AMAL: High-Fidelity, Behavior-based Automated Malware Analysis and Classification

Aziz Mohaisen[1] and Omar Alrawi[2]

[1] Verisign Labs, Reston, VA, USA
[2] Qatar Computing Research Institute, Doha, Qatar

Abstract. This paper introduces AMAL, an operational automated and behavior-based malware analysis and labeling (classification and clustering) system that addresses many limitations and shortcomings of the existing academic and industrial systems. AMAL consists of two sub-systems, AutoMal and MaLabel. AutoMal provides tools to collect low granularity behavioral artifacts that characterize malware usage of the file system, memory, network, and registry, and does that by running malware samples in virtualized environments. On the other hand, MaLabel uses those artifacts to create representative features, use them for building classifiers trained by manually-vetted training samples, and use those classifiers to classify malware samples into families similar in behavior. AutoMal also enables unsupervised learning, by implementing multiple clustering algorithms for samples grouping. An evaluation of both AutoMal and MaLabel based on medium-scale (4,000 samples) and large-scale datasets (more than 115,000 samples)—collected and analyzed by AutoMal over 13 months—show AMAL's effectiveness in accurately characterizing, classifying, and grouping malware samples. MaLabel achieves a precision of 99.5% and recall of 99.6% for certain families' classification, and more than 98% of precision and recall for unsupervised clustering. Several benchmarks, costs estimates and measurements highlight and support the merits and features of AMAL.

Keywords: Malware, Classification, Automatic Analysis.

1 Introduction

Malware classification and clustering is age old problem that many industrial and academic efforts have tackled in the past. There are two common and broad techniques used for malware detection, that are also utilized for classification: signature based [24, 29, 16] and behavior based [19, 23, 31, 20, 28] techniques. Signature based techniques use a common sequence of bytes that appear in the binary code of a malware family to detect and identify malware samples. On the one hand, while signature-based techniques are very fast since they do not require the effort to run the sample to identify it (the whole decision is based on a static scan), their drawbacks is that they are not always accurate, they can be thwarted using obfuscation, and they require a prior knowledge, including a set of known signatures associated with the tested families.

The behavior-based approach uses artifacts the malware creates during execution. While this approach to analysis and classification is more expensive since it requires running the malware sample in order to obtain artifacts and features for behavior characterization, they tend to have higher accuracy in characterizing malware samples due

to the availability of several heuristics to map behavior patterns into families. Also, behavior characterization is agnostic to the underlying code and can easily bypass code obfuscation and polymorphism, relying on somewhat easier-to-interpret features.

Several academic studies used behavioral analysis for classification and labeling of malware samples. The first work to do so is by Baily et al. [5], in which it is shown that high-level features of the number of processes, files, registry records, and network events, can be used for characterizing and classifying (multi-class clustering) malware samples. However, the work falls short in many aspects. First, the technique makes use of only high-level features, and misses explicit low-level and implicit features (the authors leave that part for future work). Second, their work also relies on a small number of samples for validation of the technique, and the only source for creating ground truth for those samples was the side channel of antivirus labeling. Third, their technique is limited to one clustering algorithm (hierarchical clustering with the Jaccard index for similarity), and it is unclear how other algorithms perform for the same task. Last, their technique is for clustering, and does not consider two-family classification problems, so it is unclear how the features work with classification.

More recently, Bayer et al. [6] improved on the results in [5] in two ways. First, the authors contributed the use of locality-sensitive hashing (LSH) for memory-efficient clustering. Second, instead of using high-level behavior characteristics, the authors proposed to use low OS-level features based on API-hooking for characterizing malware samples. While effective, the technique has several shortcomings and limitations. First of all, malware samples scan for installed drivers and uninstall or bypass the driver used for kernel logging. More important, rootkits (like TDSS/TDL and ZeroAccess–both families are studied in our evaluation), a popular set of families of malware, are usually installed in the kernel and the kernel logger can be blind to all of their activities [27]. Rieck et al [23], uses the same API-hooking technique in [6] to collect artifacts and use them for extracting features to characterize malware samples. However, their technique suffers from a low accuracy rates, perhaps due to their choice of features. While they match the highest accuracy we achieve, our lowest accuracy of classification of a malware family is 20% higher than the lowest accuracy in their system.

