected from a large number of malware samples. Specifically, we aim to extract "API call topics," which can be used as signatures for discovering *unknown* malware samples that are *similar* to existing ones. The key ideas are to characterize API calls using the bag-of-words (BoW) model and to make use of non-negative matrix factorization (NMF), which is a soft clustering algorithms. In other words, we adopt an *unsupervised* learning approach. As will be discussed in Section III-C, NMF can extract topics from the corpus of API calls automatically. We demonstrate that the extracted API call topics can be used as signatures that detect *unknown* but similar malware samples. We evaluate our approach with large-scale API call logs generated by executing thousands of distinct and randomly selected malware samples.

Our work is not the first to apply machine-learning technique to analyzing API calls. In fact, several studies on analyzing API calls have used the machine learning-based approaches [5]–[7], [10], [18], [24]. We leave the technical comparison between these studies and ours in section V. Here we note that this work is distinguishable from other studies in that it enables us to *automatically* extract signatures that can be used to identify samples similar to the existing ones.

Our main contributions can be summarized as follows:

- We present an *unsupervised* approach to extract API call topics from a large API call corpus.
- We demonstrate that the extracted API call topics can detect similar malware samples.

We believe that the proposed approach is useful for automating the process of analyzing a huge volume of logs collected from dynamic malware analysis systems.

TABLE I. TOP-10 ANTIVIRUS CHECKERS IN DETECTION RATES FOR FFRI DATASET 2013.

Antivirus checkers	Detection rates (%)
Kaspersky	76.5
GData	74.4
AntiVir	74.3
Ikarus	74.0
BitDefender	73.5
F-Secure	72.0
Panda	71.1
AVG	71.1
VIPRE	70.8
ESET-NOD32	70.5

The remainder of this paper is organized as follows. We first describe the dataset we used for analysis in Section II. Section III presents the proposed approach and its evaluation, and Section VI concludes this paper.

II. DATASET

Of the various dynamic malware analysis systems [10], this work adopts Cuckoo Sandbox [2] because of its growing popularity in both industry [8] and the research community [12], [23]. We use the dataset called FFRI Dataset that is a part of the MWS Dataset [1]. The MWS Dataset is a collection of datasets shared among the Japanese industry and security researchers. The FFRI Dataset contains the analysis results of Cuckoo Sandbox for a given set of malware samples.

The malware executable files used in the FFRI Dataset were randomly sampled from a huge collection of malware samples obtained from various sources, such as drive-by-downloads or honeypots. All the collected samples were manually inspected by experts and labeled as malware. The data consists of Cuckoo Sandbox logs of 2,641 malware samples collected from September 2012 to March 2013. The format of all malware samples was the Portable Executable (PE) format used in Windows operating systems. These malware samples were executed on Cuckoo Sandbox in 90 seconds and generated 1.7 GB of dynamic analysis logs in JSON format. The executed samples were able to access to the network and network services like DNS and Web in a controlled environment. We observed that some samples did not show any activities when we executed them in a testbed. We eliminated such samples from our analysis. Of all the Cuckoo sandbox logs in the FFRI dataset, 87.05% of lines were attributed to those for API calls.

Figure 1 shows the CDF of the number of API calls per malware sample. The dashed line shows the mean value (644.9). Whereas the majority of the samples had more than 400 API calls, a few samples had only a small number of API calls. Malware samples that exhibited a very low number of API calls likely failed to execute correctly, therefore, we pruned malware samples that had fewer than 10 API calls. As we mentioned before, it is possible that these malware samples were protected with anti-VM techniques. We leave such cases for future study.

Like other dynamic malware analysis systems, Cuckoo Sandbox provides a module that searches MD5 digests of the input samples from VirusTotal [4], which is a publicly available web portal that provides the results of various antivirus checkers. VirusTotal detection results were recorded for each input malware executable file. On average, the number of antivirus checkers used for each malware sample was greater than 40. Table I presents the breakdown of detection rates for the Top-10 antivirus checkers.

TABLE II. SUMMARY OF DATASET

Data	Period	# of samples
D1	Sep 2012 – Feb 2013	962
D2	Mar 2013	452
Total	Sep 2012 – Mar 2013	1,414

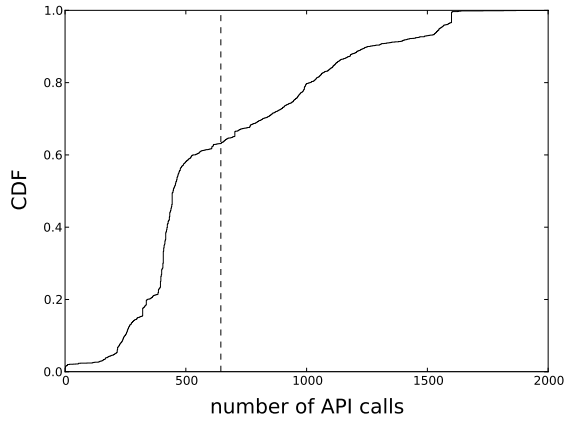

Fig. 1. CDF of the number of API calls per malware sample.

