Mining Data Provenance to Detect Advanced Persistent Threats *

Mathieu Barre †
INRIA
mathieu.barre@inria.fr

Ashish Gehani Vinod Yegneswaran
SRI International
{gehani,vinod}@csl.sri.com

Abstract

An advanced persistent threat (APT) is a stealthy malware instance that gains unauthorized access to a system and remains undetected for an extended time period. The aim of this work is to evaluate the feasibility of applying advanced machine learning and provenance analysis techniques to automatically detect the presence of APT infections within hosts in the network. We evaluate our techniques using a corpus of recent APT malware. Our results indicate that while detecting new APT instances is a fundamentally difficult problem, provenance-based learning techniques can detect over 50% of them with low false positive rates ($< 4\%$).

1. Introduction

Unlike traditional malware, APTs specifically target governments, political parties, and non-governmental organizations[17, 18]. They are typically developed by hackers funded by nation-states and commonly use social engineering techniques, like spear-phishing emails, to gain a foothold in the targeted network. Spear-phishing emails usually contain links or attachments that cause a piece of malware to be downloaded and executed on the victim's system. The targeted victims are carefully chosen individuals whose profile and responsibilities make them most likely to follow links or open the attached documents.

Our objective in this work is to develop a principled approach to building an automated APT classifier based on provenance. To that end, we execute real APT malware on virtual machines and capture their provenance activity using graphs output from the SPADE system [11]. The malware to be classified are particularly stealthy and are designed to reside within the system for an extended period. Hence, classical virus detection techniques tend to work poorly on this type of malware as they are rare and produce relatively few events. Indeed, hackers are constantly modifying them and improving them to avoid triggering existing anti-virus (AV) systems. We hypothesize that studying provenance patterns of the processes and files in a system may help explicate otherwise stealthy malicious behavior found in APTs. Our specific contributions in this paper include the following:

- Definition of the APT provenance classification problem;
- Study of the behavior of real APT malware from a provenance perspective;
- Definition of APT provenance feature vectors;
- Development of an APT classifier using computed features; and
- Comprehensive evaluation of the APT classifier.

2. Background

We focus our work on detecting Windows-based malware by collecting system provenance using the Process Monitor [28] utility and SPADE. Process Monitor is a monitoring tool for Windows that shows real-time file system, registry and process/thread activity. SPADE consumes the Process Monitor log file to generate a provenance graph that reveals the causal relationship between different files, processes, and users. SPADE supports the Open Provenance Model (OPM) [23] and includes controlling *Agent*, executing *Process*, and data *Artifact* node types, as well as dependency types that relate which process *wasControlledBy* which agent, which artifact *wasGeneratedBy* which process, which process *used* which artifact, which process *wasTriggeredBy* which other process, and which artifact *wasDerivedFrom* which other artifact. In SPADE, the Used and WasGeneratedBy edges correspond to reads and writes of data.

In addition to the causal relationships between the nodes, the provenance graphs created by SPADE contain numerous annotations that represent information contained in the Process Monitor log file. It provides details such as file name, file class, and edge category. Below, we provide examples of annotations used by nodes and edges.

A description of the process, if available
The id of the process
The version of the process, if available
The command line that launched the process
The path of the executable file associated with the process
The name of the process
The company linked with the process, if available
The architecture on which the process runs

Table 1: Annotations for Process nodes

Duration of the operation (time spent in I/O mode)
The time when the operation took place
Length of data read/written
The type of the edge that can be Used or WasGeneratedBy
The category which can be read, write, read metadata, write metadata
The operation performed such as write file, set registry key etc.

Table 2: Annotations for Used/WasGeneratedBy edges

* This material is based upon work supported by the National Science Foundation under Grants ACI-1547467 and CNS-1514503. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

† While visiting SRI.

TaPP 2019, June 3, 2019, Philadelphia, PA.

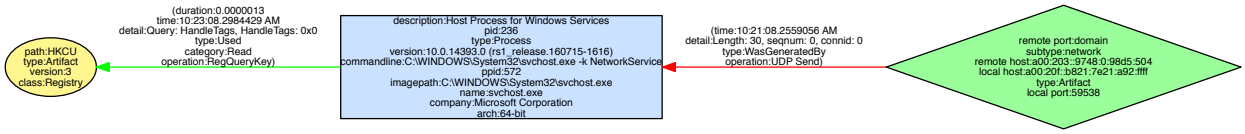

Figure 1: Output of SPADE printed with Graphviz illustrating the Windows system service manager (`svchost.exe`) reading a registry key and sending a UDP packet to port 504 of a remote host. Blue rectangles correspond to processes. Yellow ovals depict objects, such as files or registry entries. Green diamonds represent network flows. Green and red arrows illustrate read and write operations, respectively.

Figure 1 provides an example of SPADE output, printed using the Graphviz [12] software for graph representation and rendering. Here we see an artifact, specifically a registry entry, that is read by a process named *host process for Windows services* which has also sent a UDP packet. After looking at the *time* annotation, we observe that the UDP send happened before the registry read. In addition to OPM, SPADE defines a subtype for network artifacts, represented by a green diamond shape.

3. Experimental Dataset

The APT dataset used in this paper was obtained from Contagio [6], as summarized in Figure 4. While APT samples are typically hard to obtain due to the targeted and sensitive nature of such attacks, this dataset is interesting because it is a collection of malware that was successfully used in multiple high-profile campaigns. In Figure 2 we provide an overview of the mechanisms behind one of the representative APT instances called CosmicDuke, which was a spyware used in the attack against the Democratic National Committee in the 2016 elections. Here, the attached document contains a decoy which is the benign document that is printed on the screen and the dropper is the malicious component that attempts to install the initial malicious code in the system. In this case it installs the MiniDuke loader that subsequently injects the complete malware. Once all the stages have completed, the malware begins stealing information and exfiltrating it to locations on the Internet, from where the attacker can retrieve it.

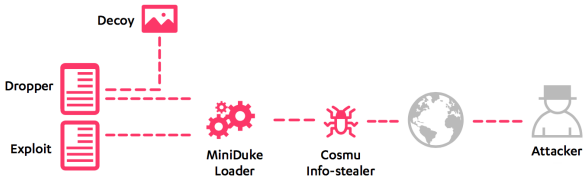

Figure 2: CosmicDuke's chain of actions

Our first approach to studying this corpus involved running many malware binaries in order to understand their malicious behavior. To mitigate potential harm to the local network, the workstation had to be secured and protected against any leak to avoid the dispersion of the threats on the enterprise network and beyond. Hence, we ran the malicious binaries in a virtual machine with no Internet connection. Not having an Internet connection can be an issue when studying malware because many of samples are programmed to try to detect if they are running in an experimental setup and to delete themselves if they notice something suspicious about the environment. In addition, most of these types of malware communicate with malicious servers that transmit orders to them and to which they send data collected from the victim's computer. Without an Internet connection, the malware can become completely inactive. To overcome the second hurdle, we collected information about the servers that malware attempt to contact and the requests they made to these servers. Then we set up a local web server that responded to the redirected malware queries with the appropriate data to "activate" the malware (when available). This depended on whether these servers were active at the time of the experiment. If so, we made offline requests to these servers to find the appropriate responses. Figure 3 presents the experimental setup.

