Lens on the endpoint: Hunting for malicious software through endpoint data analysis

Ahmet Salih Buyukkayhan[1], Alina Oprea[1], Zhou Li[2], and William Robertson[1]

[1] Northeastern University, Boston, MA, USA,
[2] RSA Laboratories, Bedford, MA, USA

Abstract. Organizations are facing an increasing number of criminal threats ranging from opportunistic malware to more advanced targeted attacks. While various security technologies are available to protect organizations' perimeters, still many breaches lead to undesired consequences such as loss of proprietary information, financial burden, and reputation defacing. Recently, endpoint monitoring agents that inspect system-level activities on user machines started to gain traction and be deployed in the industry as an additional defense layer. Their application, though, in most cases is only for forensic investigation to determine the root cause of an incident.

In this paper, we demonstrate how endpoint monitoring can be proactively used for detecting and prioritizing suspicious software modules overlooked by other defenses. Compared to other environments in which host-based detection proved successful, our setting of a large enterprise introduces unique challenges, including the heterogeneous environment (users installing software of their choice), limited ground truth (small number of malicious software available for training), and coarse-grained data collection (strict requirements are imposed on agents' performance overhead). Through applications of clustering and outlier detection algorithms, we develop techniques to identify modules with known malicious behavior, as well as modules impersonating popular benign applications. We leverage a large number of static, behavioral and contextual features in our algorithms, and new feature weighting methods that are resilient against missing attributes. The large majority of our findings are confirmed as malicious by anti-virus tools and manual investigation by experienced security analysts.

Keywords: Endpoint data analysis, Enterprise malware detection, Software impersonation, Security analytics, Outlier detection

1 Introduction

Malicious activities on the Internet are increasing at a staggering pace. The 2015 Verizon DBIR report [36] highlighted that in 2015 alone 70 million pieces of malware were observed across 10,000 organizations with a total estimated financial loss of 400 million dollars. Enterprises deploy firewalls, intrusion-detection systems, and other security technologies on premise to prevent breaches. However, most of these protections are only in effect within the organization perimeter. When users travel or work remotely, their devices lack the network-level protections offered within the organization and are subject to additional threats.

Recently, many organizations started to deploy endpoint monitoring agents [34] on user machines with the goal of protecting them even outside the enterprise perimeter. Mandiant [24] reports that in a set of 4 million surveyed hosts, 2.8 million hosts have endpoint instrumentation installed. These agents record various activities related to downloaded files, installed applications, running processes, active services, scheduled tasks, network connections, user authentication and other events of interest, and send the collected data to a centralized server for analysis. Since stringent requirements are imposed on the performance of these tools, they are usually lightweight and collect coarse-grained information. Today, this data is used mainly for forensic investigation, once an alert is triggered by other sources.

We believe that endpoint monitoring offers a huge opportunity for detection and mitigation of many malicious activities that escape current network-side defenses. Endpoint agents get visibility into different types of events such as registry changes and creation of executable files, which do not appear in network traffic. Moreover, existing research in host-based detection methods (e.g., [1, 12, 19, 2, 31, 27]) confirms our insight that endpoint monitoring can be used successfully for proactive breach detection. Nevertheless, to the best of our knowledge, endpoint monitoring technologies have not yet been used for this goal, as a number of challenges need to be overcome. Most accurate host-based detection technologies rely on much finer-grained data (e.g., system calls or process execution) than what is collected by endpoint agents. Additionally, production environments in large organizations need to handle up to hundreds of thousands of machines, with heterogeneous software configurations and millions of software variants. Ground truth is inherently limited in this setting, since we aim to detect malware that is already running on enterprise hosts, and as such has bypassed the security protections already deployed within the enterprise.

In this paper, we analyze endpoint data collected from a large, geographically distributed organization (including 36K Windows machines), and demonstrate how it can be used for detecting hundreds of suspicious modules (executables or DLLs) overlooked by other security controls. Our dataset includes a variety of attributes for 1.8 million distinct Windows modules installed on these machines. The enterprise of our study uses multiple tools to partially label the modules as *whitelisted* (signed by reputable vendors), *blacklisted* (confirmed malicious by manual investigation), *graylisted* (related to adware), or *unknown*. Interestingly, only 6.5% of modules are whitelisted, very small number (534) are blacklisted, while the large majority (above 90%) have unknown status. As the ground truth of malicious modules in our dataset in very limited, well-known techniques for malware detection such as supervised learning are ineffective.

We use several insights to make the application of machine learning successful in our setting. We first leverage the set of behaviors observed in blacklisted modules to identify other modules with similar characteristics. Towards that goal, we define a similarity distance metric on more than 50 static, behavioral and contextual features, and use a density-based clustering algorithm to detect new modules with suspicious behavior. Second, while enterprise hosts have relatively heterogeneous software configuration, it turns out that popular Windows executables or system processes have a large user base. We exploit the homogeneity of these whitelisted applications for detecting an emerging threat, that of *software impersonation* attacks [26]. We detect a class of attacks im-

personating static attributes of well-known files by a novel outlier-detection method. In both settings we use new dynamic feature weighting methods resilient to missing attributes and limited ground truth.

In summary, our contributions are highlighted below.

Endpoint-data analysis for malware detection. We are the first to analyze endpoint data collected from a realistic deployment within a large enterprise with the goal of proactively detecting suspicious modules on users' machines. We overcome challenges related to (1) lightweight instrumentation resulting in coarse-grained event capturing; (2) the heterogeneous environment; (3) limited ground truth; (4) missing attributes in the dataset.

Prioritization of suspicious modules. We propose a density clustering algorithm for prioritizing the most suspicious modules with similar behavior as the blacklisted modules. Our algorithm reaches a precision of 90.8% and recall of 86.7% (resulting in F1 score of 88.7%) relative to manually-labeled ground truth. Among a set of 388K modules with unknown status, we identified 327 executable and 637 DLL modules with anomalous behavior and the false positive rates are as low as 0.11% and 0.0284% respectively. Through manual investigation, we confirmed as malicious 94.2% of the top ranked 69 executables and 100% of the top 20 DLL modules. Among these, 69 malicious modules were new findings confirmed malicious by manual investigation, but not detected by VirusTotal.

Software impersonation. We propose an outlier-detection algorithm to identify malware impersonating popular software. Our algorithm detected 44 outlying modules in a set of 7K unknown modules with similar characteristics as popular whitelisted modules, with precision of 84.09%. Among them, 12 modules are our new findings considered malicious by manual investigation, but not detected by VirusTotal.

Novel feature weighting methods. To account for missing attributes and limited ground truth, we propose new feature weighting methods taking into account the data distribution. We compare them with other well-known feature weighting methods and demonstrate better accuracy across multiple metrics of interest.

2 Background and overview

In this section we first describe the problem definition, adversarial model, and challenges we encountered. We then give an overview of our system, provide details on the dataset we used for analysis, and mention ethical considerations.

2.1 Problem statement

Organizations deploy network-perimeter defenses such as firewalls, anti-virus software, and intrusion detection systems to protect machines within their network. To obtain better visibility into user activities and offer protection outside of enterprise perimeter, organizations started to deploy endpoint agents on user machines [34]. These agents monitor processes running on end hosts, binaries downloaded from the web, modifications to system configuration or registries through lightweight instrumentation, and report a variety of recorded events to a centralized server for analysis.

In the organization of our study, machines are instrumented with host agents that perform regular and on-demand scans, collect aggregate behavioral events, and send them to a centralized server. We address the problem of discovering highly risky and suspicious modules installed on Windows machines through analysis of this realistic, large-scale dataset. Specifically, we are looking for two common types of malicious behavior:

- Starting from a set of *blacklisted* modules vetted by security experts, we are interested in discovering other modules with similar characteristics. With the availability of malware building kits [7], attackers can easily generate slightly different malware variants to evade signature detection tools. We leverage the insight that malicious variants produced by these toolkits share significant similarity in their behavior and other characteristics.