In this paper we introduce AMAL, an operational and large-scale behavior-based solution for malware analysis and classification (both binary classification and clustering) that addresses the shortcomings of the previous solutions. To achieve its end goal, AMAL consists of two sub-systems, AutoMal and MaLabel. AutoMal builds on the prior literature in characterizing malware samples by their memory, file system, registry, and network behavior artifacts. Unlike [6], MaLabel uses low-granularity behavior artifacts that are even capable of characterizing differences between variants of the same malware family. On the other hand, and given the wide-range of functionalities of MaLabel, which includes binary classification and clustering, it incorporate several techniques with several parameters and automatically chooses among the best of them to produce the best results. To do that, and unlike the prior literature, MaLabel relies on analyst-vetted and highly-accurate labels to train classifiers and assist in labeling clusters grouped in unsupervised learning. Finally, the malware analysis and artifacts collection part of AMAL (AutoMal) has been in production since early 2009, and it enabled us to collect tens of millions, analyze several hundreds of thousands, and to manually label several tens of thousands of malware samples—thus collecting in-house intelligence beyond any related work in the literature.

The organization of the rest of this paper is as follows. In section 2, we review the related literature. In section 3 we describe our system in details, including AutoMal, the

automatic malware analysis sub-system and MaLabel, the automated malware classification sub-system. In section 4, we evaluate our system. In section 5 we outline some of the future work and concluding remarks.

2 Related Work

There has been plenty of work in the recent literature on the use of machine learning algorithms for classifying malware samples [5, 24, 23, 16, 22, 21]. These works are classified into two categories: signature based and behavior based techniques. Our work belongs to the second category of these works, where we used several behavior characteristics as features to classify the Zeus malware sample. Related to our work is the literature in [24, 19, 23, 31]. In [19], the authors use behavior graphs matching to identify and classify families of malware samples, at high cost of graph operations and generation. In [23, 24], the authors follow a similar line of thoughts for extracting features, and use SVM for classifying samples, but fall short in relying on a single algorithm and using AV-generated labels (despite their pitfalls).

To the best of our knowledge, the closest work in the literature to ours is the work in [5, 6, 23] with the shortcomings highlighted earlier. Related to our use of network features is the line of research on traffic analysis for malware and botnet detection, reported in [15, 11, 12] and for the particular families of malware that use fast flux, which is reported in [13, 18]. Related to our use of the DNS features for malware analysis are the works in [3, 4, 8]. None of those studies are concerned by behavior-based analysis and classification of malware beyond the use of remotely collected network features for inferring malicious activities and intent. Thus, although they share similarity with our work in purpose, they are different from our work in the utilized techniques.

The use of machine learning techniques to automate classification of behavior of codes and traffic are heavily studied in the literature. The reader can refer to recent surveys in [26] and [25]. More related work is deferred to the technical report [17].

3 System Design

The ultimate goal of AMAL is to automatically analyze malware samples and classify them into malware families based on their behavior. To that end, AMAL consists of two components, AutoMal and MaLabel. AutoMal is a behavior-based automated malware analysis system that uses memory and file system forensics, network activity logging, and registry monitoring to profile malware samples. AutoMal also summarizes such behavior into artifacts that are easy to interpret and use to characterize and represent individual malware samples at lower level of abstraction.

On the other hand, MaLabel uses the artifacts generated by AutoMal to extract unified representation, in the form of feature vectors, and builds a set of classifiers and clustering mechanisms to group different samples based on their common and distinctive behavior characteristics. For binary classification, AutoMal builds classifiers trained from highly-accurate, manually-inspected, analyst-vetted and labeled malware samples. MaLabel then uses the classifier to accurately classify unlabeled samples into similar groups, and to tell whether a given malware sample is of interest or not. Finally, MaLabel also provides the capability of clustering malware samples based on their behavior into multiple-classes, using hierarchical clustering with several settings to label such clusters. To perform highly accurate labeling, MaLabel uses high-fidelity expert-vetted training labels among other methods. With those overall system design goals and objectives, we now proceed to describe the system flow of both AutoMal and MaLabel.