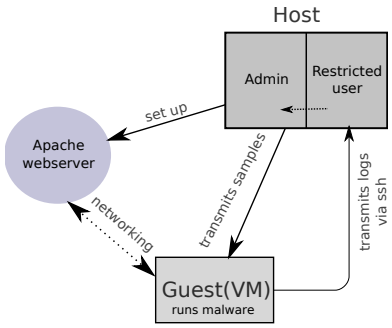

Figure 3: Experimental setup

Group	Family	# Binaries
APT29	CozyDuke	5
APT29	CosmicDuke	5
APT29	CloudDuke	4
APT29	PowerDuke	1
APT29	GeminiDuke	3
APT29	HammerDuke	1
APT29	OnionDuke	4
APT29	SeaDuke	2
APT29	MiniDuke	1
APT29	Hammertoss	2
APT29	Domain Fronting	2
APT29	CozyBear	2
APT29	Grizzly Steppe	1
APT29	MiniDionis	1
APT28	Sednit	14

Figure 4: APT malware corpus

Once this was all set up, we collected the log file of the process activities in the Windows virtual machine where the malware was running using ProcMon and transformed each log into a provenance graph using SPADE. Figure 4 provides a summary of the names and corresponding families of the binaries used in this study. Some of the binaries didn't produce meaningful activities on our virtual machine and these were removed from our analysis. This might be due to lack of triggers or embedded malware logic to detect virtual machine environments, ProcMon etc. This is a fundamental challenge in any malware analysis study and not the focus of our work.

Another key challenge was to be able to detect the malicious behavior and determine what was being attempted. To this end, we relied on different malware studies performed by anti-virus companies [2, 5, 8, 7, 9, 29] that described the observed behavior. For instance, we provide below a description of what the CozyDuke malware does when launched in a system:

Dropper writes to disk the main CozyDuke components and an encrypted configuration file.
Dropper checks whether anti-virus is installed.
Installs the components at a certain location and copies the file *rundll32.exe* to this location.
Establishing persistence: sets itself to be executed at startup.
Communicates with its Command and Control server using HTTP or HTTPS.
Attempts to gather comprehensive information on the victim's host configuration.
Takes screenshots of the victim host.
Deploys password stealer module.

Table 3: Summary of CozyDuke system activities

Among the list of actions performed by CozyDuke [8] there are some that we can detect when studying the provenance graph obtained from the run of CozyDuke on our virtual machine. For example, we notice the writing of a malicious configuration file `racss.dat`, of a malicious component `amdocl_ld32.exe`, and also notice the copy of the `rundll.exe` file. We can also observe that the malware writes to a lot of registry entries that can be used to establish its persistence. We can also see that it attempts to communicate with malicious servers (that we have replaced with our local web server). In addition, we observe that malicious processes read various files across the whole system, which is likely an attempt to gather information by the malware. However there are also activities that we can't distinctly see in the provenance graphs. They may not happen during our runs or they are not captured by the Process Monitor software. For example, we did not observe any traces with passwords being stolen or screenshots being grabbed in our graphs. An illustrative description of the provenance graph produced by CozyDuke's activities is provided in the Appendix.

4. The Classification Problem

We considered multiple variations of our classification problem. The first option was to simply consider the entire provenance graph as an instance of our problem and to be able to detect if a provenance graph is normal or abnormal based on the studies of benign graphs, as in [21]. This method avoids dealing with the problem of labelling since only graphs that are benign are used. In order to use learning algorithm, one can create a similarity measure between two graphs [32]. We can also define feature vectors [19] that describe the graphs as data points in the algorithms. However, in our case a provenance graph can represent hours of logs and contain several gigabytes of data. Hence, it seems problematic to consider a graph as only one instance of our problem. In addition, we can't learn much about attacks if the whole graph is detected as abnormal and the malware activity is camouflaged in harmless processes.

Another approach that we considered is using subgraphs of provenance graphs as instances of the classification problem. In that case, we can detect which subgraphs are abnormal and one provenance graph can be used to generate many subgraphs. However, there is still the issue of how to generate the subgraphs. Provenance graphs don't have a notion of closeness between nodes, preventing the use of classic community detection algorithms, though we note that there is some work in this domain [1].

Hence, we developed a third approach that considered each node in the provenance graphs as an instance. Only the Process nodes represent the actions inside the system. This allows us to detect which processes are acting in a harmful way, using system information. There are some challenges that need to be addressed. The first difficulty that we encountered is the labeling operation. In case of whole graphs we can say that a graph in which a malware has been launched is malicious. However, in the case of processes in a graph, there are two situations that occur. If the process node is part of a graph created from benign activity, we considered the process as benign. If the process is from a graph in which malware was present, we can't directly assign a label to this process. The process labeling method will be discussed in the next subsection. A second difficulty that occurs is the representation of the process to be used in a learning algorithm. We could design a notion of similarity between two processes but we choose to construct a feature vector that would describe the process activity and that could allow displaying sensitive distinctions between harmful and harmless behavior. The construction of the features is described below in Section 4.2.

4.1 Label Definition

An important challenge is labeling processes that appear in provenance graphs based on their potential malevolence. When launching malware in our virtual machine to obtain the mixed logs, we execute a precise sample of malware which creates a particular process in the provenance graph. We will call this process the malware origin. We know for sure that this process is malicious; so we can definitely label it as bad. We can also consider the processes that are spawned by the origin process – i.e., the processes connected to the malware origin with a *WasTriggeredBy* edge, to be malicious since they are directly created by the malware origin.

We can recursively extend this by considering propagation of the "badness" by *WasTriggeredBy* – i.e., every processes triggered by a process labelled as bad is labelled as bad. Once this is done, we have labelled as bad all the process that are connected to the malware origin by a path that contains only *WasTriggeredBy* edges. However, there can still exist many processes that will behave in a harmful way but that have not been labeled as bad. Indeed, malicious processes can write malicious code in files that will be read by usually benign processes, causing them to execute malicious operations. However in the provenance model used we don't know which part of a file a process has read, which means that every process that reads a corrupted file has to be considered as tainted.

This taint cannot simply be assimilated with "badness" since often processes that behave well most of the time can sometimes act in a harmful way. Hence, we decided to create a new label that would reflect the taint of these processes. Taint can be propagated following a causal order that corresponds to the order in which the edges and nodes appear. We consider labels to apply to every type of node in a provenance graph. If a file that has been written by a bad process is labeled as tainted, then processes that are not already labeled as bad that read tainted files are labeled as tainted; files that are written by tainted processes are labeled as tainted; and processes that are not already labeled as bad that are triggered by tainted process are labeled as tainted.