- Starting from a set of *whitelisted* modules considered legitimate, we look for malicious files impersonating them. System process impersonation has been used by Advanced Persistent Threats (APT) campaigns for evasion [25, 26]. Detecting this in isolation is difficult, but here we exploit the homogeneity of whitelisted files in an enterprise setting. These files have a large user base and should have similar behavior across different machines they are installed on. Our main insight is that malicious files impersonating these popular modules are significantly different in their behavior and contextual attributes.

Adversarial model. We assume that endpoint machines are subject to compromise through various attack vectors. An infection could happen either inside the enterprise network or outside when users travel or take their machines home. In modern attacks there are multiple stages in the campaign lifecycle, e.g., a piece of malware is delivered through email followed by download of second-stage malware that initiates communication with its command-and-control center and updates its code [23]. We assume that before attackers have complete control of the machine, the endpoint agent is able to collect and upload information to the centralized server. Of course, we cannot make any assumptions about agents once a machine is completely subverted by attackers. However, our goal is to detect infection early, before it leads to more serious consequences such as data leakage or compromise of administrator credentials.

We assume that the server storing the endpoint data is protected within the enterprise perimeter. Breaches involving a compromise of monitoring tools or servers are much more serious and can be detected through additional defenses, but they are not our focus. Here we aim to detect and remediate endpoint compromise to prevent a number of more serious threats.

Challenges. A number of unique challenges arise in our setting. Our dataset is collected from a heterogeneous environment with 1.8 million distinct modules installed on 36K machines. Most users have administrative rights on their machines and can install software of their choice. Second, we have limited ground truth with less then 10% of modules labeled as whitelisted, blacklisted or graylisted and the majority having unknown status. Third, a number of attributes are missing due to machine reboots or disconnection from corporate network. Lastly, the monitoring agents collect lightweight information to minimize their overhead.

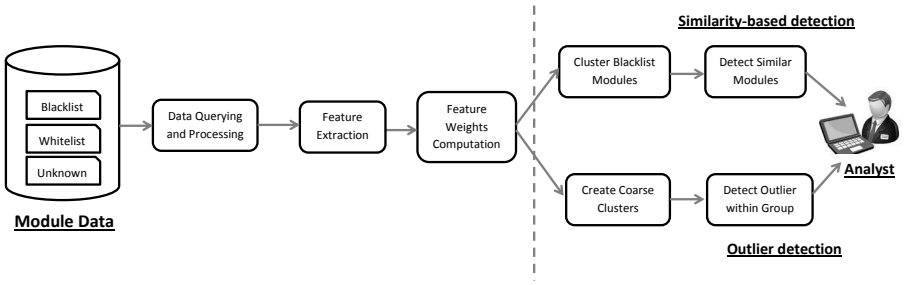

Fig. 1. System diagram.

2.2 System overview

Our system analyzes data collected from endpoint agents deployed in a large enterprise. Our goal is to identify among the large set of modules with unknown status those with suspicious behavior and prioritize them by their risk. In particular, we are looking for two types of malicious modules: (1) those with *similar behavior* as known blacklisted modules; and (2) those *impersonating* popular, legitimate whitelisted software. For our analysis, we employ a large number of features from three categories: *static* (extracted from the module's PE headers), *behavioral* (capturing file access patterns, process creation and network access events); and *contextual* (related to module location on the machines it is installed).

Our system architecture is illustrated in Figure 1. After we query the raw data from the server, we apply some data transformation and aggregation in the processing phase and extract features from these three categories. We define a module distance metric that assigns different feature weights for the two scenarios of interest. In case of similarity detection, high-entropy features are given higher weight and we adapt the DBSCAN algorithm to account for custom-defined distance metric and missing features. For software impersonation we favor features that distinguish malicious from benign files best, and design a novel two-stage outlier detection process. A detailed description of our techniques follows in Section 3.

2.3 Dataset

The dataset is collected by endpoint agents deployed on 36,872 Windows machines. Agents monitor executable and DLL modules, and perform scheduled scans at intervals of three days. Analysts could also request scans on demand. Data generated by agents is sent to a centralized server. We had access to a snapshot of the database from August 2015, including 1.8 million distinct modules. Among them, 117K were marked as *whitelisted* (through custom tools). A small set (534 modules) were labeled as *blacklisted* after detailed manual investigation by experienced security analysts. Note that we did not augment this set with results from anti-virus (AV) software, as these tools generate a large amount of alarms on low-risk modules, such as adware or spyware, which were considered "graylisted" by security analysts.

Status	#Total	#Description	#Company Name	#Signature
BL	534	440	445	520
WL	117,128	19,881	13,070	2,430
UL	1,692,157	1,304,557	1,314,780	1,503,449

Table 1. Total number of modules in each category (BL – blacklisted, WL – whitelisted, UL – unknown), and those with missing description, company name and signature fields.

We choose to only use the blacklisted modules as reference of highly risky malicious activity. The remaining 1.7 million modules have unknown status, including lesser-known applications and variants of known applications. In total, there are 301K distinct file names in our dataset.

To illustrate the noisy aspect of our dataset, Table 1 lists the total number of modules, as well as the number of modules without description, company name or signature in each category (BL – blacklisted, WL – whitelisted, UL – unknown). As seen in the table, the large majority of blacklisted modules do not include these fields, but also a fair number of unknown and whitelisted modules miss them.

To illustrate the heterogeneity of the environment, the left graph in Figure 3.3 shows the CDF for the number of hosts installing the same file name. The large majority of file names are installed on few hosts relative to the population. Even among whitelisted file names, 95% of them are installed on less than 100 hosts. 95% of the blacklisted files are installed on less than 20 hosts. Only a small percentage of files are extremely popular and these are mostly Windows executables and system processes or libraries (e.g., whitelisted `svchost.exe` and unknown `presentationcore.ni.dll` are installed on 36K and 29K machines, respectively).

The right graph in Figure 3.3 shows the CDF for the number of file variants with same name but distinct SHA256 hashes. Whitelisted and unknown file names include more variants than blacklisted ones. For instance, whitelisted `setup.exe` has 1300 variants, unknown `microsoft.visualstudio~.dll` has 26K variants, while the maximum number of blacklisted variants is 25. This is due to the limited set of blacklisted modules, as well as the evasive nature of malware changing file name in different variants.

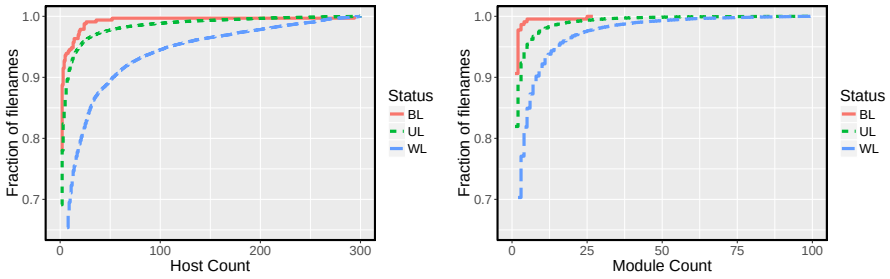

Fig. 2. CDFs of hosts (left) and modules (right) sharing same filename.

2.4 Ethical considerations

The enterprise's IT department consented to give us access to a snapshot of the data for the purpose of this study. We had access to data only within the premises of the enterprise and were only allowed to export the results presented in the paper. Our dataset did not include any personal identifying information (e.g., username and source IP of employee's machine) that put users' privacy at risk. We also took measures to prevent potential information leakage: for instance, the behavior and contextual features were aggregated across hosts installing the same module.