3.1 System Flow

AutoMal: Behavior-based Malware Analyzer AutoMal is an operational system used by many customers, including large financial institutions, AV vendors, and internal users (called analysts). AutoMal is intended for a variety of users and malware types, thus it supports processing prioritization, multiple operating system and format selection, runtime variables and environment adjustment, among other options. The main features of AutoMal are as follows. 1) Sample priority queue: Allows samples to have processing priority based on submission source. 2) Run time variable: Allows submitter to set run time for the sample in the virtual machine (VM) environment. 3) Environment adjustment: Allows submitter to adjust operating system (OS) environment via script interface before running a sample. 4) Multiply formats: Allows submission of various formats like, EXE, DLL, PDF, DOC, XSL, PPT, HTML, and URL. 5) VMware-based: Uses VMware as virtual environment. 6) OS selection: Allows submitter to select operating system for the VM, supports Windows XP, 7, and Vista with various Service Packs (SP). Adding a new OS to AutoMal systems requires very little effort. 7) Lower Privilege: Allows submitter to lower the OS privilege before running a sample. By default, samples run as a privileged user in Windows XP. 8) Reboot option: Allows submitter to reboot the system after a sample is executed to expose other activities of malicious code that might be dormant.

AutoMal is a malware analysis system that comprises of several components, allowing it to scale horizontally for parallel processing of multiple samples at a time. An architectural design consists of a sample submitter, controller, workers (known as virtual machines, or VMs), and back-end indexing and storage component (database). Each component is described in the following:

Samples submitter. The submitter is responsible for feeding samples to AutoMal. The samples are selected based on their priority in the processing queue. Given that AutoMal has multiple sources of sample input including, customer submissions, internal submissions, and AV vendor samples, prioritization is used. Each of the samples are ranked with different priority with customer submissions having the highest priority followed by the internal submissions and finally the AV vendor feeds. When the system is ideal, AutoMal's controller fetches samples for processing from the process queue, which has the highest priority.

Controller. The controller is the main component of AutoMal and it is responsible for orchestrating the main process of the system. The controller fetches highest priority samples from the queue with the smallest submission time (earliest submitted) and processes them. The processing begins by the sample being copied into an available VM, applying custom settings to the VM, if there are any, and running the sample. The configuration for each VM is applied via a python agent installed on each VM allowing the submitter to modify the VM environment as they see fit. For example if an analyst identifies that a malware sample is not running because it checks a specific registry key for environment artifact to detect the virtual environment, the analyst can submit a script with the sample that will adjust the registry key so the malware sample fails to detect the virtual environment and proceed to infect the system. The agent also detects the type of file being submitted and runs it correctly. For example, if a DLL file is submitted, the agent will install the DLL as a Windows Service and start the service to identify the behavior of the sample. If a URL is submitted, the agent would launch Internet Explorer browser and visit the URL. After the sample is run for the allotted time, the controller pauses the VM and begins artifact collection. The controller runs several tools to collect the following artifacts: 1) File system: files created, modified,

and deleted, file content, and file meta data. 2) Registry: registry created, modified, and deleted, registry content, and registry meta data. 3) Network: DNS resolution, outgoing and incoming content and meta data. 4) Volatile Memory: This artifact is only stored for one week to run YARA signatures [2] (details are below) on the memory to identify malware of interest.

The file system, registry, and network artifacts and their semantics are extracted from the VMware Disk (VMDK) [30] and the packet capture (PCAP) file. The artifacts and their semantics are then parsed and stored in the back-end database in the corresponding tables for each artifact. The PCAP files are also stored in the database for record keeping. The VMware machine also saves a copy of the virtual memory to disk when paused. The controller then runs our own YARA signatures on the virtual memory file to match any families that our analysts have identified, and tags them accordingly. The virtual memory files are stored for 1 week on the AutoMal then discarded due to the size of each memory dump. For example, if the malware sample is run in a VM that has 512 MB of RAM then the stored virtual memory file would be 512 MB for that sample plus the aforementioned artifacts. Storing virtual memory files indefinitely does not scale hence we discard them after 1 week.