To evaluate our analysis, we made use of labels generated by the Kaspersky antivirus checker [3], which exhibited the highest detection rates with our dataset. Here we note the two limitations when we use the labels generated by antivirus vendors. First, even the antivirus checker with the highest detection rate missed roughly 20 % of the malware samples. Because all the samples used for this analysis are actually malware, this observation indicates that the results obtained by a single antivirus checker are insufficient to detect malware adequately. Second, Mohaisen [21] empirically revealed that the labels created by antivirus checkers are not complete nor consistent with each other. Given these limitations in mind, we make use of labels as a *hint* to objectively interpret the clustering outputs produced by our approach.

We now show the naming convention used by Kaspersky. According to Ref. [25], it is defined as follows:

```
[Prefix:]Behaviour.Platform.Name[.Variant]
```

`Prefix` and `Variant` are optional records, which represent a subsystem that detects a malware and variants of a malware, respectively. Notice that if `Prefix` is set to `HEUR`, the malware sample was detected with some heuristics; thus, its origin is not clear in general. `Behavior` represents the activity of samples such as Viruses, Worms, Trojans, Malicious Tools, Adware, Riskware, Pornware, and etc. `Platform` represents the operating systems, e.g., `Win32`, `Linux`, `multi`, and etc. Our dataset includes only `Win32`. Finally, `Name` represents the official name of malware family.

To examine the similarity of malware samples grouped by the extracted API call topics, we used the antivirus checker labels. For this reason, we pruned malware samples that were *not* detected by the antivirus checker. We also pruned malware

TABLE III. DETECTED MALWARE FAMILIES (**D1**).

Detected malware families	# of samples
Worm.Win32.WBNA	152 (15.8%)
Trojan.Win32.Jorik.Vobfus	83 (8.6%)
Worm.Win32.Vobfus	56 (5.8%)
HEUR:AdWare.Win32.iBryte	36 (3.7%)
Trojan-PSW.Win32.Tepfer	34 (3.5%)
Trojan-Spy.Win32.Zbot	30 (3.1%)
Worm.Win32.VBNA	25 (2.6%)
Trojan.Win32.Agent	24 (2.5%)
Trojan.Win32.VBKrypt	20 (2.1%)
Trojan.Win32.SelfDel	19 (2.0%)
Others	483 (50.3%)
Total	962

TABLE IV. DETECTED MALWARE FAMILIES (**D2**).

Detected malware families	# of samples
Worm.Win32.WBNA	123 (27.2%)
Worm.Win32.Vobfus	28 (6.2%)
Trojan.Win32.Jorik.Vobfus	27 (6.0%)
Trojan.Win32.SelfDel	24 (5.3%)
Trojan-PSW.Win32.Tepfer	21 (4.6%)
Trojan-Spy.Win32.Zbot	13 (2.9%)
Worm.Win32.VBNA	11 (2.4%)
Backdoor.Win32.Simda	8 (1.8%)
Trojan.Win32.Midhos	7 (1.5%)
Trojan.Win32.VB	7 (1.5%)
Others	183 (40.6%)
Total	452

samples whose family names were detected as `Generic`, which indicates that the detected samples were *not* precisely classified by the commercial antivirus checker. However, the proposed scheme can be applied to such samples.

To evaluate the proposed detection scheme, we divided the remaining data into two subsets, datasets **D1** and **D2**. Table II summarizes the datasets used for our analysis. Tables III and IV present breakdowns of the labels generated by the antivirus checker. For brevity, we omit the differences among variants.

III. METHDOLOGIES FOR ANALYZING API CALL TOPICS

Here, we describe the methodologies developed in this study. First, we present data preprocessing schemes required to apply API call logs to the NMF clustering algorithm. Next, we show an overview of the NMF algorithm with some actual clustering examples. We also show the way of choosing an adequate parameter for the NMF algorithm.

A. Creating feature vectors with API words

This subsection describes the process for converting API calls into feature vectors that can be applied to the clustering algorithm. We first convert an API call into an "API word." We then compile feature vectors from the set of generated API words.

Figure 2 presents an example of the log output for a single API call. Note that because of space limitation, the original JSON formatted Cuckoo Sandbox log has been modified slightly without changing the content. In Fig. 2, `api` and `arguments` represent the API function name and its arguments, respectively. `arguments` consists of argument name(s) (`name`), and argument value(s) (`value`). We use these three variables to convert an API call into an "API word". This

```
"category": "system",
"status": "FAILURE",
"return": "0xc0000135",
"timestamp": "2013-02-28 12:03:55,656",
"thread_id": "432",
"repeated": 1,
"api": "LdrGetDllHandle",
"arguments":
  [{"name": "FileName",
    "value": "C:\\WINDOWS\\system32\\rpcss.dll"},
   {"name": "ModuleHandle",
    "value": "0x00000000"}]
```

Fig. 2. An example of API call log.

conversion allows us to employ several techniques used in natural language processing (NLP), e.g., the BoW model, TF-IDF weighting, etc. Note that API calls have other potentially useful features such as return values or thread IDs. However, we omitted analysis of such features in this work.