3 System design

We provide here details on our system design and implementation. Our first goal is prioritizing the most suspicious unknown modules with similar behavior as known blacklisted modules. Our second goal is detecting malware impersonating popular file names (e.g., system processes) through a novel outlier-detection algorithm. Both techniques can be used to detect suspicious unknown modules, and enlarge the set of blacklisted modules manually labeled by analysts. They both utilize the same set of 52 (*static*, *behavioral*, and *contextual*) features extracted from the dataset (see Section 3.1). However feature weights and parameters are customized for the two algorithms, as discussed in Sections 3.2 and 3.3.

3.1 Feature selection

For each module we extract a multi-dimensional feature vector, with features capturing the module's attributes according to three distinct categories: static, behavioral and contextual. Table 7 in Appendix A provides a comprehensive list of all features.

Static features. These are mainly extracted from the module's PE header and include: (1) *descriptive features* represented as either string values (description and company name) or sets (name of imported DLLs and section names); (2) *numerical features* such as file size, PE size, PE timestamp, module entropy; and (3) *binary features* denoting attributes such as signature present, signature valid, icon present, version information present, PE type (32 or 64 bit), PE machine (e.g., AMD64), and module packed.

Behavioral features. These are related to the module's behavior on all hosts where it is installed. We include features related to: (1) *file system access* – number of executable files created, deleted or renamed, files read, physical or logical drives opened; (2) *process access* – number of regular processes, browser or OS processes opened, processes or remote threads created; and (3) *network connections* such as set of domains and IP addresses the module connects to. These events are stored cumulatively at the server since the time the module was first observed on the network. Since a module might exist on many machines, we compute average number of events per machine for file system and process access features.

Contextual features. The endpoint agents collect information about the time when a module is initially installed on a machine, its full file system path, the user account that created the module and the full path of all files and processes captured by the behavior events initiated by the module. We parse the file path and match it to different categories such as Windows, Systems, ProgramFiles, ProgramData, or AppDataLocal. Additionally, the agents monitor if modules have auto-start functionality and categorizes that into different types (e.g., logon, services, startup, scheduled task). We also have access to the user category owning the module (admin, trusted installer or regular user).

From this information, we extract a number of contextual features related to: (1) *file system path* – number of directory levels, the path category, number of executable and non-executable files in the same folder, and number of sub-folders in the path; (2) *path of destination events* – the path category of destination files, and number of events created by the module in the same and in different paths; (3) *file's metadata* – file

owner, hidden attributes, and days from creation; (4) *auto-start functionality* – type of auto-start if enabled. We took the average values across all hosts installing the module.

Final set of features. We initially considered a larger set of 70 features, but we reduced the list to 52 features that are available in at least 10 blacklisted modules. Some features related to registry modifications, process and I/O activity were not encountered in our dataset of blacklisted modules, but could be applicable to an enlarged set of malicious modules. The final list of features we used is given in Table 7 in Appendix A.

3.2 Prioritizing suspicious modules

For detecting modules with similar behavior as known blacklisted modules, we first cluster the set of blacklisted modules, and then identify other unknown modules in these clusters. We prioritize unknown modules according to their distance to the blacklisted modules. We describe our definition of module similarity and distance metric, as well as our feature weighting method that is resilient against missing features.

Clustering. Many clustering algorithms are available in the literature, and we choose the DBSCAN [9] algorithm for clustering the blacklisted modules on the set of 52 features. Its advantages are that it does not require the number of clusters be specified in advance, can find arbitrarily-shaped clusters, and can scale to large datasets. DBSCAN creates clusters starting from *core samples*, points that have at least min_sample points in their neighborhood, and proceeds iteratively by expanding the clusters with points within distance ϵ (called *neighborhood radius*).

We use standard distance metrics for each feature, according to the feature's type: L1 distance for integer and real values; binary distance for binary values ($d(x, y) = 0$ if $x = y$, and $d(x, y) = 1$, otherwise); edit distance for strings; Jaccard distance for sets. The distance between two modules $M_1 = (x_1, \ldots, x_n)$ and $M_2 = (y_1, \ldots, y_n)$ is a weighted sum of distances for individual features: $d(M_1, M_2) = \sum_{i=1}^{n} w_i d(x_i, y_i)$, where $\sum_{i=1}^{n} w_i = 1$ [14].

Feature weights. One of our main observation is that features should contribute differently to overall modules similarity. While there are many established methods for feature selection and weighting in supervised settings [8, 15], the problem is less studied in unsupervised settings like ours.

We tested two methods for setting feature weights. Assume that we have n features in our dataset $X = (X_1, \ldots, X_n)$. First, a simple method is to set weights uniformly across all features, $w_i = 1/n$, for $i \in [1, n]$. In the second novel method we introduce, we choose feature weights *proportional to the feature's entropy* computed from the dataset. If feature i is categorical and has m possible values $v_1, \ldots, v_m$, we define p_{ij} as the probability that feature i takes value v_j, for $j \in [1, m]$. If feature i is numerical, we need to define a number m of bins $b_1, \ldots, b_m$ so that the probability of feature i belonging to bin b_j is p_{ij}, for $j \in [1, m]$. Then, the entropy for feature i is $H(X_i) = -\sum_{j=1}^{m} p_{ij} \log(p_{ij})$. We assign normalized feature weights proportional to their entropy, according to our intuition that *features with higher variability should contribute more towards module similarity*.

Our algorithms need to be resilient against missing features since a large fraction of behavior features are not available (as machines are offline for extended periods of

time, or machines are sometimes rebooted before sending behavior events to the server). When computing the distance between two missing values, rather than setting it at 0 we choose a fixed, *penalty value* which is a parameter of our algorithm (the distance between a missing value and any other existing value is set at the maximum value of 1). Higher penalty results in lower similarity when computing the distance metric, thus the value of the penalty needs to be carefully calibrated. We elaborate more on optimal parameter selection in Section 4.

Prioritizing unknown modules. After clustering blacklisted modules with DBSCAN and the distance metric described above, our next goal is to identify unknown modules that belong to these clusters. The algorithm is run on 388K unknown modules and assigns some of them to blacklisted clusters according to their distance to cluster points. To prioritize the most suspicious ones, we order the unknown modules that belong to a blacklisted cluster based on their minimum distance to known blacklisted modules. We describe our results in Section 4.1.

3.3 Impersonation of popular software

For detecting malware impersonating popular, legitimate software, we leverage the large machine base in our dataset to determine a set of popular modules and their common characteristics across machines. While it is relatively easy for malware to inherit some of the static features of popular modules to appear legitimate, in order to implement its functionality malware will exhibit differences in its behavioral and contextual features. We leverage this observation to detect a set of modules impersonating popular file names (e.g., system processes or software installers).

Our algorithm proceeds in two steps. First, we generate a set of "coarse" clusters whose large majority of modules are popular whitelisted files. Second, we identify a set of outliers in these clusters whose distance to other whitelisted modules is larger than the typical distance between legitimate modules in the cluster. The list of detected outliers is prioritized by the largest distance from legitimate ones. We elaborate on weight selection, distance computation, and our outlier detection algorithm below.

Weights and distance computation. As described in Section 3.2, the distance between modules is a sum of feature distances adjusted by weights. However, feature weights are computed differently in this case since we would like to give higher weights to features distinguishing benign and malicious modules. Towards this goal, we compute the information gain of the whole set of features over all whitelisted and blacklisted modules and define *static weights* proportional to the feature's information gain.