YARA signatures: YARA signatures are static signatures used to identify and classify malware samples based on a sequence of known bytes in a specific malware family. Our analysts have developed several YARA signatures based on their research and reverse engineering of malware families. Developing these signatures is time consuming because they require reverse engineering several malware samples of a family and then identifying a specific byte sequence that is common among all of them. A YARA signature is composed of 3 sections, meta section, string section, and condition section.

In our system we did not utilize memory signatures as a feature for classification or clustering because not every sample in our system has those artifacts available. We only store the memory artifacts for one week, hence we only have a window of one week that covers a small set of malware processed in AutoMal. If we identify a feature of importance in memory we can modify our system to log those features for future samples and we can add it to our feature set. We currently utilize memory files and YARA signatures to classify samples based on our analysts experience for malware families. We augment this information with our behavior-based classification and clustering for automatic labeling.

Workers. The workers' VMs are functionally independent of the controller, which allows the system to add and remove VMs without affecting the overall operation of the system. The VMs consist of VMDK images that have different versions of OSes with different patch levels. The current system supports Windows XP, Vista, and 7 with various service packs (SP). The VMs also have software such as Microsoft Office, Adobe Reader, and a python agent used to copy and configure the VM by the controller. The software installed on the VMs vary based on OS version. For most samples reported in this paper in section 4, we used VMs with Windows XP SP2 and with several software packages and programs installed, including Microsoft Office 2007, Adobe Acrobat 9.3, Java 6-21, FireFox 3.6, Internet Explorer 6, Python 2.5, 2.6, and VMware Tools. For hardware configuration for the VMs see Table 2 (all software packages are trademarks of their corresponding producers). This choice of OS was necessitated by the fact that infections are reported by customers on that OS. However, in case where samples are known to be associated with a different OS version, the proper OS is chosen with similar software packages.

Backend storage – database. The collected artifacts are parsed into a MySQL database [1] by the controller. The database contains several tables like *files*, *registry*, *binaries*, *PCAP* (packet captures), *network*, *HTTP*, *DNS*, and *memory_signature* table. Each of the table contains meta data about the collected artifacts with exception to *PCAP* and *binaries* table. The *binaries* table stores files meta data and content where the *files* table stores meta information about files created, modified, and deleted per sample run. The *files* table contains parsed meta data from the *binaries* table. The *PCAP* table is large in size, and stores the complete raw network capture of the sample during execution which would include any extra files downloaded by the sample. The *HTTP*, *DNS*, and *network* tables store parsed meta data from the *PCAP* table for quick lookups. **MaLabel: Automated Labeling.** MaLabel is a classification and clustering system that takes behavior profiles containing artifacts generated by AutoMal, extracts representative features from them, and builds classifiers and clustering algorithms for behavior-based group and labeling of malware samples. Based on the class of algorithm to be used in MaLabel, whether it is binary classification or clustering, the training (if applicable) and testing data into MaLabel is determined by the user. If the data is to be classified, MaLabel trains a model using a verified and labeled data subset and uses unlabeled data for classification. MaLabel allows for choosing among several classification algorithms, including support vector machines (SVM)—with a dozen of settings and optimization options, decision trees, linear regression, and k-nearest-neighbor, among others. MaLabel leaves the final decision of which algorithm to choose to the user based on the classification accuracy and cost (both run-time and memory consumption). MaLabel also has the ability to tune algorithms by using feature and parameter selection (more details are in section 4). Once the user selects the proper algorithm, MaLabel learns the best set of parameters for that algorithm based on the training set, and uses the trained model to output labels of classes for the unlabeled data. Those labels serve as an ultimate results of MaLabel, although they can be used to re-train the classifier for future runs. Using the same features used for classification, MaLabel uses unsupervised clustering algorithms to group malware samples into clusters. MaLabel features a hierarchal clustering algorithm, with several variations and settings for clustering, cutting, and linkage (cf. §4). Those settings are adjustable by the user. Unlike classification, the clustering portion is unsupervised and does not require a training set to cluster the samples into appropriate clusters. The testing selector component will run hierarchal clustering with several settings to present the user with preliminary cluster sizes and number of clusters created using the different settings. Based on the preliminary results the user can pick which setting fits the data set provided and can proceed to labeling and verification process.