As is shown in Table V, there are several ways to convert an API call into an API word. For instance, Level 0 consists of only API function names, and Level 3 consists of the three variables mentioned previously. In Table V, `n` and `v` denote the `name` and `value`, respectively. In the definition of Level 2 , `m()` is a masking function that is applied to hex strings. This masking function eliminates the high variability of values, such as memory addresses. After extensive experiments, we found that the Level-3 definition performed the best among the four levels. However, because of space limitations, we omit details of these experiments. Throughout this paper, we adopt Level-3 API words.

Using the API words, we generate a feature vector for each malware sample. To achieve this, we use the BoW approach with TF-IDF word weighting, which is a widely used natural language processing technique. Here, let the set of API words collected from all malware samples be $\mathbf{W} = \{w_1, w_2, \ldots, w_\Omega\}$. A feature vector of the i-th malware sample is denoted as

$$\mathbf{x}_i = \{\xi_i(w_1), \xi_i(w_2), ..., \xi_i(w_\Omega)\},$$

where $\xi_i(w_j) = \mathrm{tf}(i, w_j) \times \mathrm{idf}(\mathbf{M}, w_j)$ and $\mathbf{M}$ is a set of all malware samples. Here, tf and idf are defined as follows:

$$\begin{aligned} \mathrm{tf}(i, w_j) &= \frac{f(i, w_j)}{\sum_j f(i, w_j)} \\ \mathrm{idf}(\mathbf{M}, w_j) &= \log \frac{|\mathbf{M}|}{|\{m \in \mathbf{M} : w_j \in m\}|}, \end{aligned}$$

where $f(i, w_j)$ is the frequency of an API word w_j, in the i-th malware sample. Interested reader will refer to literatures such as [19].

B. Feature selection

Before we apply feature vectors to the clustering algorithm, we employ feature selection based on the frequencies of API words. The key idea is to eliminate API words that are unlikely to contribute to malware classification. Although the TF-IDF approach does work appropriately to some extent, we have determined empirically that feature selection improves the quality of clustering. We first apply DF-thresholding, which is widely used for text classification tasks [28]. Here, we

TABLE V. DEFINITIONS OF API WORDS AND THEIR EXAMPLES.

level	definitions	examples
Level 0	`api`	`"LdrGetDllHandle"`
Level 1	`api:n1:n2:...`	`"LdrGetDllHandle":"FileName":"ModuleHandle"`
Level 2	`api:n1:m(v1):n2:m(v2):...`	`"LdrGetDllHandle":"FileName":"C:\\WINDOWS\\system32\\rpcss.dll":"ModuleHandle":MASK`
Level 3	`api:n1:v1:n2:v2:...`	`"LdrGetDllHandle":"FileName":"C:\\WINDOWS\\system32\\rpcss.dll":"ModuleHandle":0x00000000`

eliminate very popular API words that appear in a majority of the malware samples. These API words are too generic to characterize a cluster of malware samples. Specifically, we introduce a threshold, $\theta (0 < \theta \leq 1)$. If the frequency of malware samples that consist of a given word is greater than θ, we eliminate the word. We empirically derived the optimum threshold ($\theta = 0.3$). We then eliminate API words with very low frequencies. Specifically, we eliminate the API words that appear less than three times throughout all malware samples. We note that the values of these parameters were not sensitive to the actual clustering, i.e., we may use different values such as 0.1 for θ and 5 for minimum API words count.

C. Overview of NMF

NMF is an algorithm that factorizes a matrix into two matrices with the constraint that all three matrices have no negative elements. The NMF algorithm and its extensions have been applied to a large number of applications, such as object recognition [17], text document clustering [26], and large-scale network diagnosis [16]. Because of this non-negative constraint, the results of NMF can be interpreted as a soft clustering of data [13]. The advantage of NMF is that it can extract topics that can then be used as signatures for our task. It should be noted that other algorithms, such as Latent Dirichlet allocation, can also perform topic modeling. However, we employ NMF because of its simplicity and intend to explore other algorithms in future work.

NMF factorizes the matrix, $\mathbf{X}$; i.e., $\mathbf{X} = \mathbf{TV}$. Here, $\mathbf{X}$ is defined using the feature vectors $\mathbf{x}_i$ as follows:

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T & \mathbf{x}_2^T & \cdots & \mathbf{x}_{|\mathbf{M}|}^T \end{pmatrix} = \begin{pmatrix} \xi_1(w_1) & \xi_2(w_1) & \cdots & \xi_{|\mathbf{M}|}(w_1) \\ \xi_1(w_2) & \xi_2(w_2) & \cdots & \xi_{|\mathbf{M}|}(w_2) \\ \vdots & \vdots & \ddots & \vdots \\ \xi_1(w_\Omega) & \xi_2(w_\Omega) & \cdots & \xi_{|\mathbf{M}|}(w_\Omega) \end{pmatrix},$$

and it is factorized as

$$\mathbf{X} = \mathbf{TV} = \begin{pmatrix} t_{11} & \cdots & t_{1K} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ t_{\Omega 1} & \cdots & t_{\Omega K} \end{pmatrix} \begin{pmatrix} v_{11} & \cdots & \cdots & v_{1|\mathbf{M}|} \\ \vdots & \vdots & \vdots & \vdots \\ v_{K1} & \cdots & \cdots & v_{K|\mathbf{M}|} \end{pmatrix}.$$

K is a parameter that controls the number of basis, which corresponds to clusters. The matrices $\mathbf{T}$ and $\mathbf{V}$ can be interpreted as the topic (basis) and the clustering results, respectively.