Assume that $X = (X_1, \ldots, X_n, y)$ is our dataset with n features and label y (blacklisted or whitelisted). Assume that feature i takes m values $v_1, \ldots, v_m$ and let S_{ij} be the set of records having $X_i = v_j$. The information gain for feature i in dataset X is:

$$IG(X, X_i) = H(X) - \sum_{j \in \{1, \cdots, m\}} \frac{|S_{ij}|}{|X|} H(S_{ij})$$

Here the entropy values $H(X)$ and $H(S_{ij})$ are computed from two bins (malicious and benign). We further refine our method to increase the weights of features with relative stability within the set of whitelisted modules in a cluster. In particular, we

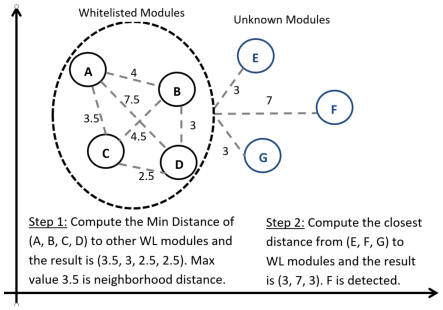

Fig. 3. Outlier detection example.

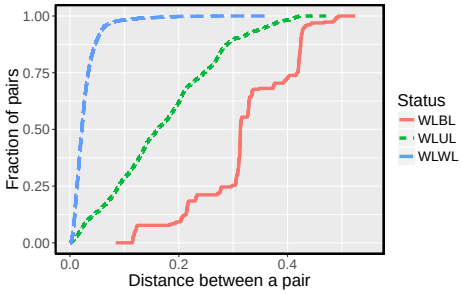

Fig. 4. Distance CDF from whitelisted to whitelisted (WLWL), unknown (WLUL) and blacklisted (WLBL) modules.

compute the average distance for feature i for all pairs of whitelisted modules (denoted Avg_i) per cluster and use $1/Avg_i$ as a factor proportional to feature i's stability. We set $Min(1/Avg_i, Max_W)$ as *dynamic weights* (Max_W is a threshold that limits the maximum weight – set at 20). The final feature weights for a cluster are defined as the product of static (global) and dynamic (cluster-specific) weights and normalized to sum up to 1. For missing values, we use a penalty value as in Section 3.2.

Coarse cluster selection. We create clusters of modules with popular file names. We select file names present on a large number of machines (more than a parameter O_γ). We enforce that our coarse clusters include sufficient benign samples through two conditions: (1) the clusters include minimum O_α whitelisted modules; and (2) the ratio of whitelisted modules to all modules in a cluster is at least a threshold O_β. Coarse clusters should also include at least one unknown (or blacklisted) module for being considered.

To account for generic file names (e.g., `setup.exe` or `update. exe`) with variable behavior, we compute the average distance of all pairs of whitelisted modules in a cluster (denoted Avg_{wdist}) and remove the clusters with Avg_{wdist} larger than a threshold O_θ. We also remove the modules developed by the company providing us the dataset, as most of the internal builds exhibit diverse behavior.

Detecting outliers. Figure 4 shows distance CDFs between whitelisted modules, as well as between whitelisted and blacklisted, and whitelisted and unknown modules in the coarse clusters. This confirms that blacklisted modules impersonating legitimate file names are at a larger distance from other whitelisted modules compared to the typical distance between legitimate modules. Based on this insight, our goal is to identify unknown modules substantially different from whitelisted ones in the coarse clusters.

Our approach involves measuring the *neighborhood distance* in a coarse cluster. For each whitelisted module, we compute the minimum distance to other whitelisted files, and the neighborhood distance (denoted $Dist_{WL}$) is the maximum of all the minimum distances. For an unknown module U the distance to the closest whitelisted module is $Dist_U$. Module U is considered an outlier if the ratio $R = \dfrac{Dist_U}{Dist_{WL}} > O_\lambda$. We illustrate this process in Figure 3.3. We experiment with different values of $O_\lambda \geq 1$ (see our results in Section 4.2).

4 Evaluation

We evaluated the effectiveness of our system using a snapshot of data from August 2015. Our dataset includes information about 534 blacklisted, 117K whitelisted and 1.7 million unknown modules installed on 36K Windows machines.

For prioritizing modules with known malicious behavior, we use 367 blacklisted modules whose static features have been correctly extracted. These modules were labeled by security experts with the corresponding malware family and we use them as ground truth to evaluate our clustering-based algorithm. Next, we selected a set of 388K unknown modules (79K executable and 309K DLL) installed on at most 100 machines (popular modules have lower chance of being malicious) and identified those that belong to the clusters generated by our algorithm. For validating the new findings, we used external intelligence (VirusTotal), internal AV scan results, as well as manual investigation by tier 3 security analysts. The results are presented in Section 4.1.

For validating our software impersonation detection algorithm, we used two datasets. First, we extracted all coarse-clusters with at least one whitelisted and one blacklisted module, and tested the effectiveness in identifying the blacklisted modules. This dataset (referred as `DS-Outlier-Black`) contains 15 clusters and 2K whitelisted, 19 blacklisted, and 2K unknown modules. Second, for higher coverage, we extracted all popular coarse-clusters (file names installed on more than 10K machines) that had at least one whitelisted and one unknown module. This dataset (`DS-Outlier-Unknown`) contains 314 clusters and a total of 11K whitelisted, 14 blacklisted, and 5K unknown modules. Unknown modules at large minimum distance from other whitelisted modules in these clusters were detected as outliers. The results are presented in Section 4.2.

Finally, both approaches are able to detect malicious modules ahead of off-the-shelf anti-virus tools. Initally only 25 out of 327 unknown executables and 463 out of 637 unknown DLLs were flagged by VirusTotal but eight months later (in May 2016), we uploaded the hashes of detected modules to VirusTotal again and noticed that 2 executables and 23 DLLs were detected in addition to previous findings (from August 2015). We identified a total of 81 modules (69 by clustering and 12 by outlier detection) confirmed malicious through manual investigation, but still not flagged by VirusTotal.

4.1 Results on prioritizing malicious modules

Results on blacklisted modules We use the 367 blacklisted modules as ground truth to select optimal values of the *penalty* and ϵ parameter in DBSCAN (we set $min\_sample = 2$ since we observed clusters with 2 malware samples). Our goal is to optimize a metric called *F1 score* that is a weighted average of precision and recall, but we also consider other metrics (precision, recall, false positives, false negatives). In our ground truth dataset, 147 modules are labeled as noise (they do not belong to any cluster). To account for these, we measure *coverage*, defined as the percentage of blacklisted modules (excluding the ones in the noise set) that belong to a cluster of size at least min_sample.

Number of modules	367 Blacklisted (273 EXE, 94 DLL)
Features	Static only, All Features
Feature weights	Uniform, Entropy-based
Missing features	$penalty \in [0.1, 0.8]$
DBSCAN Parameters	$min\_sample = 2$
	$\epsilon \in [0.05, 0.3]$

Table 2. Parameters in DBSCAN clustering.

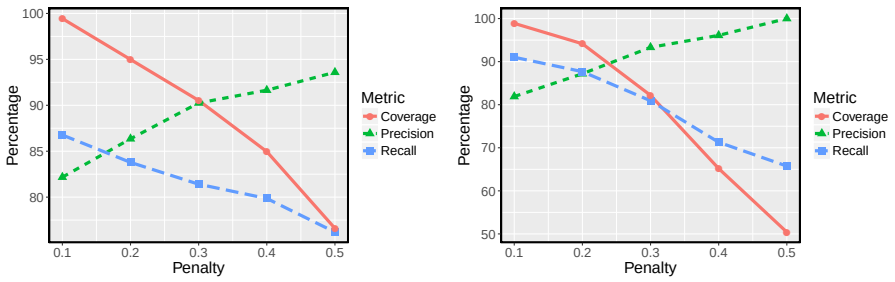

Fig. 5. Penalty dependence for Static-Unif with $\epsilon = 0.1$ (left) and All-Ent with $\epsilon = 0.2$ (right).

We experiment with different parameters in DBSCAN, as detailed in Table 2. We vary ϵ in DBSCAN between 0.05 and 0.3 and the penalty of missing features in the [0.1,0.8] range at intervals of 0.01. We consider and compare four models: (1) Static-Unif: static features with uniform weights; (2) Static-Ent: static features with entropy weights; (3) All-Unif: all features with uniform weights; (4) All-Ent: all features with entropy weights. Most of the features with highest entropy are static features but some context (time since creation, path-related features) and behavior features (set of contacted IP addresses and created processes) are also highly ranked. We used bins of 7 days for *PE timestamp* and *Days since creation*, and bins of 64KB for *File Size* and *PE Size*.