3.2 Features and Their Representation

While the artifacts generated by AutoMal provide a wealth of features, in MaLabel we used only a total of 65 features for classification and clustering. The features are broken down based on the class of artifacts used for generating them into three groups—a listing of the features is shown in Table 1:

File system features. File system features are derived from file system artifacts created by the malware when run in the virtual enviornment. We use counts for files created, deleted, and modified. We also use counts for files created in predefined paths like %APPDATA%, %TEMP%, %PROGRAMFILES%, and other common locations. We keep a count for files created with unique extensions. For example if a malware sample creates 4 files on the system, a batch file (.BAT), two executable files (.EXE), and a configuration file (.CFG), we would count 3 for the number of unique extensions. Finally, we use the file size of created files; for that we do not use raw file size but create the distribution

Table 1. List of features. Unless otherwise specified, all of the features are counts associated with the named sample.

Class	Features
File system	Created, modified, deleted, file size distribution, unique extensions, count of files under selected and common paths.
Registry	Created keys, modified keys, deleted keys, count of keys with certain type.
Network	
IP and port	Unique destination IP, counts over certain ports.
Connections	TCP, UDP, RAW.
Request type	POST, GET, HEAD.
Response type	Response codes (200s through 500s).
Size	Request and response distribution.
DNS	MX, NS, A records, PTR, SOA, CNAME.

of the files' size. We divide the file size range, corresponding to the difference between the size of the largest and smallest files generated by a malware, into multiple ranges. We typically use four ranges, one for each quartile, and create counts for files with size falling into each range or quartile.

Registry features. The registry features are similar to the file features since we use counts for registries created, modified, and deleted, registry type like `REG_SZ`, `REG_BIN`, and `REG_DWORD`. While our initial intention of using them was exploratory, those features ended up very useful in identifying malware samples, especially when combined with other features (more details are in §4).

Network features. The network features make up the majority of our 65 features. The network features have 3 groups. The first group is raw network features, which includes count of unique IP addresses, count of connections established for 18 different port numbers, quartile count of request size, and type of protocol (we limited our attention to three popular protocols, namely the TCP, UDP, RAW). The second group is the HTTP features which include counts for POST, GET, and HEAD request; the distribution of the size of reply packets (using the quartile distribution format explained earlier), and counts for HTTP response codes, namely 200, 300, 400, and 500. The third category includes DNS features like counts for A, PTR, CNAME, and MX record lookups.

4 Evaluation

To evaluate the different algorithms in each application group, we use several accuracy measures to highlight the performance of various algorithms. Those measures are the classical used literature metrics: precision, recall, accuracy, and F-1 score.

4.1 Hardware and Benchmarking

In Table 2, we disclose information about the hardware used in AMAL. While the hardware equipment used in running MaLabel are not fully utilized, the hardware specifications used in AutoMal are important for its performance. For example, memory signatures and file system scans heavily depend on those specifications. For that, the parameters are selected to be large enough to run the samples and the hosting operating system, but not too large to make the analysis part infeasible within the allotted time for each sample. Notice that, and as explained earlier, the operating system used in AutoMal can be adjusted in the initialization before running samples. However, for consistency we use the same OS to generate the artifacts for the different samples.

4.2 Datasets

The dataset used in this work is mainly from AutoMal, and as explained earlier, is fed to the system by internal user and external customers. Internal users are internal analysts of malicious code, and external users of the system are customers, who could be security analysts in corporates (e.g., banks, energy companies, etc), or other antivirus companies who are partners with us (they do not pay fees for our service, but we mutually share samples and malware intelligence). The main dataset used in this study consists of $115,157$ malware samples. The set of samples used in this study is selected as a simple random sample from a larger population of malware samples generated over that period of time. More details on the samples are in [17].

Table 2. Benchmarking of hardware used for the different parts of our system. MaLabel 1 and MaLabel 2 are platforms used for clustering and classification, respectively.