When following the aforementioned method of labeling different provenance graphs corresponding to mixed activities, we

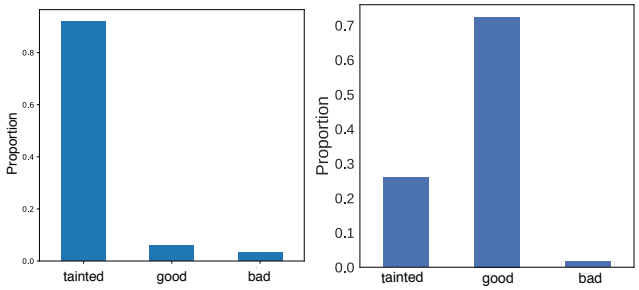

Figure 5: Proportion of each process label type in the mixed provenance graphs with (left) and without (right) consideration of registry tainting

obtain the distribution of labels represented in Figure 5 (left). We see that the vast majority of the processes end up tainted. There can be many explanations. Since the taint is something that propagates to every descendant, as soon as an important file or process has been tainted, all its descendants will be tainted. If a file read by most of the process is tainted, every process that will read the file is considered corrupted. This is problematic since most of the tainted programs are acting normally and may not have read the malicious piece of code present in the corrupted file. When observing which files transmit the taint the most, we see that malware frequently change the metadata of "important" registry entries, such as $HKLM\backslash System\backslash CurrentControlSet\backslash Control\backslash Session\ Manager$, that are read by a lot of processes. The analyzed APT instances perform the operation *RegSetInfoKey* on these registry entries. If we decide to ignore the taint transmission through the registry, we obtain the proportion of labels shown in Figure 5 (right).

However, having three labels could lead to taint that results in many processes with an ambiguous state. To take into account the taint, we give a smaller weight to tainted processes in the error formulas. Consequently, tainted processes will still be considered good but will count less than real good processes in the computation of error.

To address this distribution issue, we transform the discrete labeling problem into a continuous one. We attribute to each process a probability of being malicious depending on its distance and path to a known bad process. We make the hypothesis that the more steps there are between a process and a known malicious one, the less chance for the former to be malicious.

Processes that were labeled as bad would be assigned a value of 1, and those labeled as good would be assigned a value of 0. Probabilities for the other processes are obtained by propagation from the bad processes. The first propagation that we considered was simple. We labeled every node as 0, then labeled the bad ones as 1, and then following the order of appearance of edges, we did the following: if the incoming edge goes from a 0 label to a non-0 label l, then the 0 label becomes l/α with $\alpha > 1$. This simple label assignment strategy gives the distribution seen in Figure 6 (left). Label values are not continuously distributed and are concentrated around the middle value.

This can be improved by considering as label the mean of the labels of the parents with weights of α for bad tainted processes. If we τ_i denotes the label of the node i and π_i the parents of this

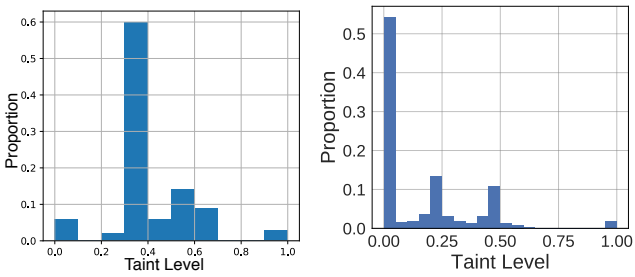

Figure 6: Distribution of the labels value with $\alpha = 1.25$ (left) and using the mean label of the ancestors (right)

node, we get $\tau_i = \frac{\sum_{j \in \pi_i} (\alpha.\tau_j.\mathbb{1}_{\tau_j > 0.5} + \tau_j.\mathbb{1}_{\tau_j \leq 0.5})}{\sum_{j \in \pi_i} (\alpha \mathbb{1}_{\tau_j > 0.5} + \mathbb{1}_{\tau_j \leq 0.5})}$. As shown in Figure 6 (right), we obtain a better distribution for the labels of the processes with low likelihood of being harmful.

To further improve the accuracy of labeling, we applied different rules depending on the type of files. We model the attribution probabilities for registry files as lower than those for system files. Probabilities of files can also depend on their extension, if they have one. By doing this, a process that reads a registry key in a registry file that has been written in by a malicious process, will be considered as potentially less dangerous than a process that reads in a `.dll` file with potentially harmful instructions to execute.

The continuous labels introduced here can be used as targets in a regression problem or transformed into weights [13] for classification. Since these labels are artificial, the objective should be to detect the heart of the attack and not identify processes with limited malicious potential.

4.2 Feature Vectors

Based on the provenance graph structure and the behavior observed in the malware study, we chose a set of basic features that should allow us to distinguish between malicious and benign activities. The following features aim to capture the significant behavioral patterns of APT malware. The collection of these features has been developed as a publicly available SPADE *filter*.

Count of outgoing Used edges
Count of incoming WasGeneratedBy edges
Count of incoming WasTriggeredBy edges
Total quantity of data read
Total quantity of data written
Average time that occurs between two Used edges
Average time that occurs between two WasGeneratedBy edges
Number of directories it reads in
Number of directories it writes in
Number of system files it used
Number of system files it generated
Number of registries it used
Number of registries it modified
Number of network artifacts it used
Number of network artifacts it generated
Number of *.exe*, *.dll*, *.dat*, *.bin* it used
Number of *.exe*, *.dll*, *.dat*, *.bin* it wrote
Number of different extension types it used
Number of different extension types it generated
Duration of the process lifetime

Preliminary Feature Set: We started with the aforementioned list of features and then developed additional features that are robust to

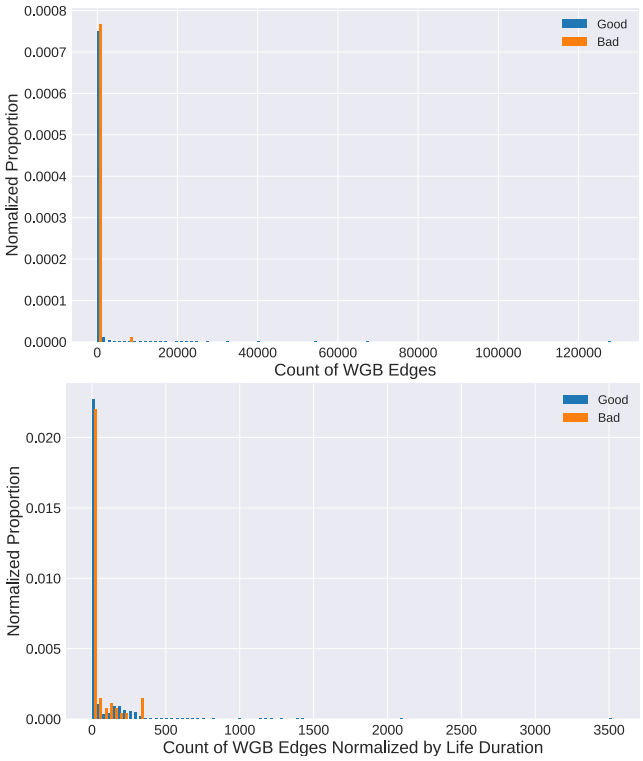

Figure 7: Distributions of the counts of WasGeneratedBy edges (x axis is the value of the feature, y axis is related to the proportion of process with the value x as feature)