Model	Penalty	ϵ	Clusters	Single clusters	FP	FN	Precision	Recall	Coverage	F1
Static-Unif	0.3	0.13	50	150	55	42	84.67	87.86	99.16	86.24
Static-Ent	0.3	0.15	59	173	34	67	90.52	82.9	92.75	86.55
All-Unif	0.2	0.17	37	215	28	89	92.2	78.8	81.05	84.98
All-Ent	0.1	0.16	49	172	33	50	90.8	86.7	93.03	88.7

Table 3. Optimal performance metrics for 4 models.

Penalty choice. We first fix the value of ϵ and show various tradeoffs in our metrics depending on *penalty* (the distance between a missing feature and any other feature value). Figure 5 (left) shows the dependence on *penalty* for three different metrics (precision, recall and coverage) for the Static-Unif model when ϵ is set at 0.1. As we increase the *penalty*, the distance between dissimilar modules increases and the coverage decreases as more modules are classified as noise. Also, smaller clusters are created and the overall number of clusters increases, resulting in higher precision and lower recall. In Figure 5 the increase in precision is faster than the decrease in recall until *penalty* reaches 0.3, which gives the optimal F1 score for the Static-Unif model.

As we include more features in our models (in the All-Unif and All-Ent models), the *penalty* contribution should be lower as it intuitively should be inversely proportional to the space dimension (particularly as a large number of behavior features are missing). Figure 5 (right) shows how *penalty* choice affects our metrics in the All-Ent model for ϵ fixed at 0.2. Similar trends as in Static-Unif are observed, but a *penalty* of 0.1 achieves optimal F1 score. In both cases, results are consistent for different values of ϵ.

Choice of ϵ. For optimal penalty values as described above, the graph in Figure 6 shows the F1 score as a function of the neighborhood size in DBSCAN (ϵ) for the four models

considered. The optimal ϵ value is slightly larger in models with all features (0.16 for All-Unif and 0.17 for All-Ent) compared to models using static features only (0.13 for Static-Unif and 0.15 for Static-Ent). When more features are used, naturally the value of the neighborhood size in a cluster needs to be enlarged to account for larger distances between modules and more noise in the feature vectors.

Model comparison. Table 3 gives all metrics of interest for the four models with choice of ϵ and penalty parameters achieving optimal F1 score. Several observations based on Table 3 and Figure 6 are described below:

- *Feature weights make a difference.* Choosing feature weights proportional to the feature's entropy in the blacklisted set improves our metrics compared to choosing weights uniformly. For static models, precision is increased from 84.97% for uniform weights to 90.52% for entropy-based weights. For models considering all features, the recall is improved from 78.8% for uniform weights to 86.7% for entropy weights. The overall F1 score for All-Ent is maximum at 88.7% (with precision of 90.8% and recall of 86.7%) compared to Static-Unif at 86.24% and All-Unif at 84.98%.

- *Benefit of behavioral and contextual features.* Augmenting the feature list with behavioral and contextual features has the effect of increasing the F1 score from 86.55% (in Static-Ent) to 88.7% (in All-Ent). While precision is relatively the same in Static-Ent and All-Ent, the recall increases from 82.9% in Static-Ent to 86.7% in All-Ent. An additional benefit of using behavioral and contextual features (which we can not though quantify in our dataset) is the increased resilience to malware evasion of the static feature list.

- *Coverage and noise calibration.* The coverage for the optimal All-Ent model is relatively high at 93.03%, but interestingly the maximum coverage of 99.16% was achieved by the Static-Unif model (most likely due to the smaller dimension of the feature space). The model All-Unif performs worse in terms of noise (as 215 single clusters are generated) and coverage (at 81.05%). This shows the need for feature weight adjustment particularly in settings of larger dimensions when missing features are common.

Results on unknown modules. We empirically created the blacklisted clusters with All-Ent for optimal parameters $\epsilon = 0.16$ and *penalty*$= 0.1$. We now compare the list of 388K unknown modules to all blacklisted modules. As an optimization, we first compute the distance between blacklisted and unknown modules using only static features and filter out the ones with distance larger than ϵ, leaving 1741 executables and 2391 DLLs. Then, we compute the distance between the remaining unknown and blacklisted modules using all features. If an unknown module is within the distance threshold ϵ to one blacklisted module, we consider it similar but continue to find the closest blacklisted module. The detected modules are prioritized based on their minimum distance to a blacklisted module. In the end, 327 executables and 637 DLLs were detected.

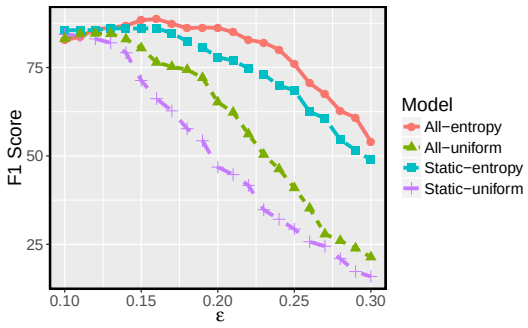

Fig. 6. F1 score as a function of ϵ for four models.

For verification, we uploaded the hashes of these modules to VirusTotal in August 2015 and 25 out of 327 unknown executables and 463 out of 637 unknown DLLs were flagged by at least one anti-virus engine. The reason for such low match on executable files is that most of them were not available in VirusTotal and company policies did not allow us to submit binary files to VirusTotal. When combining VirusTotal with the results from internal AV scan, we identified 239 out of 327 unknown executable and 549 out of 637 DLLs as suspicious, corresponding to a precision of 73% and 86%, respectively. Among the set of 79K executable and 309K DLLs, there were 88 executable and 88 DLL legitimate modules detected by our algorithm, corresponding to a false positive rate of 0.11% and 0.0284%, respectively.

To further confirm our findings, we selected a number of 89 modules with highest score (69 executables and 20 DLLs) and validated them with the help of a tier 3 security analyst. The analyst confirmed 65 out of 69 executables and all 20 DLL modules as malicious, resulting in a precision of 94.2% on executables and 100% on DLLs. Another interesting finding is that our techniques detected new malicious modules confirmed by the security analyst, but not flagged by VirusTotal. In total 60 executables and 9 DLLs from the set of 89 investigated modules were confirmed malicious by the security analyst, but not detected by VirusTotal. These new findings demonstrate the ability of our techniques to complement existing anti-virus detection technologies, and add another protection layer on endpoints.

4.2 Results of Outlier Detection

We evaluated the effectiveness of our approach in detecting software impersonation on two separate datasets (`DS-Outlier-Black` and `DS-Outlier-Unknown`). Before describing the results, we discuss how the parameters of the algorithm are selected.

Parameter selection. In the coarse cluster selection stage, we select popular file names by comparing the number of module installations to O_γ. We set O_γ to 10K, representing 25% of our set of monitored machines. This setting captures popular software (e.g., system processes, common browsers, Java). To ensure that the coarse clusters include enough benign samples for learning legitimate behavior, we use O_α and O_β as the lower-bounds for the number and ratio of whitelisted modules. We set $O_\alpha = 5, O_\beta = 0.2$ in `DS-Outlier-Black` for larger coverage and $O_\alpha = 10, O_\beta = 0.1$ in `DS-Outlier-Unknown`. As illustrated in Figure 4, the pairwise distances between whitelisted modules are usually small (below 0.05 for $\geq$ 95% pairs), while distances from whitelisted to unknown and blacklisted modules are much larger. Hence, we only include stable clusters whose Avg_{wdist} is smaller than the threshold O_θ set at 0.05.