Component	AutoMal VM	MaLabel 1	MaLabel 2
# CPUs	1	1	1
RAM	256MB	120GB	192GB
Hard drive	6GB	200GB	2TB
OS	Win XP*	CentOS 6	CentOS 6

Table 3. Malware samples and their labels used in the classification training and testing.

Size	%	Family	Description
1,077	0.94	Ramnit	File infector and a Trojan with purpose of stealing financial, personal, and system information
1,090	1.0	Bredolab	Spam and malware distribution bot
1,091	1.0	ZAccess	Rootkit trojan for bitcoin mining, click fraud, and paid install.
1,205	1.1	Autorun	Generic detection of autorun functionality in malware.
1,336	1.2	Spyeye	Banking trojan for stealing personal and financial information.
1,652	1.4	SillyFDC	An autorun worm that spreads via portable devices and capable of downloading other malware.
2,086	1.8	Zbot	Banking trojan for stealing personal and financial information.
2,422	2.1	TDSS	Rootkit trojan for monetizing resources of infected machines.
5,460	4.7	Virut	Polymorphic file infector virus with trojan capability.
7,691	6.7	Sality	same as above, with rootkit, trojan, and worm capability.
21,047	18.3	Fakealert	Fake antivirus malware with purpose to scam victims.
46,157	40.1	Subtotal	
69,000	59.9	Others	Small mal, $<$ 1k samples each
115,157	100	Total	

Labeling for validation: A selected set of families to which those samples belong (with their corresponding labels) are shown in Table 3. The dataset particularly includes 2086 samples that are entirely inspected and verified as Zeus or one of its variants by security analysts, while other labels are either generated using the same method (on a subset of the samples in the family) and the rest of the label makes use of census over returned antivirus detections. For that, we query a popular virus scanning service with 42 scan engines, and pass the MD5 of all samples in the larger dataset to it. We use the detection provided by the scan to create a census on the label of individual samples: if a sample is

detected and labeled by a majority of virus scanners of a certain label, we use that label as the ground truth (those labels are shown in Table 3). We note that the Zeus family reported in Table 3 is manually inspected and labeled by internal analysts, and results returned by the antivirus scanners for the MD5s belonging to samples this family either agree with this labeling, or assign generic labels to them, thus establishing that one can rely on this census method for labeling and validation.

4.3 High-fidelity Malware Classification

We focus on the binary classification problem using the Zeus malware family [9], given its unique ground truth, where every sample in this family is classified and labeled manually by analysts. We then show the evaluation of different algorithms implemented in MaLabel to classify other malware families using the same set of features used in Zeus. In all evaluations we use 10-fold cross validation—a formal definition and settings are provided in [17].

Classification of Analyst-vetted Samples MaLabel implements several binary classification algorithms, and is not restricted to a particular classifier. Examples of such algorithms include the support vector machine (SVM), linear regression (LR), classification trees, k-nearest-neighbor (KNN), and the perceptron method—all are formally defined along with their parameters in [17]. We note that KNN is not a binary classifier, so we modified it by providing it with proper (odd) k, then voting is performed over which class a sample belongs to. To understand how different classification algorithms perform on the set of features and malware samples we had, we tested the classification of the malware samples across multiple algorithms and provided several recommendations. For the SVM, and LR, we used several parameters for regularization, loss, and kernel functions (definitions are in [17]).

For this experiment, we selected the same Zeus malware dataset as one class, as we believe that the highly-accurate labeling provides high fidelity on the results of the machine learning algorithms. For the second class we generated a dataset with the same size as Zeus from the total population that excludes ZBot in Table 3. Using 10-fold cross validation, we trained the classifier on part of both datasets using the whole of 65 features, and combined the remaining of each set for testing. We ran the algorithms shown in Table 4 to label the testing set. For the performance of the different algorithms, we use the accuracy, precision, recall, and F-score.

The results are shown in Table 4. First of all, while all algorithms perform fairly well on all measures of performance by achieving a precision and recall above 85%, we notice that SVM (with polynomial kernel for a degree of 2) performs best, achieving more than 99% of precision and recall, followed by decision trees, which is slightly lagged by SVM (with linear kernel). Interestingly, and despite being simple and lightweight, the logistic regression model achieves close to 90% on all performance measures, providing competitive results. While they provide less accuracy than the best performing algorithms, we believe that all of those algorithms can be used as a building block in MaLabel, which can ultimately make use of all classifiers to achieve better results.