This factorization can be obtained by minimizing the distance between the matrices $\mathbf{X}$ and $\mathbf{TV}$,

$$D(\mathbf{X}, \mathbf{TV}) = \sum_i \sum_j d(x_{ij}, \mathbf{t}_i^T \mathbf{v}_j).$$

Of the several distance functions, d, we adopt the Kullback-Leibler (KL) divergence, which is defined as follows:

$$d(x_{ij}, \mathbf{t}_i^T \mathbf{v}_j) = x_{ij} \log \frac{x_{ij}}{\mathbf{t}_i^T \mathbf{v}_j} - x_{ij} + \mathbf{t}_i^T \mathbf{v}_j.$$

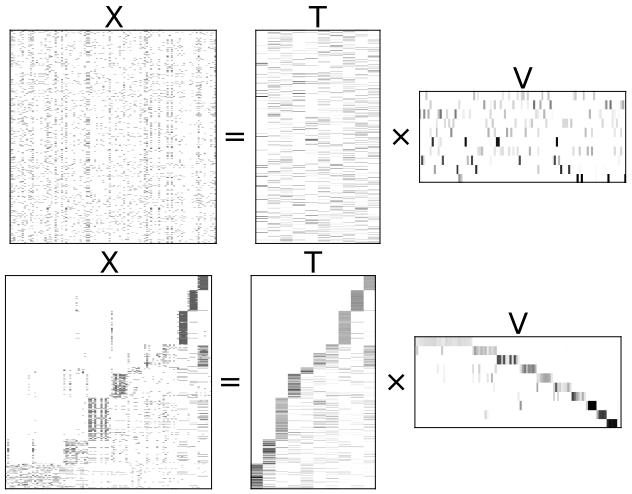

Fig. 3. Illustration of NMF outputs, $\mathbf{X} = \mathbf{TV}$. Top: original matrices. Bottom: sorted matrices.

The multiplicative update rule is a widely used method that minimizes $D(\mathbf{X}, \mathbf{TV})$. For KL divergence, the multiplicative update rule is calculated as follows [22]:

$$t_{ik} \leftarrow t_{ik} \frac{\sum_j v_{kj} \left(x_{ij} / \sum_k t_{ik} v_{kj} \right)}{\sum_j v_{kj}}$$
$$v_{kj} \leftarrow v_{kj} \frac{\sum_i t_{ik} \left(x_{ij} / \sum_k t_{ik} v_{kj} \right)}{\sum_i t_{ik}}.$$

Figure 3 shows an example of the NMF result. For illustrative purposes, the number of samples in this example was reduced to 100. The parameter K was set to $K = 10$. As is shown in Fig. 3, NMF successfully extracts latent structural patterns from the original malware sample-API words matrix, $\mathbf{X}$.

Given the NMF result, we can assign each malware sample to cluster(s). For this task, we introduce a threshold γ. If an element of $\mathbf{v}$ satisfies $v_{kj} > \gamma$, then we assign the j-th sample to the cluster k. We set γ as the quantile of the values of the elements v_{kj}. This value has been determined from the experimental observations. In Fig. 3, the gray/black squares represent clusters. It should be noted that a single malware sample can potentially belong to multiple clusters if it has multiple API topics.

Here we discuss the tuning of parameter, K. Figure 4 shows the relationship between K and the fraction of malware samples that were not assigned to clusters. While the fraction decreases as K increases, it is desirable that K be sufficiently small to be interpretable by a malware analyst. We have determined the value of K using the following rule:

$$\widehat{K} = \min(K; U(K) \leq 0.01).$$

This gave $\widehat{K} = 58$ for the data (**D1** + **D2**). $U(K)$ is the fraction of elements that were not assigned to any clusters.