Results on `DS-Outlier-Black`. We examined the 15 clusters in `DS-Outlier-Black` (including at least one blacklisted module) and inspected the 19 blacklisted and 2K unknown modules in these clusters. We found most filenames targeted by malware being Windows system files, such as `svchost.exe`, `lsass.exe`, `dwm.exe`, `services.exe` and `explorer.exe`. Malware impersonates these files to avoid causing suspicion as these processes are always present in Windows Task Manager. Additionally, file names belonging to popular software, including `wmplayer.exe` (Windows Media Player), `reader_sl .exe` (Adobe Acrobat SpeedLauncher) and `GoogleUpdate.exe` (Google Installer), are also targets for impersonation.

Dataset	#FileName	#Blacklisted	#Malicious	#Suspicious	#Unknown	#Modules	Precision%
DS-Outlier-Black	5	12	1	7	0	20	100
DS-Outlier-Unknown	10	0	5	12	7	24	70.8

Table 4. Summary of modules detected as outliers.

After coarse cluster selection, we obtained 5 clusters that met our selection criteria. These include 12 blacklisted and 12 unknown modules. We first evaluate the coverage of our algorithm in detecting blacklisted modules. To this end, our outlier detection algorithm captures all 12 blacklisted modules in these 5 clusters, as their distance from whitelisted modules is above 4, much larger than the threshold O_λ set at 1 (see Section 3.3). Among the 12 unknown modules, 8 modules in 4 clusters are alarmed and are all confirmed to be either malicious (flagged by VirusTotal) or suspicious (experiences unusual behavior, but is not yet confirmed as malicious by domain experts). In particular, a malicious module impersonating `services.exe` is detected one week ahead of VirusTotal, but other instances of this file are also suspicious (one of them is the `ZeroAccess` rootkit [26]). The summary of our results is in Table 4.

Results on `DS-Outlier-Unknown`**.** We use the data from `DS-Outlier-Unknown` to evaluate our approach on a larger set of clusters including at least one unknown module, but not necessarily any blacklisted modules. `DS-Outlier-Unknown` includes 314 clusters with 5K unknown modules, and we show that our approach can still achieve high precision in this larger dataset.

After applying our filtering steps, 14 clusters (with 30 unknown and no blacklisted modules) were handed to the outlier detection algorithm. New system processes (e.g., `mpcmdrun.exe`) and new applications (e.g., `installflashplayer.exe`) were identified in this dataset. Among the 30 unknown modules, 24 were flagged as outliers based on their distance to the closest whitelisted module. Among them, 17 were confirmed malicious, but only 5 were detected by VirusTotal. Thus, our outlier detection technique identified 12 modules not detected by VirusTotal as malicious.We did not find enough information to validate the remaining 7 modules and we labeled them as unknown. By considering the malicious and suspicious instances as true positives, the overall precision is 70.8%. In total, 44 modules were detected (combining the results on `DS-Outlier-Black`) with an overall precision of 84.09%. We summarize our findings in Table 4, provide more details on the detected modules in Table 6, and present a case study in Appendix B.

Dataset	Count	O_λ 1	4	7	10
DS-Outlier-Black	Confirmed	20	18	8	4
	Unknown	0	0	0	0
DS-Outlier-Unknown	Confirmed	17	13	5	4
	Unknown	7	4	3	2

Table 5. Detection results based on different O_λ.

We also assess the impact of the threshold O_λ on the result. We increase O_λ incrementally from 1 to 10 and measure the number of confirmed (malicious and suspicious) and unknown modules for both datasets. The results shown in Table 5 suggest that setting O_λ to 1 achieves both high accuracy and good coverage.

5 Limitations

An adversary with knowledge of the set of features employed by our algorithms might attempt to evade our detection. Most static features (e.g., description, size) can be modified easily. Even if the attacker is successful in evading a subset of static features, our

Dataset	FileName	#Blacklisted	#Malicious	#Suspicious	#Unknown	Anomalous features
DS-Outlier-Black	services.exe	2	1	2	0	Unsigned, path, DLLs
	svchost.exe	4	0	0	0	Unsigned, path, DLLs, size, description, company name, Auto_Logon, hidden attribute
	googleupdate.exe	1	0	1	0	Invalid signature, DLLs, newly created, ssdeep similar
	dwm.exe	4	0	1	0	Unsigned, path, DLLs
	wmplayer.exe	1	0	3	0	Unsigned, description, DLLs, ssdeep similar to malware
DS-Outlier-Unknown	udaterui.exe	0	0	0	1	Invalid signature
	googleupdatesetup.exe	0	0	3	0	Unsigned, path, version info, similar to malicious by ssdeep
	installflashplayer.exe	0	5	5	0	5 Confirmed by VirusTotal, similar to malicious by ssdeep
	intelcphecisvc.exe	0	0	1	0	Unsigned, size, entropy, ssdeep similar to malware
	mpcmdrun.exe	0	0	1	0	Unsigned, size, network connections, ssdeep similar to malware
	pwmewsvc.exe	0	0	0	1	Unsigned, no version info, size, compile time
	tphkload.exe	0	0	2	0	Invalid signature, size, compile time, creates remote thread
	flashplayerupdateservice.exe	0	0	0	3	Invalid signature
	vpnagent.exe	0	0	0	1	Invalid signature
	vstskmgr.exe	0	0	0	1	Invalid signature

Table 6. Summary of the modules alarmed by outlier detection algorithm.

dynamic feature weighting method still provides resilience against this attack. Since feature weights are adaptively adjusted in our case, other features (behavior and contextual) get higher weights, and static features become less significant.

To evade the behavior and contextual features, malware authors need to adjust multiple functionalities like processes creation, file access and communications which could incur high cost in the malware development process. For example, we consider abnormal remote IPs as one behavior feature and evading this requires changes to the attacker's or target's network infrastructure. At the same time, most contextual features (e.g., file path, number of executables in the same folder, auto-start functionality) are dependent on the organization's configuration, typically not known by attackers.

Another concern is that behavior-based techniques could be vulnerable to mimicry attacks [5], in which malware simulates system call sequences of legitimate software to avoid detection. We argue that mimicry attacks are less likely to succeed in our setting as we collect a more diverse set of behavioral and contextual features.

Advanced attackers could suppress events generated by the monitors or even inject fake events for evasion. Approaches that protect the agent integrity, like PillarBox [4], could be deployed to defeat against these attacks.

6 Related Work

Malware clustering. To automatically detect malware variants and reduce the security analysts' workload, malware clustering techniques (e.g., [1, 37, 2, 31, 29, 18, 17, 27]) were proposed by the security community. These techniques perform static and dynamic analysis by running known malware samples in controlled environments. They extract fine-grained features related to file system access, registry modification, OS activities, and network connections. Our work differs from these approaches in the following aspects. First, our features are extracted from data collected by agents installed on a large set of user machines in an enterprise network. Second, we only have access to coarse-grained aggregated behavioral events as stringent performance constraints are imposed on the agents. Moreover, our ground truth is limited with the large majority of modules (more than 90%) having unknown status. Lastly, we introduce a new set of contextual features (e.g., location of files on user machines, file metadata, auto-start functionality) that leverage the large, homogeneous user base in enterprise settings.

Host-based anomaly detection. Many previous works proposed algorithms for detection of unusual program behavior based on runtime information collected from hosts. So far, system calls [32, 11, 16, 22, 21], return addresses from call stack [10], system state changes [1], memory dumps [3], and access activities on files and registries [20] have been used to detect suspicious behavior. We used a more comprehensive set of features, extracted from a much larger realistic deployment.

