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

Aziz Mohaisen
Verisign Labs, VA, USA

Omar Alrawi*
Qatar Foundation, Doha, Qatar

ABSTRACT

This paper introduces AMAL, an operational automated and behavior-based malware analysis and labeling (classification and clustering) system that addresses many concerns and shortcomings of the literature. 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 (4,000 samples) and large-scale datasets (115,000 samples) 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 are an 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 [53, 63, 34] and behavior based [45, 52, 68, 47, 59] 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.

For example, antivirus companies use static signatures to detect a known malware, which completely miss in the case of a zero-day malware that has not been seen before [12]. Also malware has become more sophisticated by using polymorphic obfuscation, packing, and code rearranging to thwart antivirus signatures [55]. Antivirus companies also use heuristic signatures to detect and classify known malicious behavior. However, the problem with heuristics is that the signature groups families of malware together and gives them generic labels which are not useful for most security and intelligence applications. Even worse, labels given by antivirus companies to the same malware sample also vary by vendor. There are many inconsistencies and disagreement among the antivirus vendors for different malware families [7].

The behavior-based approach to classification 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. Thus, this technique does not require the expertise required for signature-based techniques—those techniques require reverse-engineering skills to create signatures [38, 56].

Indeed, several academic studies in the past made use of behavioral analysis for classification and labeling of malware samples. The first work to do so is by Baily et al. [7], 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 intended only for clustering, and does not consider two-family classification problems, so it is unclear how meaningful the features used in their technique to this problem are. Indeed, binary classification has an appealing unique business opportunity to it, and it was not considered before by any related work.

More recently, Bayer et al. [8] in another seminal work considered improving on the results in [7] in two ways. First, the authors contributed the use of locality-sensitive hashing (LSH), a probabilistic dimensionality reduction method, 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 the technique is demonstrated to be effective, it has several shortcomings and limitations. First of all, malware samples scan for installed drivers and

*This work was done while the author was with VeriSign.

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 [58]. Also, this work has been tested on only one hierarchical clustering algorithm, does not handle two-families classification, and relies on a set of small AV-labeled samples as a ground truth (despite their inconsistencies as highlighted in [8] and their inaccuracies as more recently shown in [42]).

Rieck et al [52], uses the same API-hooking technique in [8] to collect artifacts and use them for extracting features to characterize malware samples. However, in addition to the limitations common with the work in [8], 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. Their method is only limited to SVM classification, while we provide insight into several other learning algorithms and how they succeed or fail in classifying samples.

On the experimental comparison with the prior literature: The work in [52, 8, 7] motivated us and are compared to us in method and end results as shown above. It is worth noting that [7] does not provide any insight into accuracy, unlike [8], although different from our work in the aspects stated in earlier. An empirical comparison was considered with [8] beyond timing measurements, and we believe it is impossible to do with their feature-level dataset because we need to generate the same set of features used in our system from their binaries. Assuming obtaining binaries is possible, there is no guarantee to obtain the same behavior profiles by running those binaries, given that the samples are five years old, and infrastructure used by malware are likely down in disinfection efforts. Using their system for analyzing our malware samples to generate comparable features was not possible (the system does not allow for the large number of samples we have). Finally, a great part of our evaluation requires highly accurate labels, which are naturally obtained in our system, however this will not be available for the comparison benchmark bringing our comparison to sole reliance on biased AV labels. Finally, our solution is a complete operation system since 2009, unlike [7, 8], two clustering systems with less insight into operational aspects discussed in our work.

To this end, in this paper we introduce AMAL, the first 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 the best of our knowledge, all literature works that report on analyzing more than 75k malware samples use only static analysis techniques. 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. Furthermore, MaLabel tries to address the shortcomings and limitations of the prior work in practical ways. For example, unlike [8], 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 that goes beyond any related work in the literature. Unlike labeling (for training and validation) in the literature which is subject to errors, our labeling is done by analysts who are domain experts and human errors in their labeling are negligible. In this study, we evaluate MaLabel on variety of datasets obtained from AutoMal and show the effectiveness of AMAL in analyzing, characterizing, classifying, and labeling malware samples.

Why are both binary classification and clustering interesting (and important)? Binary and supervised classification is expensive, since it requires training a model with solid ground-truth, and using representative artifacts of the families (classes) of interest, both of which are nontrivially obtained. However, the cost of binary classification in our operational settings is justified. The classification problem is interesting to us because the volume of Malware we receive on daily basis is larger than the capacity of our analysts. Classification enables us to train a model on a small set of known malware and extrapolate our model to find new samples in large volumes of malware we receive on daily basis. For the majority of our customers, who consist of large financial institutes, the threat of banking Trojans and specifically new and unidentified variants of known families is of interest to them. We use this classification system to identify malware variants of the same family based on their behavior to inform our customers about new malware threats pertaining to their interest. Another benefit of this approach is that we are able to ignore or give low priority to known insignificant malware families. For example, by identifying FakeAV, a family that tricks the victim into purchasing a fake antivirus product by alerting them to fake infections on their system, we can use our classifier to filter out all FakeAV samples from our malware feed to focus on undiscovered threats that are relevant and interesting.

Clustering is also interesting to us as it always remains a challenging and open-ended problem. Clustering allows us to group malware samples of similar behavior together. For that we manually inspect the samples in each cluster, and augment the labels we have of identified malware over each cluster to identify the majority in that cluster. Furthermore, we use memory signatures, like YARA rules, to tag a specific signature of a family based on its memory artifacts and then use that information to label clusters. Finally in the rare cases of giving a cluster a name when all other methods are exhausted we would use majority voting of labels returned by a large number of antivirus scanners. The automatic labeling problem remains partly unsolved, as it is the case in the literature [7], and we leave improving on that for future work.

To this end, our contribution is as follows:

- We introduce AMAL, a fully automated system for analysis, classification, and clustering of malware samples. AMAL consists of two subsystems, AutoMal and MaLabel. AutoMal is a feature-rich and low granularity, behavior-based artifact collecting system that runs malware samples in virtualized environments and characterizes them by reporting memory, file system, registry, and network behavior. On the other hand, MaLabel uses artifacts generated by AutoMal to create features and then use them in classifying and clustering malware samples into families with similar characteristics. Both systems have been in production and helped analyze and identify hundreds of thousands of malware samples. AutoMal by design follows several guidelines in [54] for safety, and MaLabel follows several guidelines for data and and algorithms transparency, correctness, and realism.