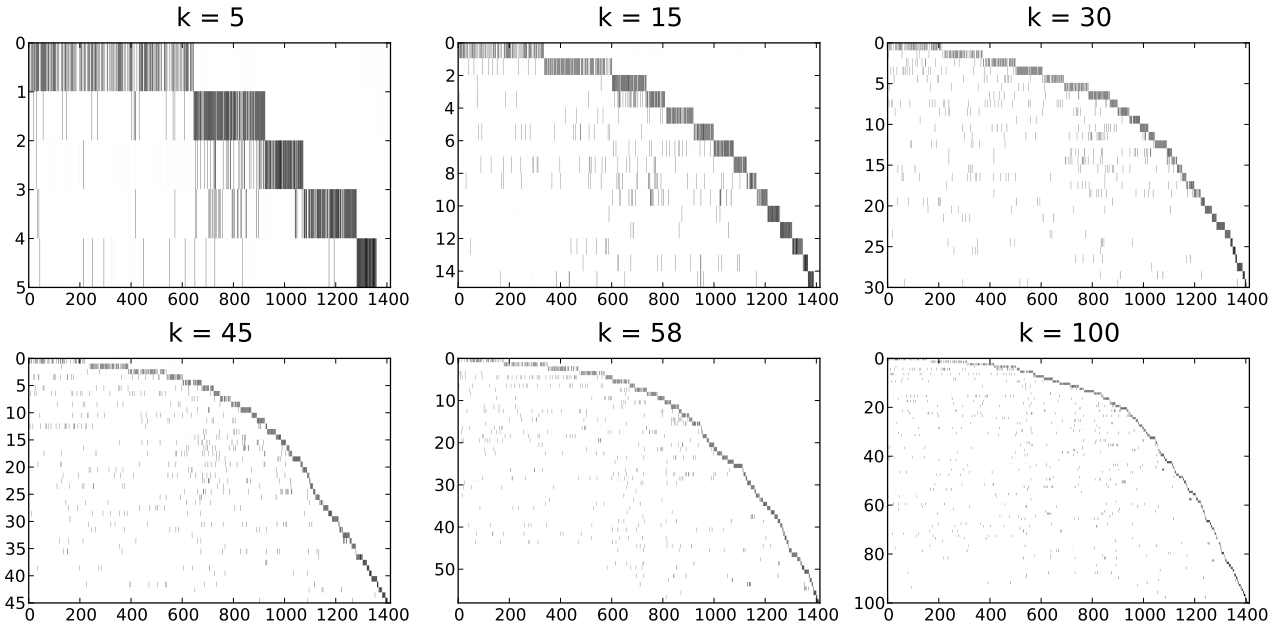

Fig. 5. Visualizing the matrices **V** (sorted). Note that elements are sorted according to the primary cluster IDs.

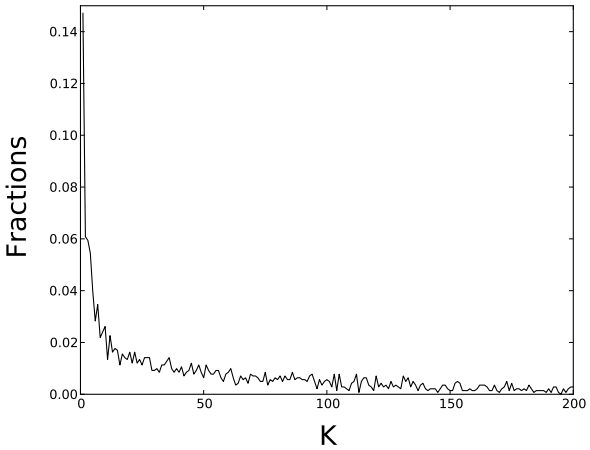

Fig. 4. K vs. fraction of unclassified malware samples.

Figure 5 visualizes the matrices **V** for various K values. As can be seen, $K = 5$ is too small to classify malware samples into clusters, and $K = 100$ seems too fine grained. Slight differences can be observed for other settings. In fact, the performance of a similar malware detection scheme is approximately equal among those settings. Our heuristic was able to employ an adequate range for parameter K.

IV. Analysis of API call topics.

In this section, we first show how we extracted the API call topics using the methodologies we developed. Next, we present how the extracted API call topics can be used to detect new malware samples that are similar to the known ones.

A. Extraction of API call topics

Here, we use the data **D1** to extract API call topics, and we use data **D1** and **D2** to evaluate similar malware detection using the extracted API topics. All of the samples in **D2** were collected after **D1**; therefore **D2** can be regarded as *unknown* future data.

From the matrix **T**, we can extract "topics" for each cluster (basis). In other words, for each cluster, k, we sort the API words t_{ik} and extract the highest S words. We call these S-API words "API topics." Table VIII summarizes the examples of API call topics for the four clusters. We present the top 10 API words for each API topic. We selected these four clusters because they consist of a large number of samples. We note that these are selected for illustration purpose and API call topics were extracted for other clusters as well. Also, it should be noted that we used only data **D1** for this task.

Each cluster consists of API words that are similar to the API topics. These API words are associated with the behavior of malware because the extracted APIs with specific arguments should constitute user-defined functions that characterize malware activities. For example, a majority of malware samples for cluster No. 25 were labeled as `Backdoor.Win32.Simda`, which is known to replace the original volume boot record of a hard disk drive with its own data. Such activities can also be seen from the API topics. Thus, the detected API topics can provide analysts investigating malware with some insight.

B. Detecting similar malware samples

For each malware sample i ($i = 1, 2, \ldots$), we compute $R_k(i)$, which is the number of API words from the API-topic of cluster k. To identify malware samples that are similar to those in cluster k, we define a threshold, η ($\leq S$). If $R_k(i) \geq \eta$, we consider that malware sample i is similar to those in cluster k.

TABLE VI. DETECTION RESULTS FOR *training* SAMPLES (**D1**).