As for the cost of running the different algorithms, we notice that the SVM with polynomial kernel is relatively slow, while the decision trees require the most number of features to achieve high accuracy (details are omitted). On the other hand, while the dual SVM provides over 95% of performance on all measures, it runs relatively quickly. For that, and to demonstrate other aspects in our evaluation, we limit our attention to the dual SVM, where possible. SVM is known for its generalization and resistance to noise [23].

Features Ranking and Selection We also followed the recent literature [14, 8, 7] to rank the different features by their high-level category. We ran our classifier on the file system, memory (where available), registry, and network features independently. For the network features, we further ranked the connection type, IP and port, request/response type and size, and DNS as sub-classes of features. From this measurement, we found that while the file system features are the most important for classification—they collectively achieve more than 90% of precision and recall for classification—the port features are the least important. It was not clear how would the memory feature rank for the entire population of samples, but using them where available, they provide competitive and comparable results to the file system features. Finally, the rest of the features were ranked as network request/response and size, DNS features, then registry features. All features and their rankings are deferred to [17].

Table 4. Results of binary classification using several algorithms in terms of their accuracy, precision, recall, and F-score.

Algorithm	A	P	R	F
SVM Polynomial Kernal	**99.22**%	**98.92**%	**99.53**%	**99.22**%
Classification Trees	99.13%	99.19%	99.06%	99.13%
SVM Linear Kernal	97.93%	98.53%	97.30%	97.92%
SVM Dual (L2R, L2L)	95.64%	96.35%	94.86%	95.60%
Log. Regression (L2R)	89.11%	92.71%	84.90%	88.63%
K-Nearest Neighbor	88.56%	93.29%	83.11%	87.90%
Log. Regression (L1R)	86.98%	84.81%	90.09%	87.37%
Perceptron	86.15%	84.93%	87.89%	86.39%

4.4 Large Scale Classification

One limitation of the prior evaluation of the classification algorithm is its choice of relatively small datasets that are equal in proportion for training and testing, for both the family of interest and the mixing family. This, however might not be the case in operational contexts, where even a popular family of malware can be as small as 1% of the total population as shown in Table 3 for several examples. Accordingly, in the following we test how the different classifiers are capable of predicting the label of a given family when the testing set is mixed with a larger set of samples. For that, we use the labeled samples as families of interest, while the rest of the population of samples as the "other" family (they are collectively indicated as one class). We run the experiment with the same settings as before (5% is saved for training the classifier and the rest is used for testing). Where possible, we use 10-fold cross validation to minimize bias. In the following we summarize the results of seven of interest. The results are in Table 5.

First of all, we notice that although the performance measures are less than those reported for Zeus in section 4.3, we were still able to achieve a performance nearing or above 90% on all performance measures for some of the malware families. For the worst case, those measures where as low as 80%. While these measures are competitive compared to the state-of-the-art results in the literature (e.g., the results in [23] were as low as 60% for some families), understanding the reasons behind false alarms is worth investigation. To understand those reasons, we looked at the samples marked as false alarms and concluded the following reasons behind the degradation in the performance. First, we noticed that many of the labels used for the evaluation that resulted into the final result are not by analysts, but come from the census over antivirus scans—even though a census on a large number of AV scans provides a good accuracy, it is still imperfect. Second, we notice that the class of interest is too small, compared to the total population of samples, and a small error is amplified for that class—notice that this effect is unseen in [23] where classes are more balanced in size (e.g., 1 to 9 ratios versus 1 to 99 ratio in our case). Finally, part of the results is attributed to the relatively

similar context of the different families of malware samples, as shown in Table 3, thus in the future we will explore enriching the features to achieve higher accuracy.

Table 5. Binary classification of several malware families.