- Based on an in-house product of AMAL, we use both medium

and large-scale datasets to show AMAL's effectiveness by demonstrating more than 99% of precision and recall in classification and more than 98% of precision and recall in clustering malware. Our validation makes use of several algorithms and settings and demonstrate the practicality of our system at scale, even when using off-the-shelf algorithms.

We emphasize that our work does not only systemize a literature knowledge—and demonstrate the power of off-the-shelf algorithms in malware classification at scale, but also augment this knowledge by several methodical and novel contributions. First, while a comparative study of various algorithms under various settings should be done in any applied machine learning to the security problem at hand, this was unfortunately not done in the prior literature. Our system addresses many timely problems highlighted in [54]. Our novel contribution is not only the reliance on multiple algorithms but highly accurate evaluation, multiple fine-grained features for multiple families characterization, the build of a system that extracts those features, and demonstrating its efficiency at scale. Our system for classification always matches accuracy of state-of-the-art [52], and improves it in many settings. Our clustering study shows the relevance and efficiency of off-the-shelf techniques; in [8] a scalable system takes 138 minutes with LSH optimization to cluster 75k samples, the largest literature dataset. We cluster 115k samples in under 1 hour without optimization (table 9) at the expense of additional memory.

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: A SYNOPSIS

There has been plenty of work in the recent literature on the use of machine learning algorithms for classifying malware samples [62, 7, 53, 45, 63, 52, 34, 51, 50, 14]. 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 [53, 45, 52, 68]. In [45], the authors use behavior graphs matching to identify and classify families of malware samples, at high cost of graph operations and generation. In [52, 53], 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). In the following, we dive into some of those related work, and refer to the reader to our complete work in [4] for a complete exposition and comparison of those works.

Dynamic analysis for malware classification: To the best of our knowledge, the closest work in the literature to this work is by Bailey et al in [7]. Similar to our work, the authors' goal is to use behavior characteristics to cluster malware samples. However, our work is different in three aspects. First, although we share similarity with their high level grouping of features, our system makes use of low granularity set of features, which expose richer behavior than theirs. Second, we try several clustering and classification algorithms, and demonstrate the performance-accuracy tradeoff of using these algorithms, whereas their work is limited to the hierarchical clustering with one distance measure. Finally, we use highly-accurate analyst-vetted labels for evaluation, where they use heuristics over AV-returned labels.

"Memory artifacts as features" research: Our system utilizes memory features for characterizing samples. Related to our use of memory features, Willems et al. introduced CWXDetector [66] which detects illegitimate code by analyzing memory sections that cause memory faults—artificially triggered by marking those section non-executable. The work can be integrated into our system, although at cost: the mechanism is intrusive to other running processes in the memory. Our current system, on the other hand, does not require any memory modifications. Kolbitsch et al. [35] introduce Inspector, which is used for automatically reverse engineering and highlighting codes responsible for "interesting" behaviors by malware. Related to that, Sharif et al. proposed to understand code-level behavior by reverse-engineering code emulators [55]. Those are examples among other works in the literature. However, all of those works do not generate malware artifacts other than memory-related signatures, which by themselves have limited insight into characterizing generic malware samples.

"Network artifacts as features" research: Related to our use of network features is the line of research on traffic analysis for malware and botnet detection, reported in [32, 22, 24, 26, 25] and for the particular families of malware that use fast flux, which is reported in [28, 43]. Related to our use of the DNS features for malware analysis are the works in [5, 6, 13]. 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.

Malware evasion research: Broadly related to our work are systems for overcoming malware evasion techniques. Improving on malware detection, analysis and classification have been investigated as well in several works in the literature. In [37], K-Tracer is introduced for extracting kernel malware behavior and mitigating the circumvention of loggers deployed in the kernel by rootkits. In [46], MacBoost is used for prioritizing malware samples by determining benign (or less severe) from malicious piece of codes. A system to prevent drive-by-malware based on behavior, named BLADE, is introduced in [40]. Finally, a nicely written survey on such systems and tools is in [19].

Scalability and improvement research (static analysis): There are many efforts in attempting to scale machine learning-based techniques for malware classification and analysis. However, unfortunately, those attempts all fall under the static analysis direction. For example, Jang et al [33] proposed BitShred for dimensionally reduction of *static analysis-based* features for fast and accurate clustering. A non-cryptographic hash is proposed in [65] for efficient representation of malware artifacts. Genome analysis techniques are suggested (but not well studied) in [16] for analyzing malware. Cluster ensembles over static features are used in [67] for highly accurate maleware clustering. In the same direction, several static filters and tools are proposed in the literature to speed up the detection of similar malware samples [31, 9, 23, 60, 39, 44, 61, 41, 17, 27, 36, 48, 49]. While many of the techniques claim to be scalable or novel, most of those works are applied on relatively small datasets that cannot bring insight into their scalability features. Furthermore, they rely on features driven from the static nature of binaries and suffer form the aforementioned limitations of static analysis techniques. Bayer et al. [8] also attempted to scale malware clustering using LSH, but they use low-level system calls that are easy to evade, and give less insight into various algorithms and their scalability features.

Machine learning for malware analysis and automation: Finally, the use of machine learning techniques to automate classi-

fication of behavior of codes and traffic are heavily studied in the literature. The reader can refer to recent surveys in [57] and [54].

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

Given the different purposes of the two sub-systems used in achieving our end goal, we build them as separate systems on separate operational platforms. In the following, we elaborate on the flow and functionalities of both sub-systems.

3.1.1 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:

- Sample priority queue: Allows samples to have processing priority based on submission source.
- Run time variable: Allows submitter to set run time for the sample in the virtual machine (VM) environment.
- Environment adjustment: Allows submitter to adjust operating system via script interface before running a sample.
- Multiply formats: Allows submission of formats like, EXE, DLL, PDF, DOC, XSL, PPT, HTML, and URL.
- VMware-based: Uses VMware as virtual environment.
- 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.