Cluster	# samples	labels (%)
1	128	**Vobfus** (85.9), SelfDel (4.7), Diple (3.1), others (6.3)
15	30	**Zbot** (70.0), others (30.0)
25	20	**Simda** (80.0), Agent (10.0), others (10.0)
41	16	**Azbreg** (62.5), Lebag (12.5), others (25.0)

TABLE VII. DETECTION RESULTS FOR *test* SAMPLES (**D2**).

Cluster	# samples	labels (%)
1	94	**Vobfus** (87.2), SelfDel (4.3), Swisyn (2.1), others (6.4)
15	13	**Zbot** (69.2), Tepfer (15.4), others (15.4)
25	9	**Simda** (88.9), others (11.1)
41	6	**Azbreg** (66.7), others (33.3)

Tables VI and VII show the detected malware samples for data **D1** and **D2**. To demonstrate similarity, we use labels created by the Kaspersky antivirus checker. For both datasets, the API call topics successfully extracted malware samples that had the similar intrinsic API calls. In addition, the detected malware samples were actually similar to each other because the majority of the detected samples have a single primary label in each cluster. For example, in cluster No. 1, the majority of detected samples were labeled as `Trojan.Win32.Vobfus` for both **D1** and **D2**. It should be noted that an extracted cluster can include multiple labels created by an antivirus checker because the extracted set of API calls may constitute a function that is common among different malware, such as common libraries or routines shared among malware developers. As we mentioned in section II, labels created by an AV vendor are not perfect [21]. However, the labels enables us to have hints to objectively interpret the outputs.

Figure 6 shows the relationship between API call topics (clusters) and primary labels. We plot the primary labels that accounted for greater than 20% of the samples in each cluster. As can be seen, a cluster may be composed of malware samples from multiple labels. For example, several clusters have malware samples for both `Vobfus` and `WBNA`. We can also see that `VB`, `VBNA`, `Vobfus`, and `WBNA` tend to have the same API call topics. These four malware families are similar [20], which is why we can observe common API call topics among them. Our methodology reveals such relationships without using any a priori domain knowledge. Thus, by carefully examining these similarities across a cluster, we can discover latent relationships among malware families. We can see that malware samples labeled as `Vobfus` and `WBNA` have multiple distinct API call topics. Although not conclusive, we conjecture that a malware family may have multiple API call topics that reflect a set of particular functions used in malware. Further investigation of the relationships among API call topics and malware functions is planned for future work.

V. RELATED WORK

Several studies have analyzed API calls with the machine learning-based approaches [5]–[7], [18], [24]. They are broadly split into two categories: classification [5], [6], [24]. and clustering [7], [18].

Classification: Ahmed et al. [5] presented a malware detection scheme that leverages spatio-temporal information available in API calls. The key idea was to combine two different features,

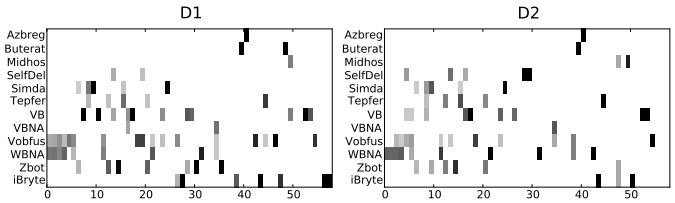

Fig. 6. Relationship between API call topics (clusters) and primary labels.

spatial features and temporal features, which are essentially input/return values and call sequences, respectively. Alazab et al. [6] proposed a method to classify Windows Executable files into malicious files or benign files. Their key idea was to apply the *n*-gram model to extract features from API calls, which were derived by IDA Pro Disassembler. They applied supervised machine learning (SVM) to solve the classification task. Similarly, Ravi et al. [24] presented a method that leverages supervised machine learning algorithm to classify Windows Executable files into malware of benign files. Again, the key idea was to apply the *n*-gram model to extract features from sequence of API calls, which were derived API emulator they developed. Among several classification algorithms, they adopted association rule mining that exhibited the best performance with their dataset.

Clustering: While the above studies used supervised machine learning approach, Refs. [7], [18] used unsupervised machine learning approach, i.e., clustering of malware samples. Bayer et al. [7] applied clustering algorithm to analyzing malware samples. They also developed a method to extract abstracted features from each sample; i.e., system calls, their dependences, and the network activities. Using the abstracted features, they demonstrated that the clustering results are accurate in that they are similar to the clusters that are built with antivirus checkers and manual inspection; i.e., they compared the ground-truth data with the clustered outputs. Li et al. [18] conducted empirical evaluation of malware clustering techniques by examining several algorithms and malware features, which include the one proposed in [7] and several variants that use API calls. Through the extensive analysis, the concluded that the knowledge on the ground-truth data such as cluster size distributions strongly biases the clustering results. We note that our approach did not rely on any ground-truth information to extract the API call topics.

While these studies aim to evaluate the accuracy of malware clustering, our work aims to automatically compile *signatures* that can be used to detect *unknown* malware samples by using the clustering approach. We also note that while the past studies adopted hard clustering algorithms, we adopted NMF, which is a soft clustering algorithm; i.e., an item (malware sample) can belong to multiple clusters (topics). We believe that allowing this flexibility is natural because a malware sample may have multiple features that are derived from different malware families.