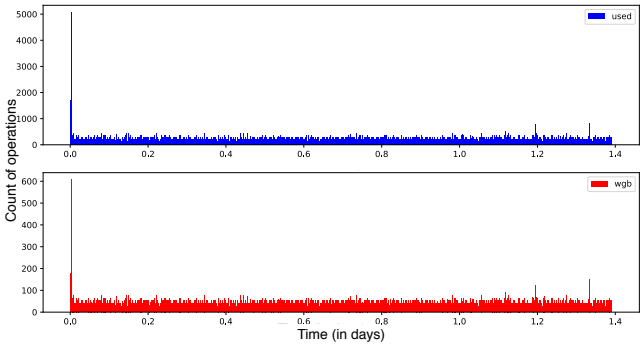

Figure 8: Activity of the process *cinfo.exe*, part of CozyDuke binary (1 unit is one day)

recording of system provenance in the APT context. Many features would have much higher values if a process ran for 1 hour in comparison to running for 1 minute. One solution is to normalize the features by their duration. Another solution could be to divide all features that count particular subtypes of edges by the total count of edges of that type class. However, with these kinds of transformations we lose information about which process had more activity than another. In Figure 7, we observe that dividing by duration permits us to have a better representation of the feature distributions.

Adding APT Inception Features: When observing the activity of a piece of malware over time (as represented in Figure 8) we notice that there is a large volume during the process inception stage. This flattened when normalizing the counts with life duration. To take into account the particular behavior that appears during the inception of a process, we add for each previous feature (except life duration and average time between two operations) the count during the first δ seconds of the life of the process without normalization. Furthermore, to account for the fact that we don't observe the early life of all processes, we add a feature that indicates whether a process has been launched before of after we started recording activity with ProcMon [26].

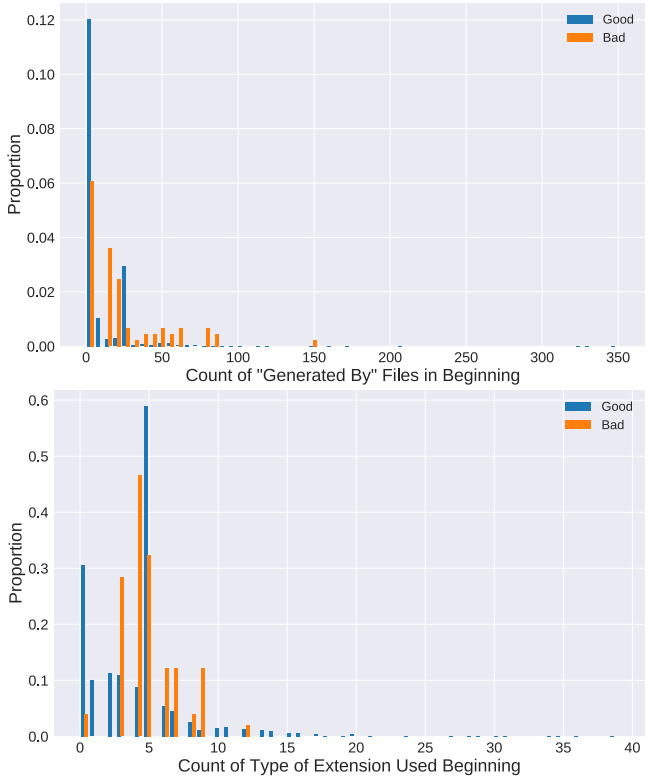

Figure 9: Distributions of APT inception features

In Figure 9 we can observe more distinctions between good and bad processes with these "early" features than with the previous ones. They should be more useful in the classification process. However, we do not always have access to the early life of a piece of malware on a host. Monitoring may start long after the malware installed itself in the system.

Accounting for agents, processes, and files names: We also added a feature based on the type of agent that controls a process. The agent name give us information on the authority level that started the process. We create a binary variable that indicates if an agent is has system privilege or is a user. This is valuable information since malicious processes are very often launched at user level.

We can also use the name of the process to get an indication of the possible risk that it represents. One way to do that could be to add a binary feature for each process name that appears in a training set. If the name of a process matches one in this set, the corresponding feature would be set to 1. This approach allows us to add a lot of information but it can have the drawback of making classification rely heavily on process names. To counter

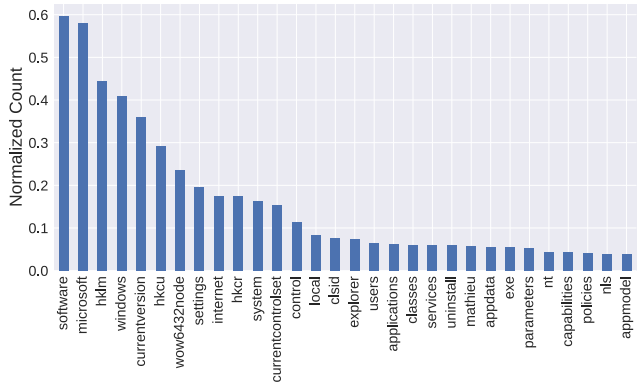

Figure 10: Proportion of words in paths generated by malicious processes

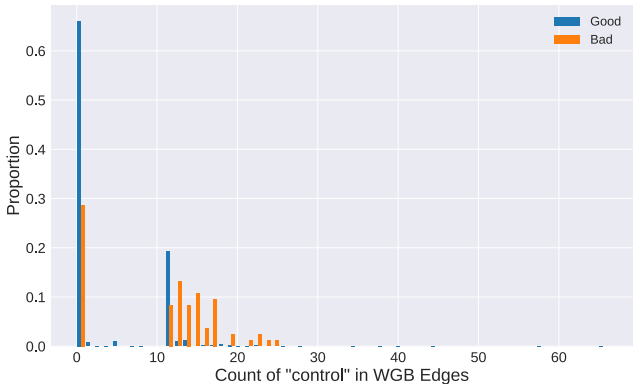

Figure 11: Frequency distributions of the word 'control' in the paths generated by benign and malicious processes

that we created a "grey list" of process names (and associated MD5 checksums). This list consists of process names that appear in a log of benign activity on Windows.

During classification, we also want to include the information contained in the names of the files that are used and generated by processes. We create two features – one for files *used* and another for files *generated* – for each word that appears in the paths of accessed files. Each feature could be the frequency of appearance of the word among the paths. However, in practice this approach results in too many features to work with. We can reduce the number of word by looking at the ones that are present in the paths of the files most frequently accessed by malicious processes. Figure 10 shows a histogram of the most often present string in paths generated by bad processes. Based on this we create a small vocabulary that we use to generate aforementioned features. We found that string-based features seem to bring useful information for the classification task. For instance Figure 11 exhibits a notable difference between the frequency distributions of the word 'control' in the paths generated by good and bad processes.