Recently, researchers proposed malware detection systems based on data collected from a large number of endpoints (e.g., Polonimum [6], AESOP [35], MASTINO [30]). These approaches rely on file-to-machine and file-to-file affinities, and cannot detect isolated infections. In contrast, our approach is exempted from such restrictions. Gu et. al. [13] developed a detection system against camouflaged attacks (malicious code injected in legitimate applications at runtime). Our system covers camouflage attacks as part of software impersonation, but addresses a larger set of attacks. A recent trend in this area is to combine network and host-based behavioral features for anomaly detection [40, 33].

Enterprise security analytics. Previous research showed that security logs collected in a large enterprise, such as web proxy, Windows authentication, VPN, and DHCP, can be leveraged to detect host outliers [39], predict host infection [38], and detect malicious communications in multi-stage campaigns initiated by advanced persistent threats [28]. We focus here on analyzing a different source of data (collected by monitoring agents deployed on Windows machines) with the goal of identifying suspicious modules installed on user machines. We believe that combining endpoint and network-based monitoring data is most promising for identifying increasingly sophisticated threats in the future.

7 Conclusions

In this paper, we present the first study analyzing endpoint data collected from Windows monitoring agents deployed across 36K machines in a large organization with the goal of identifying malicious modules. We had to address some unforeseen challenges encountered in a large-scale realistic deployment as ours. Using a large set of static,

behavioral and contextual features, we propose algorithms to identify modules similar to known blacklisted modules, as well as modules impersonating popular whitelisted software applications. Our validation based on internal AV scanning, VirusTotal and manual investigation by security experts confirms a large number of detected modules as malicious, and results in high precision and low number of false positives. In future work, we plan to extend our techniques to obtain higher coverage and identify other types of suspicious activities in this environment.

Acknowledgement

We are grateful to the enterprise who permitted us access to their endpoint data for our analysis. We would like to thank Justin Lamarre, Robin Norris, Todd Leetham, and Christopher Harrington for their help with system design and evaluation of our findings, as well as Kevin Bowers and Martin Rosa for comments and suggestions on our paper. We thank our shepherd Alfonso Valdes and anonymous reviewers for their feedback on drafts of this paper.

References

[1] Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: Proceedings of Recent Advances in Intrusion Detection. pp. 178–197. RAID (2007)

[2] Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Proceedings of Network and Distributed System Security Symposium. NDSS, vol. 9, pp. 8–11 (2009)

[3] Bianchi, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Blacksheep: Detecting compromised hosts in homogeneous crowds. In: Proceedings of of ACM Conference on Computer and Communications Security. pp. 341–352. CCS, ACM (2012)

[4] Bowers, K.D., Hart, C., Juels, A., Triandopoulos, N.: PillarBox: Combating next-generation malware with fast forward-secure logging. In: Proceedings of Research in Attacks, Intrusions and Defenses. pp. 46–67. RAID (2014)

[5] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of International Symposium on Software Testing and Analysis. pp. 122–132. ACM (2012)

[6] Chau, D.H., Nachenberg, C., Wilhelm, J., Wright, A., Faloutsos, C.: Polonium: Tera-scale graph mining and inference for malware detection. In: Proceedings of SIAM International Conference on Data Mining. SDM, SIAM (2011)

[7] Damballa: First zeus, now spyeye. look at the source code now! `https://www.damballa.com/first-zeus-now-spyeye-look-the-source-code-now/` (2011)

[8] Dash, M., Choi, K., Scheuermann, P., Liu, H.: Feature selection for clustering - a filter solution. In: Proceedings of International Conference on Data Mining. pp. 115–122. ICDM, IEEE (2002)

[9] Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of 2nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining. pp. 226–231. KDD, ACM (1996)

[10] Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings of IEEE Symposium on Security and Privacy. pp. 62–75. S&P, IEEE (2003)

[11] Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: Proceedings of ACM Conference on Computer and Communications Security. pp. 318–329. CCS, ACM (2004)
[12] Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting malware infection through IDS-driven dialog correlation. In: Proceedings of USENIX Security Symposium. pp. 12:1–12:16. SECURITY, USENIX Association (2007)
[13] Gu, Z., Pei, K., Wang, Q., Si, L., Zhang, X., Xu, D.: LEAPS: Detecting camouflaged attacks with statistical learning guided by program analysis. In: Proceedings of International Conference on Dependable Systems and Networks. pp. 57–68. DSN, IEEE/IFIP (2015)
[14] Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer (2009)
[15] He, X., Cai, D., Niyogi, P.: Laplacian score for feature selection. In: Proceedings of Advances in Neural Information Processing Systems. pp. 507–514. NIPS (2005)
[16] Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security 6(3), 151–180 (Aug 1998)
[17] Hu, X., Shin, K.G.: DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In: Proceedings of 29th Annual Computer Security Applications Conference. pp. 79–88. ACSAC (2013)
[18] Hu, X., Shin, K.G., Bhatkar, S., Griffin, K.: MutantX-S: Scalable malware clustering based on static features. In: Proceedings of USENIX Annual Technical Conference. pp. 187–198. ATC, USENIX Association (2013)
[19] Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: Proceedings of USENIX Security Symposium. pp. 351–366. SECURITY, USENIX Association (2009)
[20] Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: AccessMiner: Using system-centric models for malware protection. In: Proceedings of ACM Conference on Computer and Communications Security. pp. 399–412. CCS, ACM (2010)
[21] Lee, W., Stolfo, S.J.: Data mining approaches for intrusion detection. In: Proceedings of USENIX Security Symposium. SECURITY, USENIX Association (1998)
[22] Lee, W., Stolfo, S.J., Chan, P.K.: Learning patterns from unix process execution traces for intrusion detection. In: Proceedings of AAAI Workshop on AI Approaches to Fraud Detection and Risk Management. pp. 50–56. AAAI (1997)
[23] MANDIANT: APT1: Exposing one of China's cyber espionage units. Report available from www.mandiant.com (2013)
[24] Mandiant Consulting: M-TRENDS 2016. https://www2.fireeye.com/rs/848-DID-242/images/Mtrends2016.pdf (2016)
[25] McAfee Labs: Diary of a "RAT" (Remote Access Tool). https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/23000/PD23258/en_US/Diary_of_a_RAT_datasheet.pdf (2011)
[26] McAfee Labs: ZeroAccess Rootkit. https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/23000/PD23412/en_US/McAfee%20Labs%20Threat%20Advisory-ZeroAccess.pdf (2013)
[27] Neugschwandtner, M., Comparetti, P.M., Jacob, G., Kruegel, C.: Forecast: skimming off the malware cream. In: Proceedings of 27th Annual Computer Security Applications Conference. pp. 11–20. ACSAC (2011)
[28] Oprea, A., Li, Z., Yen, T., Chin, S.H., Alrwais, S.A.: Detection of early-stage enterprise infection by mining large-scale log data. In: Proceedings of 45th Annual International Conference on Dependable Systems and Networks. pp. 45–56. DSN, IEEE/IFIP (2015)
[29] Perdisci, R., Lee, W., Feamster, N.: Behavioral clustering of HTTP-based malware and signature generation using malicious network traces. In: Proceedings of Symposium on Net-