VI. SUMMARY AND FUTURE WORK

We have proposed a new malware detection scheme that employs a corpus of API calls obtained by dynamic malware analysis systems. The proposed scheme extracts API calls

TABLE VIII. EXAMPLES OF API CALL TOPICS: TOP-10 API WORDS FOR EACH CLUSTER. BOLD FONTS INDICATE UNCOMMON (SUB)STRINGS.

Cluster 1:
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:**0x00d33000**:RegionSize:0x00001000:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:**0x00d34000**:RegionSize:0x00001000:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:**0x00d3c000**:RegionSize:0x00001000:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:**0x00d3b000**:RegionSize:0x00001000:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:**0x00d3d000**:RegionSize:0x00001000:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:0x00e10000:RegionSize:**0x00005000**:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:0x00e10000:RegionSize:**0x00004000**:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:0x00e10000:RegionSize:**0x00009000**:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:0x00e10000:RegionSize:**0x00007000**:Protection:0x00000004"
"NtAllocateVirtualMemory:ProcessHandle:0xffffffff:BaseAddress:0x00e10000:RegionSize:**0x00006000**:Protection:0x00000004"
Cluster 15:
"LdrGetProcedureAddress:ModuleHandle:**0x7c800000**:FunctionName:**GetComputerNameW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x77d80000**:FunctionName:**LookupPrivilegeValueW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x77cf0000**:FunctionName:**CharUpperW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x77cf0000**:FunctionName:**PeekMessageW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x77d80000**:FunctionName:**GetLengthSid**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x7c800000**:FunctionName:**CreateMutexW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x77cf0000**:FunctionName:**CharLowerW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x7c800000**:FunctionName:**CreateDirectoryW**:Ordinal:0"
"LdrGetProcedureAddress:ModuleHandle:**0x7c800000**:FunctionName:**OpenEventW:Ordinal**:0"
"LdrGetProcedureAddress:ModuleHandle:**0x77d80000**:FunctionName:**CreateProcessAsUserW**:Ordinal:0"
Cluster 25:
"**WSAStartup:VersionRequested:0x00000000**"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive15**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive3**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive2**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive10**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive8**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive11**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive1**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive7**:CreateDisposition:1:ShareAccess:3"
"NtCreateFile:FileHandle:0x00000000:DesiredAccess:0x00100080:FileName:**PhysicalDrive9**:CreateDisposition:1:ShareAccess:3"
Cluster 41:
"NtQueryValueKey:KeyHandle:0x00000084:ValueName:**SaferFlags**:Type:**4**:Information:**0**"
"NtQueryValueKey:KeyHandle:0x00000084:ValueName:**ItemSize**:Type:**11**:Information:"
"NtQueryValueKey:KeyHandle:0x00000084:ValueName:**HashAlg**:Type:**4**:Information:**32771**"
"NtQueryValueKey:KeyHandle:0x00000084:ValueName:**ItemData**:Type:**3**:Information:"
"NtOpenKey:KeyHandle:0x00000080:DesiredAccess:**1**:ObjectAttributes:Registry\\Machine\\Software\\Policies\\Microsoft\\Windows\\Safer\\CodeIdentifiers",
"NtOpenKey:KeyHandle:0x00000080:DesiredAccess:**131097**:ObjectAttributes:Registry\\Machine\\Software\\Policies\\Microsoft\\Windows\\Safer\\CodeIdentifiers"
"NtEnumerateKey:KeyHandle:0x00000080:Index:**0**",
"NtEnumerateKey:KeyHandle:0x00000080:Index:**1**",
"NtQueryValueKey:KeyHandle:**0x00000084**:ValueName:**ItemData:Type:2:Information:%HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Shell Folders\\Cache%OLK*\\\\x00**"
"NtQueryValueKey:KeyHandle:**0x00000080**:ValueName:**DefaultLevel:Type:4:Information:262144**"

topics by applying the NMF algorithm to the corpus of API calls. Through the analysis of thousands of malware samples, we have demonstrated that the proposed approach can successfully extract API call topics that can be used to detect similar malware samples. The result is useful for automating the creation of signatures that can be used to extract similar malware samples. One unexpected benefit of the proposed approach is that it can extract information to statically analyze the malware samples because the extracted API calls are intrinsic to such samples and may reflect the unique behaviors among the malware samples; e.g., use of intrinsic parameters or access to non-existing physical drives. The proposed approach is based on unsupervised learning and automatically extracts the API calls that are specific to a group of malware samples without relying on any domain knowledge. Such automation enables malware analysts to focus on new malware samples, thereby ignoring a large number of similar samples that have already been analyzed.