We added some features that could reflect malicious behavior. *writesAndExecutes* indicates if a process is launching another process using a file that it has previously written. *launchSameProcessName* indicates that a process launched another process with

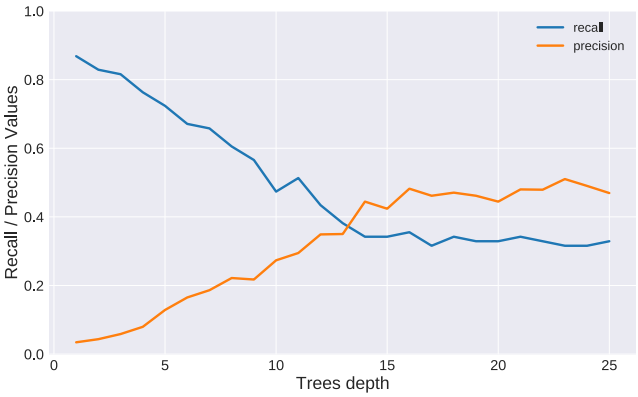

Figure 12: Recall and precision of a random forest model versus max depth of the trees in the forest

the same name as itself. This didn't manifest often in the activity of malicious binaries in our dataset but the behavior was deemed worth recording. We also added a feature called *suspiciousTriggers* that indicates if a process launched `cmd.exe`, `conhost.exe` or `rundll.exe`, which are the processes often launched by the malicious binaries in our training set.

5. Evaluation

Our evaluation dataset for cross validation included 4606 processes in the training set with 76 known bad processes. The processes come from 10 independent benign runs of varied durations (between 10 minutes and one hour). Benign Windows activity was generated using 14 runs of PCMark. Malicious activity was generated using intrusions from 11 types of APTs. Our test set comprises of 937 processes with five bad processes corresponding to four runs with malicious intrusions from four types of APTs – three binaries from families that are also present in the training set and one from another family).

We evaluate our results in terms of recall and precision. The first is defined as $recall = \frac{True\,Positive}{True\,Positive + False\,Negative}$, which corresponds to the proportion of malicious processes that are classified as bad. The second metric is defined as $precision = \frac{True\,Positive}{True\,Positive + False\,Positive}$, which is the proportion of actual malware among the processes that are classified as bad by the classifier. A low recall means that we are not able to detect the malware. A low precision means that we have many "false alarms".

5.1 Tests with Basic Feature Set

When we use the first set of features defined in Section 4.2, we do not obtain satisfying results in the cross validation phase. The recall was consistently less than 0.5 while precision was less than 0.4. Hence, we conclude that these features by themselves are not sufficient to discriminate between good and bad behavior. Figure 12 shows the precision and recall obtained in cross validation with the random-forest algorithm.

Figure 13 shows the importance of each feature in the random forest classifier. The y-axis values are calculated for each tree by summing the reduction of classification error at each node that involves the feature, and then averaging over the whole forest.

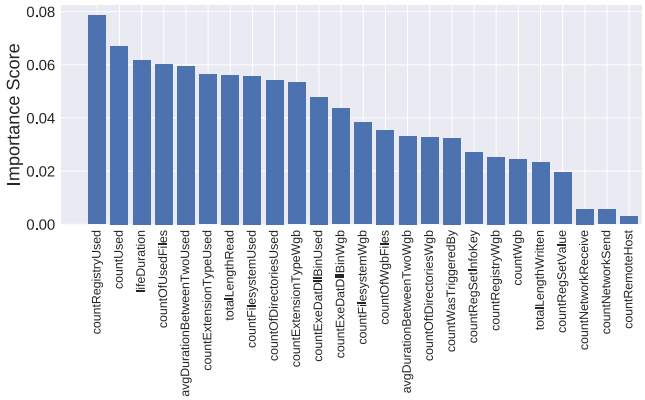

Figure 13: Features' importance as computed with a random forest model

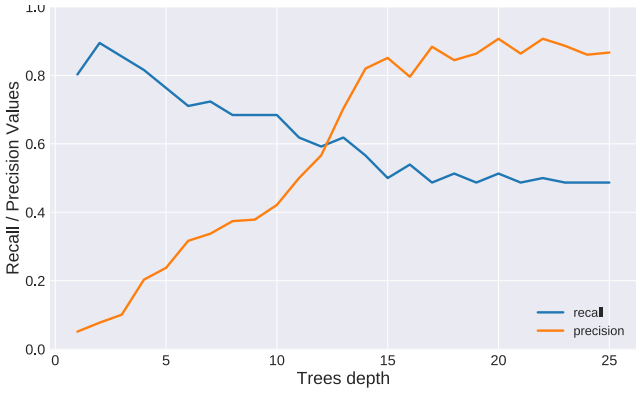

Figure 14: Recall and precision of a random forest model versus max depth of the trees in the forest

5.2 Tests with Inception Features

Next, we try to add the APT inception features in our model and watch for improvements. In Figure 14 we see that we are able to reach better precision and recall than before. This means that inception features are useful for detecting malicious processes. This can be confirmed with the feature importance graph in Figure 15. Indeed, many of the APT inception features are among the most important ones.

5.3 Using Process Names and Agent Information

Using all the process names as binary features helps increase the precision but is a problematic choice, as explained in Section 4.2. Using a grey list also has a positive effect on precision. We see in Figure 16 that we can reach higher precision values with same recall. While we can classify good processes using agent and process names, these are not features that are helpful for detecting new malware. This can also become a problem if new piece of malware uses a name that appears in the grey list.

5.4 Using Filenames and Recent Features

We chose a vocabulary of 21 words that were among the most frequently present in malware generated paths. This provides us

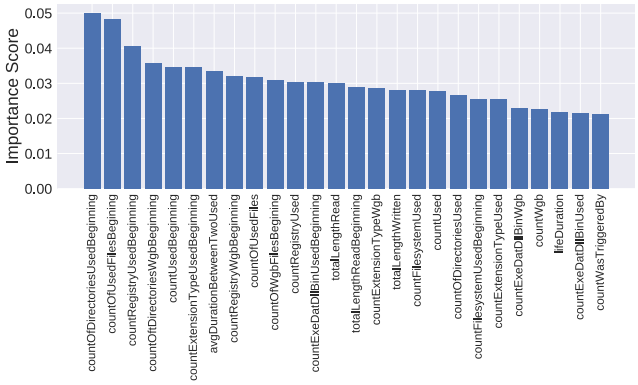

Figure 15: Features importance using APT inception features

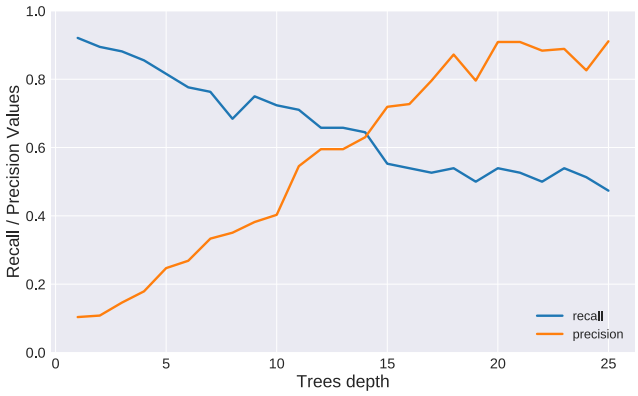

Figure 16: Recall and precision of a random forest model

42 new features that record the number of paths used or generated by a process that contains the chosen word. When adding the features mentioned in Section 4.2 we obtain a set of 91 features. With these features, the best cross validation result was a recall of 0.69 and a precision of 0.82 using a gradient boosting algorithm with a decision threshold of 0.62. We could not get better results than detecting 9/14 families during cross validation. It is possible to have models that have a very high precision but their recall is less than 0.5. Alternatively, we can obtain results with high recall but low precision. The more complex the model, the higher the precision. The simpler the model, the higher the recall becomes. Figure 17 shows the recall and precision using a random forest algorithm with all our features. The optimal result between recall and precision was obtained with gradient boosting.