worked Systems Design and Implementation. pp. 391–404. NSDI, USENIX Association (2010)
[30] Rahbarinia, B., Balduzzi, M., Perdisci, R.: Real-time detection of malware downloads via large-scale URL → file → machine graph mining. In: Proceedings of ACM Asia Conference on Computer and Communications Security. pp. 1117–1130. AsiaCCS, ACM (2016)
[31] Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Journal of Computer Security 19(4), 639–668 (2011)
[32] Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of IEEE Symposium on Security and Privacy. pp. 144–155. S&P, IEEE (2001)
[33] Shin, S., Xu, Z., Gu, G.: EFFORT: A new host-network cooperated framework for efficient and effective bot malware detection. Computer Networks (Elsevier) 57(13), 2628–2642 (Sep 2013)
[34] Symantec: The Rebirth Of Endpoint Security. `http://www.darkreading.com/endpoint/the-rebirth-of-endpoint-security/d/d-id/1322775`
[35] Tamersoy, A., Roundy, K., Chau, D.H.: Guilt by association: large scale malware detection by mining file-relation graphs. In: Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining. pp. 1524–1533. KDD, ACM (2014)
[36] Verizon: 2015 data breach investigations report. `http://www.verizonenterprise.com/DBIR/2015/` (2015)
[37] Wicherski, G.: peHash: A novel approach to fast malware clustering. In: 2nd Workshop on Large-Scale Exploits and Emergent Threats. LEET, USENIX Association (2009)
[38] Yen, T.F., Heorhiadi, V., Oprea, A., Reiter, M.K., Juels, A.: An epidemiological study of malware encounters in a large enterprise. In: Proceedings of ACM Conference on Computer and Communications Security. pp. 1117–1130. CCS, ACM (2014)
[39] Yen, T.F., Oprea, A., Onarlioglu, K., Leetham, T., Robertson, W., Juels, A., Kirda, E.: Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In: Proceedings of 29th Annual Computer Security Applications Conference. pp. 199–208. ACSAC (2013)
[40] Zeng, Y., Hu, X., Shin, K.G.: Detection of Botnets Using Combined Host- and Network-Level Information. In: Proceedings of International Conference on Dependable Systems and Networks. pp. 291–300. DSN, IEEE/IFIP (2010)

A Feature set

Our feature set includes features with different types, such as string, set, binary, and numerical attributes. Table 7 displays the full set of features used for our analysis, as well as their category and type.

B Case Studies

In this section, we present several detailed case studies of our findings. First, we detail two clusters of similar modules we identified, one with executable modules and another with DLLs, and we highlight the features that our new findings share with blacklisted modules. Second, we give more details on some of the detected outliers and emphasize the difference from the legitimate whitelisted modules they impersonate.

B.1 Similarity

We found 12 unknown modules all with different file names, but similar to a blacklisted module `house of cards s03e01∼.exe`. These modules impersonate popular movie

Category	Sub-category	Feature	Description	Type
Static	Descriptive	Description	File description	String
		Company name	Name of company	String
		Imported DLLs	Name of all imported DLLs	Set
		Section names	Name of all section names	Set
	Numerical	File size	Size of module	Integer
		PE size	Size from PE header	Integer
		PE timestamp	Time when PE file was created	Date
		Entropy	Module code entropy	Real
		DLL count	Number of imported DLLs	Integer
	Attributes	Icon present	Is icon present?	Binary
		Version information present	Is version information present?	Binary
		PE type	Type of PE (32 or 64 bit)	Binary
		PE machine	Type of targeted CPU (Intel 386, AMD64 etc.)	Categorical
		Packed	Is module obfuscated by a packer?	Binary
		.NET	Is it built with .NET?	Binary
		Signature	Signature name	String
		Signature valid	Is signing certificate issued by a trusted authority?	Binary
Behavior	File-system access	Written/Renamed executables	Avg. number of executables written/renamed	Real
	Process access	Created processes	Avg. number of created processes	Real
		Opened processes	Avg. number of opened processes	Real
	Network connections	Set of domains	Set of domain names connected to	Set
		Set of IPs	Set of IP addresses connected to	Set
Context	Module path	Path level	Avg. number of levels in path	Real
		Path_System	Is located in System folder?	Real
		Path_Windows	Is located in Windows folder?	Real
		Path_ProgramFiles	Is located in ProgramFiles folder?	Real
		Path_ProgramData	Is located in ProgramData folder?	Real
		Path_AppDataLocal	Is located in AppDataLocal folder?	Real
		Path_AppDataRoaming	Is located in AppDataRoaming folder?	Real
		Path_User	Is located in user-specific folder?	Real
		Number executables	Avg. number of executables in same folder	Real
		Number executables same company	Avg. number of executables with same company in same folder	Real
		Number non-executables	Avg. number of non-executables in same folder	Real
		Number sub-folders	Avg. number of sub-folders in same folder	Real
		Machine count	Number of installations	Integer
	Destination path	Dest_SamePath	Is destination path same as the module path?	Real
		Dest_DifferentPath	Is destination path different than the module path?	Real
		Dest_System	Is destination in System(syswow64/system32) folder?	Real
		Dest_Windows	Is destination in Windows folder?	Real
		Dest_ProgramFiles	Is destination in ProgramFiles folder?	Real
		Dest_ProgramData	Is destination in ProgramData folder?	Real
		Dest_AppDataLocal	Is destination in AppDataLocal folder?	Real
		Dest_AppDataRoaming	Is destination in AppDataRoaming folder?	Real
		Dest_User	Is destination in user-specific folder?	Real
		Dest_Temp	Is destination in Temp folder?	Real
	Metadata	Administrator	Does owner have administrator privileges?	Real
		Hidden attribute	Does file have hidden attribute set?	Real
		Days since creation	Avg. days since first observed on hosts	Real
	Auto-start	Auto_Services	Does the module have auto-start for services?	Real
		Auto_ServiceDLL	Does the module have auto-start for service DLL?	Real
		Auto_Logon	Does the module have auto-start for logon?	Real
		Auto_ScheduledTasks	Does the module have auto-start for scheduled tasks?	Real

Table 7. Final list of features. To note, all contextual features and numerical behavior features are computed by averaging the corresponding values across all hosts including the module.

or application names such as `Fifty Shades of Grey`$\sim$`.exe` and `VCE Exam Simulator`$\sim$`.exe` to deceive users. They all imported a single DLL (`KERNEL32.dll`) and used the same very common section names (`.text, .rdata, .data, .rsrc, .reloc`). One of them is even signed with a rogue certificate. Interestingly, these modules could not be grouped together only based on their static features, as these are common among other modules. However, when we consider the behavioral and contextual features, they are similar in some unusual ways. For instance, these modules write executables to a temp directory under `AppData` and create processes from that location. Moreover, they used the same autostart method (AutoLogon) to be persistent in the system and they reside in the same path under the `ProgramData` folder.

Another DLL cluster including 15 unknown and 1 blacklisted modules is intriguing as they have randomized 14-character file names (e.g. `oXFV2lbFU7dgHY.x64.dll`). The modules are almost identical in their features except for slightly different entropy values and creation dates. VirusTotal reported 10 of them, but different modules were detected by different number of AVs. One of them was not detected initially, but when we queried VirusTotal later the module was detected by 29 AVs. After eight months, the remaining 5 modules have not yet been detected by any AVs in VirusTotal but confirmed manually by the security analysts.

B.2 Outlier Detection

Our system identified 2 blacklisted and 3 unknown modules of `services.exe` as outliers. We found out that one of them was infected by ZeroAccess [26], a Trojan horse that steals personal information, replaces search results, downloads, and executes additional files. This module was confirmed by VirusTotal one week later after our detection. For the remaining two, we performed manual analysis. One of the modules has a description in Korean without a company name and signature. It has additional section names `.itext, .bss, .edata, .tls` compared to the legitimate process. The module imports some common DLLs such as `kernel32 .dll`, `user32.dll`, `oleaut32.dll`, but also imports `shell32.dll` and `wsock32.dll`, which is unusual for benign variants of `services.exe` modules. In addition, the module size is $\sim$ 1MB whereas other whitelisted modules have sizes between 110KB to 417KB. Unfortunately, no behavior features were captured in this module but it has several suspicious contextual features. The module is installed in only a single machine with hidden attributes and it is located in `C:\Windows\winservice` instead of `C:\Windows\System32`. The second detected `services.exe` module is missing the signature field and imports different set of DLLs. Even though the module is 32 bit, the DLLs it imports are usually included in 64-bit versions of benign `services.exe`. It also has some suspicious contextual features since it is installed only in a single machine relatively recently and its file system path is $\sim$`\Download\ffadecffa baffc` instead of the usual `C:\Windows\System32`. Both of these modules were confirmed as malicious by security experts in the organization.