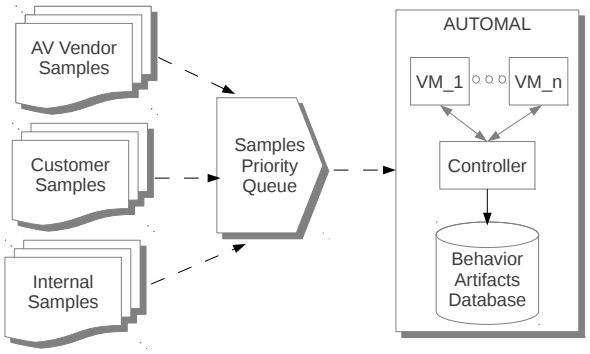

Figure 1: AutoMal flow diagram show multiple sources of malware samples, a controller used for assigning samples from the priority queue to virtual machines, and a back-end storage system used by the controller to log in behavior artifacts.

- Lower Privilege: Allows submitter to lower the OS privilege before running a sample. By default, samples run as a privileged user in Windows XP.
- 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 and flow of AutoMal is shown in Figure 1. AutoMal has 4 components, which are the 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.

```
rule silent_banker : banker
{
meta:
        description = "Just an example"
        thread_level = 3
        in_the_wild = true
strings:
        $a = {7B 50 79 11 41 11 11 7B 25}
        $b = {9E 5E C1 3C D2 94 D1 38 AA 7B}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
condition:
        $a or $b or $c
}
```

Figure 2: An example of a YARA signature

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:

- File system: files created, modified, and deleted, file content, and file meta data.
- Registry: registry created, modified, and deleted, registry content, and registry meta data.
- Network: DNS resolution, outgoing and incoming content and meta data.
- 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) [64] 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. The meta section describes the signature and contains author information. The string section defines the strings of interest and (or) a sequence of bytes. Finally, the condition section is a logical statement used to combine the different definitions in the string section to find a target family. Figure 2 shows an example of a typical YARA rule.

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

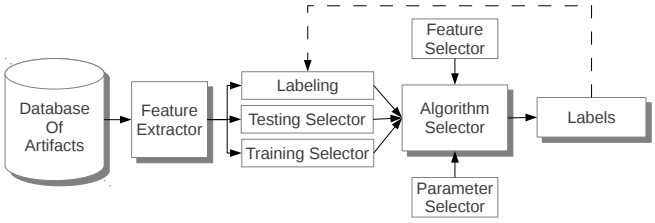

Figure 3: MaLabel's flow. Artifacts collected using AutoMal are fed into MaLabel and used to extract features. Labels are assigned using experts, as well as AV census, while features selection is done for each algorithm to achieve highest performance. Resulting labels from the learning algorithm are used to train future runs of the algorithm on newly fed testing data.

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.

3.1.2 *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,

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.

whether it is binary classification or clustering, the training (if applicable) and testing data into MaLabel is determined by the user (a flow of the process for classification is shown in Figure 3). 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.

While the clustering feature in MaLabel is not intended for labeling malware samples, but rather for grouping different samples that are similar in their behavior, we provide the system with the intelligence required for malware labeling. After the clustering algorithm runs, we enable one of the following options to label the data. First, using analyst-vetted samples we augment the resulting clusters with labels and extrapolate the labels on unlabeled samples falling with the same clusters. Second, for those clusters that we do not have significant analyst-vetted data, we make use of memory signatures, where available, and further manual inspections. Finally, while we try to avoid that as much as possible for the known inconsistencies of their labeling systems, in cases where none of the two options above are viable we use census over labels of several antivirus scans for clusters' members—we use 42 independent antivirus vendors; for further details see §4.

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 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.

For the safety of potential victims of the malware samples we run in AutoMal, we use several safety guidelines. First, we block a list of wormable ports, including port 25, 587, 445, 139, and 137 at the router level. We do this blocking although we believe it will limit visibility to crucial features, although we log the outgoing requests. We further add safety by limiting the run time of samples, and throttling bandwidth, to avoid sizable damage.

Although we do not utilize memory features in the evaluation of MaLabel, the current in-production AutoMal collects and archive memory artifacts. However, this functionality was implemented in the system after many samples used in this study were already analyzed and their artifacts are archived. Thus, it is impossible to create memory features for those samples without rerunning them in the system. However, since those samples are collected over a period of time, even if we try to rerun them we will miss some other features that are time-dependent. For example, a command and control server could be taken down since the last time the sample was analyzed in AutoMal, thus rerunning the samples allows us to get memory features but would make us loose the network artifacts. As we are collecting memory signatures that can be used to derive memory features for newly fed samples, it is left for future work to systematically see how those features will influence the clustering and classification—preliminary small-scale results are promising.

Features normalization. Following the related literature [29, 8],

Table 2: Benchmarking of hardware components used for the different parts of our system. MaLabel 1 and MaLabel 2 are platforms used for clustering and classification, respectively. * Win XP is default, and others are used where needed.

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

we map the different features' values in the range of 0 and 1, thus not biasing the feature selection process towards any feature except of its true importance. Fortunately, all of the features we use in MaLabel are normalizable.

4. EVALUATION

To evaluate the different algorithms in each application group, we use several accuracy measures to highlight the performance of various algorithms. Considering a class of interest, $\mathcal{S}$, the *true positive* (t_p) for classification is defined as all samples in $\mathcal{S}$ that are labeled correctly, while the *true negative* (t_n) is all samples that are correctly rejected. The *false positive* (f_p) is defined as all samples that are labeled in $\mathcal{S}$ while they are not, whereas the *false negative* (f_n) is all samples that are rejected while they belong to $\mathcal{S}$. For validating the performance of the classifiers, we use the precision defined as $P = t_p/(t_p + f_p)$, the recall as $R = t_p/(t_p + f_n)$, the accuracy as $A = (t_p + t_n)/(f_p + f_n + t_p + t_n)$, and the F-score defined as $F = 2(p \times r)/(p + r)$.