In Figure 18 we see the importance that is attributed to each feature by a random forest algorithm. The process names remain important in the classification. Using malicious samples with existing Windows process names could help to improve generalization. While the recall is not great (0.6), the precision is much better if suboptimal recall (0.5) is acceptable. We focus on tree sizes that tradeoff precision and recall, trying to pick a balance. Since there is a significant increase in precision in exchange for little reduction of recall, we focus on the highest precision feasible. At this point in the design space, the system can identify half the malware (0.5

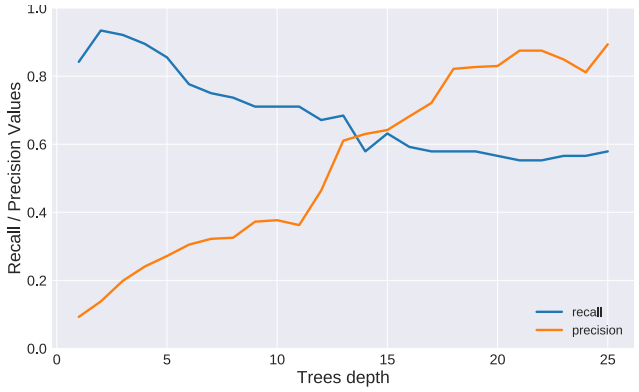

Figure 17: Recall and precision of a random forest model using all features

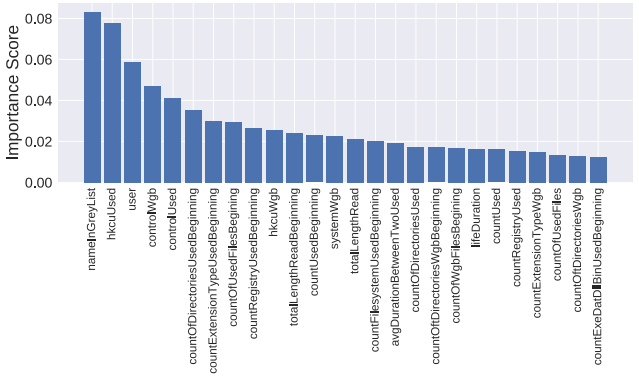

Figure 18: Features importance using our final feature set

recall in Figure 17) with very low false positives (0.98 precision in Figure 17). This is useful in practice.

To summarize, precision is only limited to 0.6 when we use only the feature set of counts of certain vertex and edge annotations. Once we add the features that are based on the lifetime of processes, the maximum precision increases to 0.8 when large enough decision trees are used. This makes sense since the APTs have a lot of activity at the outset that is captured in the second class of features. By adding knowledge about the filesystem paths used by malware, we are able to obtain very high precision. Finally, we provide a confusion matrix that provides a visual illustration of the relevance of various features in the Appendix.

6. Related Work

One of the simplest and most commonly used methods for malware detection is signature-based static analysis that looks for patterns in the code that indicate malicious behavior by comparing with a large database of known malware patterns. While this approach works well for previously encountered malware instances, these methods can easily be countered by simple transformations such as instruction substitution, `noop` or junk code insertion [24]. These methods are often inefficient for APTs, which are used on specific targets, with few samples available ahead of deployment, and designed to be stealthy. Virus scanners typically would not have a signature for APTs in their database nor have information to deem them malicious.

Another way to detect malware involves dynamic analysis. In this case executable files are run in sandboxed environments [33]. Every action performed by the executable is recorded and compared with known malware patterns. Several studies have explored the use of various dynamic analysis techniques for the purpose of malware behavioral classification [3, 27, 4]. A fundamental issue with dynamic analysis is code coverage – i.e., not knowing how long to run the malware and the required execution environment needed to ensure that malicious behavior is exercised. Hence, naive dynamic analysis may not find these APTs suspicious. For instance, CozyDuke instances do not exhibit bad behavior until they contact a malicious server that transmits instructions to be executed.

King *et al.* first introduced the notion of dependency graphs for intrusion tracking in the Backtracker system [16]. Our work enables significantly more fine-grained information flow tracking than their system. Some approaches, such as ProTracer [20], focus on optimizing instrumentation to reduce runtime overhead, while others, such as HERCULE [25], avoid new instrumentation by using logs from multiple existing sources. NoDoze [31] computes anomaly scores for edges in the provenance graph using a network diffusion algorithm. In contrast, we limit the propagation of decaying adversarial taint to process descendants. Other efforts monitor provenance from clusters of machines that run a homogeneous mix of applications [10, 30, 14] to detect anomalous activity.

We are also informed by efforts [15, 13] that use data mining and machine learning techniques on graphs representing system activity. Manzoor *et al.* [21] used graphs where vertices are similar to artifacts and processes in OPM while edges are system calls. To distinguish graphs representing an APT attack from those of benign activity, they define a notion of similarity, and use graph clustering to identify ones that are outliers. A key difference is that we consider graphs with labels for malicious process vertices. Thus, we are able to more accurately identify other graph elements that correspond to malicious APT activity. Recent systems, such as HOLMES [22], leverage a pre-defined model of the APT kill-chain to identify attacks across multiple platforms. We believe that adding support for detecting new classes of APTs can be automated with our approach since the we do not rely on expert-defined models.

7. Conclusion

We develop a provenance-based approach to detection and classification of APT malware and evaluate our system against a corpus of 48 malware instances from 15 families. Our analysis reveals several interesting findings. We find that inception features are particularly useful to detect malicious processes. While filenames and registry names are also useful, the precision is greatly increased when we combine this with additional context such as process lifetimes and path information. Overall, our system is able to deliver a detection rate of over 50% with a false positive rate under 4%. While such a result might seem less than ideal in a traditional malware detection scenario, the stealthy nature of APTs makes such a system a useful complement to other detection systems. In future work, we plan to extend our evaluation to a broader collection of APTs over longer time scales and integrate bare metal platforms into our evaluation framework.