Family	A	P	R	F
ZAccess	85.9%	80.7%	94.3%	87.0%
Ramnit	91.0%	87.1%	96.3%	91.5%
FakeAV	85.0%	82.5%	88.8%	85.6%
Autorun	87.9%	85.2%	91.8%	88.4%
TDSS	90.3%	89.6%	91.2%	90.4%
Bredolab	91.2%	88.0%	95.3%	91.5%
Virut	86.6%	85.9%	87.5%	86.7%

Benchmarking and scalability. We benchmarked our 115,157 samples using several distance calculation algorithms and hierarchal clustering methods with a cut off threshold of 0.70. From this benchmarking, we observe the high variability of time it takes for computing the distance matrix, which is the shared time between all algorithms settings. For example, computing the distance matrix using the Jaccard index (which is the only distance measure used in the literature for this purpose thus far [6]) takes 5820 sec (97 min) whereas all other distance measures require between 27.8 and 36.2 minutes.

5 Conclusion

In this paper we introduced AMAL, the first operational large-scale malware analysis, classification, and clustering system. AMAL is composed of two subsystems, AutoMal and MaLabel. AutoMal runs malware samples in virtualized environments and collects memory, file system, registry, and network artifacts, which are used for creating a rich set of features. Unlike the prior literature, AutoMal combines signature-based techniques with purely behavior-based techniques, thus generating highly-representative features, and use them for both classification and clustering.

References

1. —. MySQL. `http://www.mysql.com/`, May 2013.
2. —. Yara Project: A malware identification and classification tool. `http://bit.ly/3hbs3d`, May 2013.
3. M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a dynamic reputation system for dns. In *USENIX Security Symposium*, 2010.
4. M. Antonakakis, R. Perdisci, W. Lee, N. V. II, and D. Dagon. Detecting malware domains at the upper dns hierarchy. In *USENIX Security Symposium*, 2011.
5. M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *RAID*, 2007.
6. U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.
7. L. Bilge, D. Balzarotti, W. K. Robertson, E. Kirda, and C. Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *ACSAC*, 2012.
8. L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *NDSS*, 2011.
9. N. Falliere and E. Chien. Zeus: King of the Bots. Symantec Security Response (`http://bit.ly/3VyFV1`), November 2009.

10. R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.
11. C. Gorecki, F. C. Freiling, M. Kührer, and T. Holz. Trumanbox: Improving dynamic malware analysis by emulating the internet. In *SSS*, 2011.
12. G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security Symposium*, 2008.
13. T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling. Measuring and detecting fast-flux service networks. In *NDSS*, 2008.
14. C.-Y. Hong, F. Yu, and Y. Xie. Populated ip addresses: classification and applications. In *ACM CCS*, pages 329–340, 2012.
15. G. Jacob, R. Hund, C. Kruegel, and T. Holz. Jackstraws: Picking command and control connections from bot traffic. In *USENIX Security Symposium*, 2011.
16. J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.
17. A. Mohaisen, O. Alrawi, and M. Larson. Amal: High-fidelity, behavior-based automated malware analysis and classification. Technical report, Verisign Labs, 2013.
18. J. Nazario and T. Holz. As the net churns: Fast-flux botnet observations. In *MALWARE*, pages 24–31, 2008.
19. Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *CSIIR Workshop*. ACM, 2010.
20. R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *USENIX NSDI*, 2010.
21. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, N. Modadugu, et al. The ghost in the browser analysis of web-based malware. In *USENIX HotBots*, 2007.
22. M. Ramilli and M. Bishop. Multi-stage delivery of malware. In *MALWARE*, 2010.
23. K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125, 2008.
24. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.
25. C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *IEEE Sec. and Privacy*, 2012.
26. R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*, 2010.
27. R. Strackx and F. Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *ACM CCS*, 2012.
28. W. T. Strayer, D. E. Lapsley, R. Walsh, and C. Livadas. Botnet detection based on network behavior. In *Botnet Detection*, 2008.
29. R. Tian, L. Batten, and S. Versteeg. Function length as a tool for malware classification. In *IEEE MALWARE*, 2008.
30. VMWare. Virtual Machine Disk Format (VMDK). `http://bit.ly/e1zJkZ`, May 2013.
31. H. Zhao, M. Xu, N. Zheng, J. Yao, and Q. Ho. Malicious executables classification based on behavioral factor analysis. In *IC4E*, 2010.