Our work leaves several research questions that need to be addressed in our future work. As we have shown, our approach groups multiple families into a single cluster. This characteristics is meaningful because it can capture similarities between them. If we simply rely on labels of malware samples, we cannot reveal such similarities. Given the clustering results with our approach, the important next step would be to further look into those similarities and identify where they originate from. In other words, is a cluster really capturing the malicious behaviors exhibited by the samples, or just some common API calls due to the use of a common library? Are those API calls also present in goodware? Are they related to the particular machine used to perform the analysis since they contain hard-coded addresses? etc. Addressing these questions will be useful to deepen understanding of malware families and their origins. In our future work, we plan to work on these questions. We also plan to extend the proposed topic analysis to other features, such as network communication, imported DLL files, dropped files, and printable strings.

ACKNOWLEDGEMENTS

We thank the contributors from the Anti-Malware Engineering WorkShop (MWS) for sharing the invaluable datasets with the research community. We also thank Tatsuaki Kimura for his useful comments regarding the theory and application of the NMF algorithm.

REFERENCES

[1] Anti-malware engineering workshop (mws) 2013. http://www.iwsec.org/mws/2013/ (in Japanese).

[2] Cuckoo sandbox. http://www.cuckoosandbox.org.

[3] Kaspersky. http://www.kaspersky.com.

[4] Virustotal. http://www.virustotal.com.

[5] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2Nd ACM Workshop on Security and Artificial Intelligence*, AISec '09, pages 55–62, New York, NY, USA, 2009. ACM.

[6] M. Alazab, S. Venkataraman, and P. Watters. Towards Understanding Malware Behaviour by the Extraction of API Calls. In *Prof. of Cybercrime and Trustworthy Computing Workshop (CTC)*, pages 52–59, 2010.

[7] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krgel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. of Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2009.

[8] J. Cannell. Automating malware analysis with cuckoo sandbox. http://blog.malwarebytes.org/intelligence/2014/04/automating-malware-analysis-with-cuckoo-sandbox/, 2014.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of ACM CCS*, pages 51–62, 2008.

[10] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, Mar. 2008.

[11] O. Ferrand. How to detect the cuckoo sandbox and to strengthen it? *Journal of Computer Virology and Hacking Techniques*, September 2014.

[12] M. Graziano, C. Leita, and D. Balzarotti. Towards network containment in malware analysis systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 339–348, New York, NY, USA, 2012. ACM.

[13] D. Greene, G. Cagney, N. J. Krogan, and P. Cunningham. Ensemble non-negative matrix factorization methods for clustering protein-protein interactions. *Bioinformatics*, 24(15):1722–1728, 2008.

[14] H. Huang, C. Lee, M. Wang, and H. Kao. It2fs-based ontology with soft-computing mechanism for malware behavior analysis. *Soft Comput.*, 18(2):267–284, 2014.

[15] Kaspersky Lab. Number of the year: Kaspersky Lab is detecting 315,000 new malicious files every day. http://www.kaspersky.com/about/news/virus/2013/number-of-the-year.

[16] T. Kimura, K. Ishibashi, T. Mori, H. Sawada, T. Toyono, K. Nishimatsu, A. Watanabe, A. Shimoda, and K. Shiomoto. Spatio-temporal Factorization of Log Data for Understanding Network Events. In *Proceedings of the IEEE INFOCOM 2014*, May 2014.

[17] D. D. Lee and H. S. Seung. Learning the parts of objects by nonnegative matrix factorization. *Nature*, 401:788–791, 1999.

[18] P. Li, L. Liu, D. Gao, and M. K. Reiter. On challenges in evaluating malware clustering. In *Proceedings of 13th International Symposium on Recent Advances in Intrusion Detection*, RAID 2010, pages 238–255, 2010.

[19] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[20] Microsoft Malware Protection Center. Win32/Vobfus.F. http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Worm:Win32/Vobfus.F#tab=2.

[21] A. Mohaisen and O. Alrawi. Av-meter: An evaluation of antivirus scans and labels. In *DIMVA*, pages 112–131, 2014.

[22] M. Nakano, H. Kameoka, J. L. Roux, Y. Kitano, N. Ono, and S. Sagayama. Convergence-guaranteed multiplicative algorithms for nonnegative matrix factorization with β-divergence. In *Proceedings of IEEE International Workshop on Machine Learning for Signal Processing*, MLSP 2010, pages 283–288, 2010.

[23] Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu. Cbm: Free, automatic malware analysis framework using api call sequences. In F. Sun, T. Li, and H. Li, editors, *Knowledge Engineering and Management*, volume 214 of *Advances in Intelligent Systems and Computing*, pages 225–236. Springer Berlin Heidelberg, 2014.

[24] C. Ravi and R. Manoharan. Malware detection using windows api sequence and machine learning. *International Journal of Computer Applications*, 43(17):12–16, April 2012. Published by Foundation of Computer Science, New York, USA.

[25] SECURELIST. Rules for naming detected objects. http://www.securelist.com/en/threats/detect?chapter=136.

[26] F. Shahnaz, M. W. Berry, V. P. Pauca, and R. J. Plemmons. Document clustering using nonnegative matrix factorization. *Inf. Process. Manage.*, 42(2):373–386, Mar. 2006.

[27] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2):32–39, 2007.

[28] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 412–420, 1997.