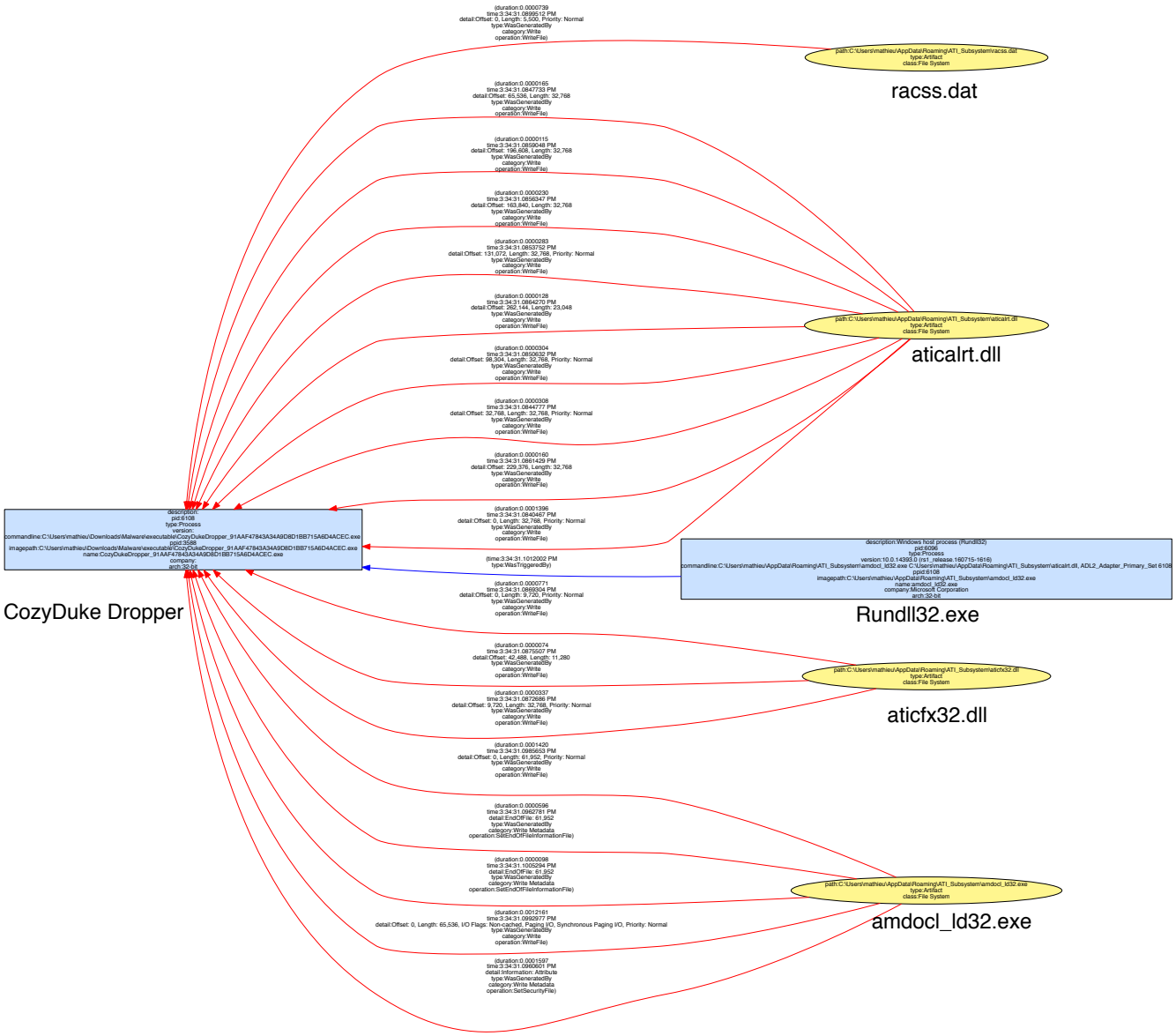

Figure 19: The *CozyDukeDropper* process (shown in blue at left) writes four files (`racss.dat`, `aticalrt.dll`, `aticfx32.dll`, and `amdocl_ld32.exe`, shown in yellow) in the `AppData\Roaming\ATI_Subsystem`directory, after which it runs one of them (`amdocl_ld32.exe`), passing another (`aticalrt.dll`) as an argument. (Due to their large number, registry accesses are omitted in the illustration.)

Appendix

Illustration of CozyDuke's Provenance

Figure 19 represents a part of the provenance graph. It shows (only system) files written by the CozyDuke Dropper as well as the process triggered by the dropper. The `amdocl_ld32.exe` process is the one that contacts the malicious server.

Such analysis was performed on 48 binaries in order to find patterns or particularities in provenance graphs linked with the malicious behavior of the APTs. APTs were observed to read numerous files in various system directories and modify many registry entries. Such empirical observations informed the design and development of our malware classification system.

Confusion Matrix of APT Provenance Features

In Figure 20 we provide a summarized representation of the correlations between the different features and also a label which corresponds to the ground truth about our processes (good or bad). If we look carefully, we can observe correlations among features that collect similar types of information.