For clustering, we use the same definition of accuracy, precision, and recall as in [8]. In short, the precision measures the ability of the clustering algorithm to distinguish between different samples and associate them to different clusters, whereas the recall measures how well the clustering algorithm assigns samples of the same type to the same cluster. To that end, given a reference (ground truth) clustering setting $T = \{T_i\}$ for $0 < i \leq n_t$ and a set of learned clusters $L = \{L_i\}$ for $0 < i \leq n_l$, the precision for the $j-$th learned cluster is computed as $P_j = \max\{|L_j \cap T_i|\}$ for $0 < i \leq n_l$ while the recall for the j-th reference cluster is computed as $R_j = \max\{|L_i \cap T_j|\}$ for $0 < i \leq n_l$. The total precision and recall of the algorithm are computed as $\frac{1}{n_t}\sum_{i=1}^{n_t} P_i$ and $\frac{1}{n_l}\sum_{i=1}^{n_l} R_i$, respectively.

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 (cf. §4.5.2), 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 of results 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

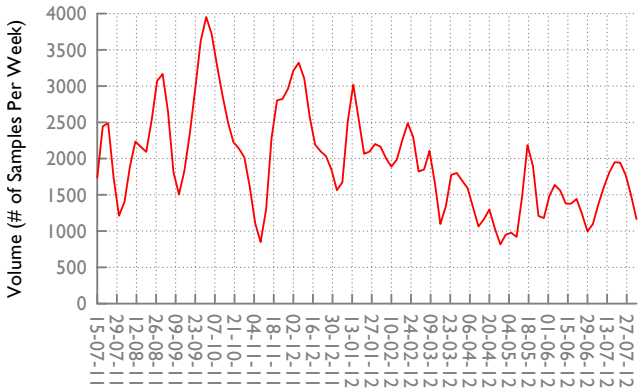

Figure 4: The time samples used in this study are populated in AutoMal—Samples accumulated over 13 months. Notice that the used samples are sampled from larger set of samples accumulated over the same period of time.

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 time those samples were populated in the system is shown in Figure 4. 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.

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.

Note: In the process of conducting this study, we referred to the recommendations in [54] for the transparency, correctness, and realism of this work. For reproducibility of results, we intend to release part the dataset used in this study to the public domain.

4.3 High-fidelity Malware Classification

In this section, we focus on the binary classification problem using the Zeus malware family [20], 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. Unless otherwise is specified, in all evaluations we use 10-fold cross validation—a formal definition and settings are provided in Appendix B.

Table 3: Malware samples, and their corresponding 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	

4.3.1 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 Appendix A. 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 of those settings are in Appendix A)

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.

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 Poly. 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%

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 [52].

To understand causes for the relatively high false alarms (causing part of the degradation in precision and recall) with some of the algorithms we tried, we looked into mislabeled Zeus and non-Zeus malware samples. We noticed that distance in the feature vector between misclassified samples is far from the majority of other samples within the class. This is however understandable, given that a single class of malware (Zeus and non-Zeus) includes within itself multiple sub-classes that the high-level label would sometimes miss. This observation is further highlighted in the clustering application, where those mislabeled samples are grouped in the same group, representing their own sub-class of samples.

4.3.2 Features Ranking and Selection

While the number of features used in MaLabel is relatively small when compared to other related systems [5, 6], not all features are equally important for distinguishing a certain malware family. Accordingly, this number can be perhaps greatly reduced while not affecting the accuracy of the classification algorithms. The reduction in the number of samples can be a crucial factor in reducing the cost of running the classification algorithm on large-scale datasets.

In order to understand the relative importance of each feature, with respect to the (linear) classification algorithms, we ran the recursive feature elimination (RFE) algorithm [3], which ranks all features from the most important to the least important feature. Given a set of weights of features, the RFE selects the set of features to prune recursively (from the least to the most important) until reaching the optimal number of features to achieve the best performance. In the linear classification algorithms, weights used for ranking features are the coefficients in the prediction model associated with each feature (variable).

Table 5 shows the performance measures for the SVM using different numbers of features. We notice that, while the best performance is achieved at the largest number of features, indicating the importance of all features together, the improvement in the performance is very small, particularly for the SVM. The lowest 50 features in rank improve the accuracy, precision, and recall by less than 2%. However, this improvement is as high as 20% with decision trees (results not shown and deferred to [4] for the lack of space). To this end, and for the algorithm of choice (SVM), we confirm that a minimal set of features can be used to achieve a high accuracy while maintaining efficiency.

Table 5: The accuracy measures versus the number of features used for classification (SVM with L2R and L2L).

Features	A	P	R	F
3	65.3%	66.9%	60.5%	63.6%
6	73.2%	76.1%	67.6%	71.6%
9	89.6%	87.6%	92.3%	89.9%
15	94.1%	94.0%	94.1%	94.1%
25	94.4%	94.9%	93.9%	94.4%
35	94.6%	95.3%	93.8%	94.6%
45	94.9%	95.6%	94.0%	94.8%
65	95.6%	95.8%	95.3%	95.5%

We also followed the recent literature [29, 13, 11] 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 [4].

4.3.3 Choosing Classification Parameters

Our system does not only feature several algorithms, but also uses several parameters for the same algorithm. For example, regularization and loss functions are widely used to improve estimating unknown variables in linear classification. For that, regularization imposes penalty for complexity and reduces over-fitting, while loss function penalizes incorrect classification. Widely used function types of parameters for linear classification are the L1 and L2 functions (more details on both types of functions are summarized in Appendix 1 and discussed in details in [21]). In addition, since linear classification or programming problems can be stated as primal problems, they can also converted to dual problems, which try to solve the primal problem by providing an upper bound on the optimal solution for the original (primal) problem. In the following we test how the choice of the proper set of parameters—problem representation into primal or dual and the choice of regularization and loss functions—affects classification by considering SVM and LR as two examples with a select set of parameters. We use the same dataset as above in this experiment as well.

The results of this measurement are shown in Table 6. We observe that while all algorithm perform reasonably well on all measures of performance (namely, above 90% for all measures, for most of them), and can be used as a building block for MaLabel, the L2-regularization L2-loss functions, when combined with the dual optimization representation, provides the highest performance with all accuracy measures above 95%. All algorithms do not use kernel methods, and are very fast to run even on large datasets.