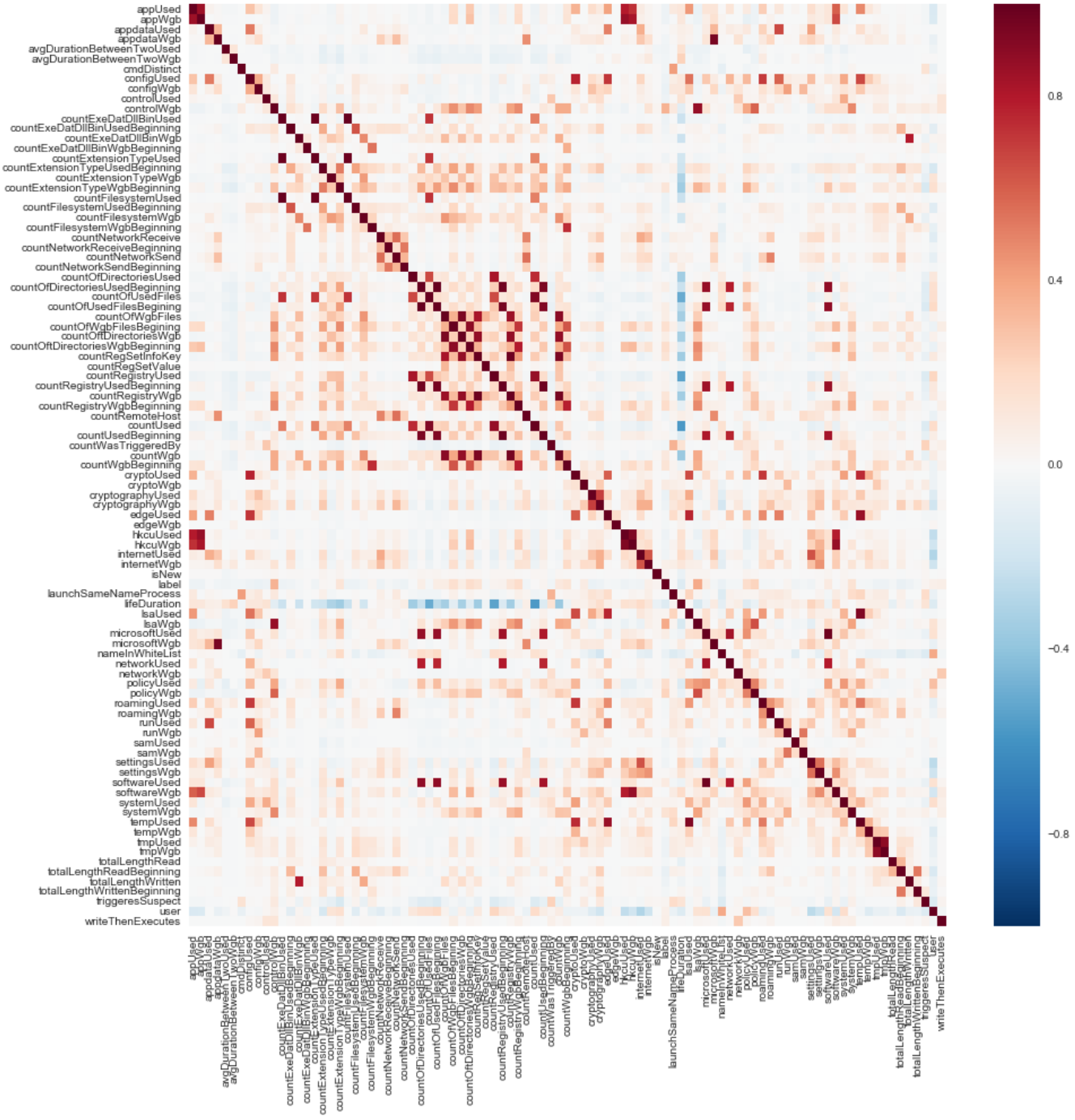

Figure 20: Correlation between the different features and labels

References

[1] Rui Abreu, Dave Archer, Erin Chapman, James Cheney, Hoda Eldardiry, and Adria Gascon, **Provenance segmentation**, *8th USENIX Theory and Practice of Provenance*, 2016.

[2] APT29 Domain Fronting With TOR, `https://www.fireeye.com/blog/threat-research/2017/03/apt29_domain_frontin.html`

[3] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario, **Automated classification and analysis of Internet malware**, *10th Symposium on Recent Advances in Intrusion Detection*, 2007.

[4] Ulrich Bayer, Paolo Comparetti,Clemens Hlauschek,Christopher Kruegel, and Engin Kirda, **Scalable, behavior-based malware clustering**, *17th Network and Distributed System Security Symposium*, 2009.

[5] Bears in the Midst: Intrusion into the Democratic National Committee, `https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/`

[6] Contagio malware dump, `http://contagiodump.blogspot.com/`

[7] CosmicDuke Cosmu with a twist of MiniDuke, `https://www.f-secure.com/documents/996508/1030745/cosmicduke_whitepaper.pdf`

[8] CozyDuke Malware Analysis, `https://www.f-secure.com/documents/996508/1030745/CozyDuke`

[9] The Dukes: 7 years of Russian cyberespionage, `https://www.f-secure.com/documents/996508/1030745/dukes_whitepaper.pdf`

[10] Ashish Gehani, Basim Baig, Salman Mahmood, Dawood Tariq, and Fareed Zaffar, **Fine-grained tracking of Grid infections**, *11th ACM/IEEE International Conference on Grid Computing*, 2010.

[11] Ashish Gehani and Dawood Tariq, **SPADE: Support for Provenance Auditing in Distributed Environments**, *13th ACM / IFIP / USENIX International Conference on Middleware*, 2012.

[12] Graphviz, `http://www.graphviz.org/`

[13] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu, **LEAPS: Detecting camouflaged attacks with statistical learning guided by program analysis**, *45th IEEE/IFIP Conference on Dependable Systems and Networks*, 2015.

[14] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer, **FRAPpuccino: Fault-detection through Runtime Analysis of Provenance**, *9th USENIX Workshop on Hot Topics in Cloud Computing*, 2017.

[15] Saiful Islam, Rafiqul Islam, A. Kayes, Chengfei Liu, and Irfan Altas, **A survey on mining program-graph features for malware analysis**, *10th International Conference on Security and Privacy in Communication Networks*, 2014.

[16] Samuel King and Peter Chen, **Backtracking Intrusions**, *19th ACM Symposium on Operating Systems Principles*, 2003.

[17] Stevens Le Blond, Adina Uritesc, Cedric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda, **A look at targeted attacks through the lens of an NGO**, *23rd USENIX Security Symposium*, 2014.

[18] Stevens Le Blond, Cedric Gilbert, Utkarsh Upadhyay, Manuel Gomez Rodriguez, and David Choffnes, **A broad view of the ecosystem of socially engineered exploit documents**, *25th Network and Distributed System Security Symposium*, 2017.

[19] Geng Li, Murat Semerci, Bulent Yener, and Mohammed Zaki, **Effective graph classification based on topological and label attributes**, *Journal of Statistical Analysis and Data Mining*, Vol. 5(4), 2012.

[20] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu, **ProTracer: Towards practical provenance tracing by alternating between logging and tainting**, *23rd Network and Distributed System Security Symposium*, 2016.

[21] Emaad Manzoor, Sadegh Milajerdi, and Leman Akoglu, **Fast memory-efficient anomaly detection in streaming heterogeneous graphs**, *22nd ACM Conference on Knowledge Discovery and Data Mining*, 2016.

[22] Sadegh Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, Venkat Venkatakrishnan, **HOLMES: Real-time APT detection through correlation of suspicious information flows**, *40th IEEE Symposium on Security and Privacy*, 2019.

[23] Luc Moreau, Ben Clifford, Juliana Freire, Yolanda Gil, Paul Groth, Joe Futrelle, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche, The Open Provenance Model core specification (v1.1), *Future Generation Computer Systems*, 2010.

[24] Andreas Moser, Christopher Kruegel, and Engin Kirda, **Limits of static analysis for malware detection**, *23rd Annual Computer Security Applications Conference*, 2007.

[25] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu, **HERCULE: Attack story reconstruction via community discovery on correlated log graph**, *32nd Annual Conference on Computer Security Applications*, 2016.

[26] SPADE ProcMon Reporter, `https://github.com/ashish-gehani/SPADE/wiki/Collecting-system-wide-provenance-on-Windows`

[27] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dussel, and Pavel Laskov, **Learning and classification of malware behavior**, *5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[28] Process Monitor, `https://docs.microsoft.com/en-us/sysinternals/downloads/procmon`

[29] En Route with Sednit, `https://www.welivesecurity.com/wp-content/uploads/2016/10/eset-sednit-part1.pdf`

[30] Dawood Tariq, Basim Baig, Ashish Gehani, Salman Mahmood, Rashid Tahir, Azeem Aqil, and Fareed Zaffar, **Identifying the provenance of correlated anomalies**, *26th ACM Symposium on Applied Computing*, 2011.

[31] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, Adam Bates, **NoDoze: Combatting threat alert fatigue with automated provenance triage**, *26th Network and Distributed System Security Symposium*, 2019.

[32] S. V. N. Vishwanathan, Nicol Schraudolph, Risi Kondor, and Karsten Borgwardt, **Graph kernels**, *Journal of Machine Learning Research*, Vol. 11(4), 2010.

[33] Carsten Willems, Thorsten Holz, and Felix Freiling, **Toward automated dynamic malware analysis using CWSandbox**, *IEEE Security and Privacy*, Vol. 5(2), 2007.