Table 6: Selection of the support vector classifier with the best performing parameters. A, P, R, and F correspond to the accuracy, precision, recall, and F-score, respectively.

Algorithm	A	P	R	F
L1-reg. log. regression (/)	93.7%	93.7%	93.7%	93.7%
L2-reg. log. regression (p)	92.3%	91.4%	93.4%	92.4%
L2-reg. L2-loss SVM (d)	**95.6%**	**95.8%**	**95.3%**	**95.5%**
L2-reg. L2-loss SVM (p)	89.1%	84.5%	95.7%	89.7%
L2-reg. L1-loss SVM (d)	94.1%	95.6%	92.5%	94.0%
L1-reg. L2-loss SVM (/)	94.0%	94.0%	94.0%	94.0%
L2-reg. log. regression (d)	94.3%	94.5%	94.1%	94.3%

Table 7: 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%

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. We use 10-fold cross validation to minimize bias. In the following we summarize the results of seven families of interest. The results are shown in Table 7.

First of all, we notice that although the performance measures are less than those reported for Zeus in section 4.3.1, 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 [52] 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 [52] 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.

4.5 Malware Clustering

One of the limitations of the prior literature on malware samples clustering is that it did not try different algorithms. For example, while the hierarchical clustering has the same overall procedure for producing clustering (details are in Appendix 1), altering the cutting parameter, distance metric or linkage criteria would greatly

influence the shape of the final cluster. For our application, this would mean different resulting clusters for different parameters. MaLabel employes several distance metrics, like the Jaccard index, cosine similarity, hamming distance, Euclidean distance, and correlation. On the other hand, options for linkage include average, complete (maximum), median, single (minimum), ward, and centroid, among others. In this part, we evaluate the clustering part of MaLabel, with different parameters and settings, and report on some of the relevant results and findings.

4.5.1 *Clustering of Manually Labeled Samples*

Using the various options listed earlier, MaLabel gives the choice to the user to pick the best clustering size based on the user's data and, when available, the performance measures. Multiple cut threshold are calculated for each distance and method to give an overview of how each clustering method performed. The user then makes a judgement to choose the most relevant results to fit the data to.

To evaluate the performance of the clustering, we use the manually labeled Zeus family. We further use tags in the manual labeling that divide the Zeus family into multiple sub-families (also called Zeus variations), and consider that as a reference (or ground-truth) clustering. To add variety of families to the problem, and challenging the clustering algorithms, we also picked an equal number of samples, from the families shown in Table 3. However, this time, we limit our selection to samples for which we already know a correct label. We ran our manually labeled malware data set against the clustering algorithms and evaluated the performance using the precision and recall defined earlier.

Table 8 shows the precision, recall, and cutting threshold for several distance metrics. First of all, we notice that one can achieve high performance using easy-to-set parameters. While one can brute-force the cutting parameter range to achieve the highest possible accuracy [8], this option might not be always available with partially labeled data. Second, and most important, we notice that the achieved precision and recall outperform the classification algorithms evaluated in section 4.3. This in part is attributed to the power of the clustering algorithm in distinguishing subclasses into distinct clusters, whereas subclasses in the binary classification that are close to each other in the feature vector space are grouped erroneously with each others. To this end, one may actually use the results of the clustering to guide the binary classification, and to reduce its false alarms, thus improving its performance. We leave the realization of this feature for future work.

4.5.2 *Large Scale Clustering*

We ran 115,157 samples of malware (the total of samples in Table 3) through our clustering system to identify two main aspects. First, we wanted to know how would those samples of malware cluster among each other and how large would each cluster be given that we have a very large sample set. Second, we wanted to identify the time required for a large set of samples to cluster and to identify the required hardware for such process in an operational settings. We tested the 115,157 sample on a machine with 120 GB of RAM, four core processor, and 200 GB hard disk—as shown in Table 2. We were able to process 175,000 samples on the same machine in a very reasonable time.

4.5.3 *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. The timing results are shown in Table 9. 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 [8]) takes 5820 seconds (about 97 minutes) whereas all other distance measures require between 27.8 minutes to 36.2 minutes. By considering other evaluation criteria, like timing of the linkage cost and the performance measures, one can make several interesting conclusions. For example, given that the Hamming and Jaccard index for distance computation perform equally well for clustering, as shown in Table 8, one can use the Hamming distance and save up to 70% of the total time required for clustering the same dataset. Those results show that it is highly visible to perform large-scale clustering, even using off-the-shelf algorithms with settings that are overlooked in the literature. Finally, we measured the time it takes to extract $115,157$ samples' features from artifacts. We do that over network to measure a distributed operation of our system. We found the total time taken is 222 minutes (116 millisecond per sample), which is a reasonable time that supports larger scales of deployment.

Table 8: Clustering precision and recall for several linkage and cutting criteria and parameter values.

	Linkage	Cutting	Precision	Recall
Correlation	Average	0.40	93.4%	100%
	Centroid	0.25	96.2%	100%
	Complete	0.70	89.7%	100%
	Median	0.25	89.6%	96.6%
	Single	0.40	90.2%	100%
	Ward	0.25	93.5%	98.2%
Cosine	Average	0.25	84.1%	100%
	Centroid	0.25	84.6%	100%
	Complete	0.40	85.5%	97.1%
	Median	0.25	94.4%	95.2%
	Single	0.40	91.2%	100%
	Ward	0.25	94.2%	96.9%
Hamming	Average	0.25	98.9%	97.6%
	Centroid	0.25	98.5%	100%
	Complete	0.25	98.7%	97.5%
	Median	0.25	100%	100%
	Single	0.25	98.3%	98.8%
	Ward	0.25	99.3%	97.6%
Jaccard	Average	0.25	99.9%	100%
	Centroid	0.25	99.9%	100%
	Complete	0.25	99.9%	100%
	Median	0.40	99.9%	99.8%
	Single	0.25	99.9%	100%
	Ward	0.40	99.9%	100%

4.6 Limitations

Like many techniques that rely on machine learning for classification and behavior for characterization, our technique has several shortcomings and limitations. First of all, since we run the malware samples in a virtualized environment, some of the malware samples may not run by detecting that fact, or even run but generate irrelevant behavior profile. For samples that do not run, which are mostly due to detecting that the sample works in a virtualized analysis environment, or because some environment variables are missing, we have two options that we implemented and turned successful in addressing the issue. First, we modify the VM environment using the AutoMal configuration features to meet the expected environment requirement for the sample to run resulting in evasion. Second, if that fails, we sometimes do manual debugging and reversing, and

Table 9: Timing (seconds) for a benchmark of 115,157 samples using different distance measures: Correlation (Co), Ward (W), Single (S), Average (A), Median (M), and Centroid (Ce). DM is the time for computing the pair-wise distance matrix.

Method	DM	Co	W	S	A	M	Ce
Correlation	1612	1436	1614	1396	1458	4954	5013
Cosine	1572	1429	1599	1390	1453	5323	5365
Hamming	2177	1460	1683	1412	1492	1453	1575
Jaccard	5820	1427	1686	1450	1449	1445	1604

occasionally run the sample on bare-metal (on a non-virtualized system). An alternative to that is to use hardware virtualization, as in Ether [18], which still supports the scalability features provided by virtualized execution while addressing the problem.

According to our operations, and based on our data used in this study, only less than 10% of the malware samples we collect in our system have this issue, contrary to recent studies showing that 80% of malware samples deploy anti-VM techniques [15]—In this latter particular study, the authors analyze only 883 of "cherry-picked" samples to support their statement. The sample is very small, and a generalization on malware populations is hard to establish based on that. Furthermore, the overwhelming majority of the less than 10% samples in our system (more than 98%) are made to run using at least one of the aforementioned techniques.

Another theoretical limitation of our technique is that it relies on behavior features that a malware sample can try to evade by generating misleading artifacts to poison those features. Indeed, adversarial machine learning has recently received a lot of attention in studying strategies an adversary can use to mislead the classifier [10]. However, with our current dataset, we noticed that the features used for classification and clustering are robust to minor changes imposed by the adversary. Furthermore, once the adversary becomes aware of the features used for classification and a way to misguide our models, it becomes an open problem to always update the features and select distinctive properties of the malware sample that can resist such attacks. We emphasize that this kind of attack pertains to theory, more than to practice.

A final limitation of our work is an inherit scalability bottleneck associated with sandboxing systems: each malware sample needs to be executed in a VM to extract representative features. While this could have been an issue five 10 or even 5 years ago, when virtualization technologies were not a mature domain, it is nowadays an easily solvable problem. When our project, AMAL, started four years ago it consisted of 4 virtual machines and a single controller that can process a malware sample every 2 minutes at average. Today, both vertical and horizontal scalability are possible. Vertically, the system has virtually unlimited resources, and can process 128 malware samples simultaneously. Adding resources and scaling the system is very simple, and cost associated with the scalability is very small. Horizontally, optimizations in determining a subset of features that are of interest to the analyzed malware families and limiting the sandboxing to them (as discussed in the context of features selection) would greatly reduce the time taken in analyzing the malware samples and collecting artifacts from them.

5. CONCLUSION AND FUTURE WORK

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.

Using the artifacts generated by AutoMal, MaLabel creates feature vectors and use them to 1) perform binary classification of malware samples into distinctive, but generic families, and 2) cluster malware samples into different families. Those families can be at the granularity of a subclass (variant). By evaluating AutoMal on a large corpus of analyst-vetted malware samples, we conclude its many features and advantages. It provides high levels of precision, recall, and accuracy, for both clustering and classification. Furthermore, it provides the operator with the key feature of choosing among several equally performing algorithms to do the same task (e.g., clustering) and at low cost.

Beside memory features that we did not examine systematically as features–beyond using the YARA signatures in data filtering, we avoided the use of implicit features, side-channel information on the reputation of network-related features (e.g., IP reputation, and passive and reverse DNS information, among others), and white- and blacklists—many of which are already available for improving AMAL. In the future, we will look at utilizing those as additional features to our system, and study how they impact its performance.

6. 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] E. Alpaydin. *Introduction to machine learning*. MIT press, 2004.

[4] Anonymized for Review. High-fidelity, behavior-based automated malware analysis and classification. Technical report, Anonymized for Review, 2013.

[5] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a dynamic reputation system for dns. In *USENIX Security Symposium*, 2010.

[6] 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.

[7] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *RAID*, 2007.

[8] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[9] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Abstraction-based malware analysis using rewriting and model checking. In *Computer Security–ESORICS 2012*, pages 806–823. Springer, 2012.

[10] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[11] 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.

[12] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM CCS*, 2012.

[13] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *NDSS*, 2011.

[14] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust*, 2010.

[15] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Blackhat, 2012.

[16] E. Carrera and G. Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Virus bulletin conference*, volume 11, 2004.

[17] P. M. Comar, L. Liu, S. Saha, P.-N. Tan, and A. Nucci. Combining supervised and unsupervised learning for zero-day malware detection. In *INFOCOM, 2013 Proceedings IEEE*, pages 2022–2030. IEEE, 2013.

[18] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM CCS*, 2008.

[19] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on

automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, Mar. 2008.

[20] N. Falliere and E. Chien. Zeus: King of the Bots. Symantec Security Response (`http://bit.ly/3VyFV1`), November 2009.

[21] 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.

[22] C. Gorecki, F. C. Freiling, M. Kührer, and T. Holz. Trumanbox: Improving dynamic malware analysis by emulating the internet. In *SSS*, 2011.

[23] K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, pages 101–120. Springer, 2009.

[24] 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.

[25] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security Symposium*, 2007.

[26] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *NDSS*, 2008.

[27] Z. Hanif, T. Calhoun, and J. Trost. Binarypig: Scalable static binary analysis over hadoop. `http://bit.ly/17ykNdW`, 2013.

[28] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling. Measuring and detecting fast-flux service networks. In *NDSS*, 2008.

[29] C.-Y. Hong, F. Yu, and Y. Xie. Populated ip addresses: classification and applications. In *ACM CCS*, pages 329–340, 2012.

[30] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear svm. In *ICML*, 2008.

[31] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer, 2013.

[32] G. Jacob, R. Hund, C. Kruegel, and T. Holz. Jackstraws: Picking command and control connections from bot traffic. In *USENIX Security Symposium*, 2011.

[33] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011.

[34] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[35] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *IEEE Sec. and Privacy*, 2010.

[36] J. Kwon and H. Lee. Bingraph: Discovering mutant malware using hierarchical semantic signatures. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 104–111. IEEE, 2012.

[37] A. Lanzi, M. I. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *NDSS*, 2009.

[38] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

[39] B. Liang, W. You, W. Shi, and Z. Liang. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 217–227. ACM, 2011.

[40] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *ACM CCS*, pages 440–450, 2010.

[41] H. D. Macedo and T. Touili. Mining malware specifications through static reachability analysis. In *ESORICS*, pages 517–535. Springer, 2013.

[42] A. Mohaisen, O. Alrawi, M. Larson, and D. McPherson. Towards a methodical evaluation of antivirus scans and labels. In *The 14th International Workshop on Information Security Applications (WISA2013)*. Springer, 2013.

[43] J. Nazario and T. Holz. As the net churns: Fast-flux botnet observations. In *MALWARE*, pages 24–31, 2008.

[44] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. Forecast: skimming off the malware cream. In *Proceeding of ACSAC*, pages 11–20. ACM, 2011.

[45] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *CSIIR Workshop*. ACM, 2010.

[46] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *ACSAC*, 2008.

[47] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *USENIX NSDI*, 2010.

[48] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. Bay, et al. Malware analysis and attribution using genetic information. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 39–45. IEEE, 2012.

[49] J. Pfoh, C. Schneider, and C. Eckert. Leveraging string kernels for malware detection. In *Network and System Security*, pages 206–219. Springer, 2013.

[50] 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.

[51] M. Ramilli and M. Bishop. Multi-stage delivery of malware. In *MALWARE*, 2010.

[52] 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.

[53] 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.

[54] 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.

[55] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *IEEE Sec. and Privacy*, 2009.

[56] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[57] 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.

[58] R. Strackx and F. Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *ACM CCS*, 2012.

[59] W. T. Strayer, D. E. Lapsley, R. Walsh, and C. Livadas. Botnet detection based on network behavior. In *Botnet Detection*, 2008.

[60] G. Tahan, C. Glezer, Y. Elovici, and L. Rokach. Auto-sign: an automatic signature generator for high-speed malware filtering devices. *Journal in computer virology*, 6(2):91–103, 2010.

[61] G. Tahan, L. Rokach, and Y. Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *The Journal of Machine Learning Research*, 98888:949–979, 2012.

[62] R. Tian, L. Batten, R. Islam, and S. Versteeg. An automated classification system based on the strings of trojan and virus families. In *IEEE MALWARE*, 2009.

[63] R. Tian, L. Batten, and S. Versteeg. Function length as a tool for malware classification. In *IEEE MALWARE*, 2008.

[64] VMWare. Virtual Machine Disk Format (VMDK). `http://bit.ly/e1zJkZ`, May 2013.

[65] G. Wicherski. pehash: A novel approach to fast malware clustering. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[66] C. Willems, F. C. Freiling, and T. Holz. Using memory management to detect and extract illegitimate code for malware analysis. In *ACSAC*, 2012.

[67] Y. Ye, T. Li, Y. Chen, and Q. Jiang. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–104. ACM, 2010.

[68] H. Zhao, M. Xu, N. Zheng, J. Yao, and Q. Ho. Malicious executables classification based on behavioral factor analysis. In *IC4E*, 2010.

APPENDIX

A. MACHINE LEARNING ALGORITHMS

We outline definitions and algorithms used in MaLabel. For optimizations used in MaLabel, see [21] and [4].

A.1 Classification Algorithms

A.1.1 Support Vector Machines (SVM)

Given a training set of labeled pairs $(\mathbf{x}_i, y_i)$ for $0 < i \leq \ell$, $\mathbf{x}_i \in R^n$, and $y_i \in \{1, -1\}$, the (L2R primal) SVM solces:

$$\min_{\mathbf{w},b,\xi} \ \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{\ell}\xi_i \tag{1}$$

$$\text{subject to} \quad y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \tag{2}$$

$$\xi_i \geq 0 \tag{3}$$

where the training vectors $\mathbf{x}_i$ are mapped into a higher dimensional space using the function ϕ, and the SVM finds a linear separating hyperplane with the maximal margin in this space. $C > 0$ is the penalty parameter of the error term (set to 0.01 in our work).

Kernel functions. $K(\mathbf{x}_i, \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$ is the kernel function. In MaLabel (§4) we use the linear and polynomial kernels, defined as $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$ and $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma\mathbf{x}_i^T\mathbf{x}_j + r)^d, \gamma > 0$, respectively. In §4 we use $\gamma = 64$ and $d = 2$.

Loss functions. $\xi(\mathbf{w}, \mathbf{x}, y_i)$ is called the *loss* function, where we use the L1-loss and L2-loss, defined as

$$\xi(\mathbf{w}, \mathbf{x}, y_i) = \max(1 - y_i\mathbf{w}^T\mathbf{x}_i, 0), \text{ and} \tag{4}$$

$$\xi(\mathbf{w}, \mathbf{x}, y_i) = \max(1 - y_i\mathbf{w}^T\mathbf{x}_i, 0)^2. \tag{5}$$

Regularization. The formalization in (1) is called the L2-regularization. The L1-regularized L2-loss SVM solves the following primal problem (where $||\cdot||_1$ is the L1-norm):

$$\min_{\mathbf{w}} \ ||\mathbf{w}||_1 + C\sum_{i=1}^{\ell}\max(1 - y_i\mathbf{w}^T\mathbf{x}_i, 0)^2 \tag{6}$$

Dual SVM. The dual SVM problem [30] is defined as:

$$\min_{\alpha} f(\alpha) = \frac{1}{2}\alpha^T\bar{Q}\alpha - \mathbf{e}^T\alpha \tag{7}$$

subject to $0 \leq \alpha_i \leq U$ for $1 \leq i \leq \ell$, where $\bar{Q} = Q + D$, D is a diagonal matrix, and $Q_{ij} = y_iy_j\mathbf{x}_i^T\mathbf{x}_j$. For L1-Loss SVM, $U = C$ and $D_{ii} = 0$ for $1 \leq i \leq \ell$, while for L2-Loss SVM, $U = \infty$ and $D_{ii} = \frac{1}{2C}$ for $1 \leq i \leq \ell$.

A.1.2 Logistic Regression (LR)

Here we provide definitions for the logistic regression algorithms used in §4. The L2-regularized primal LR solves the following unconstrained optimization problem:

$$\min_{\mathbf{w}} \ \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{\ell}\log(1 + e^{-y_i\mathbf{w}^T\mathbf{x}_i}) \tag{8}$$

and the dual version of the problem (L2R LR dual) is

$$\min_{\alpha} \frac{1}{2}\alpha^TQ\alpha + \sum_{i:\alpha_i>0}\alpha_i\log\alpha_i + \sum_{i:\alpha_i<C}(C - \alpha_i) - \sum_{i=1}^{\ell}C\log C \tag{9}$$

subject to $0 \leq \alpha_i \leq C$ where $1 \leq i \leq \ell$.

On the other hand, the L1 regularized LR solves

$$\min_{\mathbf{w}} ||w||_1 + C\sum_{i=1}^{\ell}\log(1 + e^{-y_i\mathbf{w}^T\mathbf{x}_i}). \tag{10}$$

A.1.3 Perceptron

The perceptron is an artificial neural network (ANN) that maps its input $\mathbf{x}$ into a binary value (class label) as follows:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{If } \mathbf{w}\cdot\mathbf{x} + b > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{11}$$

where $\mathbf{w}$ is a weights vector, and b is a bias parameter. We defer detailed explanation of the learning and testing method to our technical report for the lack of space [4]. In §4, we use a maximum network depth of 100, a b of 0.05, and a learning rate α of 0.1. We do not concern ourselves by finding the optimal parameters.

A.1.4 K-Nearest-Neighbor

The KNN is a non-linear classification algorithm. In the training phase, we provide the algorithm of two labels and a set of training samples. In the testing phase, for each sample vector $\mathbf{a}$, we give it the label of the most frequent among the training samples nearest to it. For the lack of space, we defer details to [4]—the same technique is well explained in [3]. For evaluation, we use an odd k (to break potential ties), and vote the class to which $\mathbf{a}$ belongs.

A.1.5 Decision Trees

For the lack of space, we defer most of the details on the classification trees used in this study to [4] for the lack of space. However, we use the standard classification tree described in [3], and using a single split tree (for two classes classification). For that, we utilize all the set of features provided in the study (the same technique with its optimizations is used in [13]).

A.2 Hierarchical Clustering

We use the classical hierarchical clustering algorithm in [3].We defer the description of the algorithm to [4], and here only focus on the mathematical description of the different settings used in §4.

A.2.1 Distance metrics

In §4, we use several distance metrics, defined as in (12)-(15)

$$\text{Jaccard:} \quad d(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \vee \mathbf{b}}{\mathbf{a} \wedge \mathbf{b}} \tag{12}$$

$$\text{Cosine:} \quad d(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}\cdot\mathbf{b}}{||\mathbf{a}||||\mathbf{b}||} \tag{13}$$

$$\text{Hamming:} \quad d(\mathbf{a}, \mathbf{b}) = \sum_i |a_i - b_i| \tag{14}$$

$$\text{Correlation:} \quad d(\mathbf{a}, \mathbf{b}) = \frac{\frac{1}{n}\sum_i a_ib_i - \mu_\mathbf{a}\mu_\mathbf{b}}{\sigma_\mathbf{a}\sigma_\mathbf{b}} \tag{15}$$

In the definitions above, $\mu_\mathbf{a} = \frac{1}{|\mathbf{a}|}\sum_{a\in\mathbf{a}} a$ and $\mu_\mathbf{b} = \frac{1}{|\mathbf{b}|}\sum_{b\in\mathbf{b}} b$.

A.2.2 Linkage metrics

Let A and B be clusters, and $\mathbf{a}$ and $\mathbf{b}$ be arbitrary samples (vec-

tors) in them. We define the following linkage metrics.

$$\text{Average:} \quad d(A,B) = \frac{1}{||A||||B||} \sum_{\mathbf{a}\in A} \sum_{\mathbf{b}\in B} d(\mathbf{a},\mathbf{b}) \tag{16}$$

$$\text{Centroid:} \quad d(A,B) = ||c_A - c_B||_2, c_A = \frac{1}{|A|} \sum_{i=1}^{|A|} a_{ki}. \tag{17}$$

$$\text{Complete:} \quad d(A,B) = \max_{\mathbf{a}\in A,\mathbf{b}\in B} d(\mathbf{a},\mathbf{b}) \tag{18}$$

$$\text{Median:} \quad d(A,B) = ||\tilde{c_A} - \tilde{c_B}||_2, \tilde{c_A} = \frac{1}{2}(c'_A + c''_A) \tag{19}$$

$$\text{Single:} \quad d(A,B) = \min_{\mathbf{a}\in A,\mathbf{b}\in B} d(\mathbf{a},\mathbf{b}) \tag{20}$$

$$\text{Ward:} \quad d(A,B) = \frac{|A||B|}{|A|+|B|} ||c_A - c_B||_2 \tag{21}$$

In (17) and (19), $||\cdot||_2$ is the Euclidean distance defined as $d(a,b) = \sum_{\forall i}(a_i - b_i)^2$. In (19), $\tilde{c_B} = \frac{1}{2}(c'_B + c''_B)$ for recursively linked clusters B' and B'' in B.

B. K-FOLD CROSS VALIDATION

We use the classical k-fold cross validation (where $k = 10$) for testing our models. We divide the dataset D into $D_1, D_2, \ldots, D_k$. We spare D_i for $1 \leq i \leq k$ for testing, and use the remaining $k-1$ folds for training. We repeat the process k times by altering the testing and training datasets. As a sanity check, we rerun the test several times by re-randomizing the order of samples in D.