MalFamAware: Automatic Family Identification and Malware Classification Through Online Clustering

Gregorio Pitolli · Giuseppe Laurenza · Leonardo Aniello · Leonardo Querzoni · Roberto Baldoni

the date of receipt and acceptance should be inserted later

Abstract The skyrocketing grow rate of new malware brings novel challenges to protect computers and networks. Discerning truly novel malware from variants of known samples is a way to keep pace with this trend. This can be done by grouping known malware in families by similarity and classifying new samples into those families. As malware and their families evolve over time, approaches based on classifiers trained on a fixed ground truth are not suitable. Other techniques use clustering to identify families but they need to periodically recluster the whole set of samples, which does not scale well. A promising approach is based on incremental clustering, where periodically only yet unknown samples are clustered to identify new families, and classifiers are re-trained accordingly. However, the latter solutions usually are not able to immediately react and identify new malware families. In this paper we propose MalFamAware, a novel approach to malware family identification based on an online clustering algorithm, namely BIRCH, which efficiently updates clusters as new samples are fed without requiring to re-scan the entire dataset. MalFamAware is able to both classify new malware in existing families and identify new families at runtime. We present experimental evaluations where MalFamAware outperforms both total reclustering and incremental clustering solutions in terms of accuracy and time. We also compare our solution with classifiers re-trained over time, obtaining better accuracy, in particular when samples belong to yet unknown families.

Keywords malware analysis · malware family identification · incremental clustering · BIRCH

G. Pitolli · G. Laurenza · L. Querzoni · R. Baldoni
Research Center of Cyber Intelligence and Information Security, Sapienza University of Rome, Italy
E-mail: {laurenza,querzoni,baldoni}@diag.uniroma1.it

L. Aniello
Cyber Security Research Group, University of Southampton, UK
E-mail: l.aniello@soton.ac.uk

1 Introduction

Over the last few years there has been an outbreak of cyber-attacks against computers and networks. The leading tools employed by cyber-criminal are *malware*, malicious software crafted to deliver specific attacks and evade existing security measures. Cyber threats keep evolving relentlessly in response to the corresponding progress of security defenses, resulting in an impressive number of new malware that are being discovered daily, in the order of millions[1]. The field of *malware analysis* is advancing in response to this escalation of malware-driven cyber threats. Although the primary objective is understanding whether an executable is malicious or not, this cannot be achieved without developing a very precise and updated knowledge on malware characteristics. Thus, there is also the need for an up-to-date *malware knowledge base* where insights can be extracted from to design new effective countermeasures.

Reverse engineering a malicious sample can add truly valuable information to such malware knowledge base, but it is an undoubtedly time-consuming task. Given the huge amount of new malware appearing every day, manually analysing in details each of them is obviously unfeasible. To this end, advanced malware analysis workflows commonly adopt a *triage stage* [1,2] where samples are quickly and automatically inspected to deter-

[1] `blog.trendmicro.com/malware-1-million-new-threats-emerging-daily`

mine if they need to be further investigated manually by an analyst.

A typical approach is relying on the concept of *malware families*, where samples are grouped by similarity in such a way that malware of a same family share large portions of code or display very similar behaviours, i.e. they are *variants* of each other. Such concept can be used to save a huge amount of time of work for malware analysts; indeed, only some samples for each family need to be analyzed manually. *Malware classification*, i.e. the process of assigning new malware to known and previously studied families, helps analysts to focus only on those malware belonging to groups that have not been analyzed yet. Thus, accurately characterizing families is crucial to develop an effective malware knowledge base; a flawed model would, in fact, lead to wrong triage decisions, which in turn would be detrimental to the whole malware analysis workflow.

The challenge to address in this case is keeping the malware family knowledge base as updated as possible, which requires revising it on the basis of the latest samples analyzed to discover new families and amend those already known. In this paper, we refer to this process as *family identification*, which should be tightly integrated with malware classification to properly tackle the fast evolution of malware. Indeed, an ideal malware analysis mechanism for triage should provide at the same time classification of samples in families and identification of new families as they arise.

There exist several approaches for automatic malware classification that are based on classifiers trained with a fixed ground truth of families [3,4]. Although they are highly accurate, they fail in family identification as they can not recognize new families unless they are frequently re-trained on an updated and reliable ground truth, which is commonly not available. This problem is often mitigated by adopting machine learning unsupervised methods. In particular, *clustering algorithms* are used to group samples by similarity on the basis of some representative feature vector. In this case, families correspond to the clusters identified by the algorithm. However, in order to recognize new families, the whole dataset of available malware needs to be clustered from scratch (*re-clustering*) whenever new samples have to be analyzed. This is the main limitation of this approach because the clustering time grows at least linearly with the size of the dataset, hence the current pace of new malware generation makes this approach impractical.

Such scalability issue can be mitigated by using an *incremental clustering* approach, like the one applied by Malheur [5]. A classifier is periodically trained with the families identified so far through clustering. This classifier is used to analyse new samples and determine what known families they belong to. Only malware that have not been classified confidently enough to any known family are then clustered to identify new families and incrementally improve the current knowledge. The classifier is then retrained on this updated knowledge. The limitation of this approach lies in the difficulty of choosing the correct batch size or retraining period. In fact, a too frequent retraining likely brings again scalability problems, as training a classifier on an ever-growing ground truth becomes more expensive as time goes by. On the other hand, waiting too long for retraining prevents to steadily identify and react to new malware families.

To solve this problem, in this paper we propose MalFamAware, a novel approach to *incremental automatic family identification and malware classification*, based on *online clustering*, where samples are efficiently assigned to families as soon as they are fed to the clustering algorithm, and the families (i.e. the clusters) are updated accordingly. This *update* only modifies the number and the composition of the families without altering the features space, which remains fixed and consistent among the various executions. The advantage over incremental clustering lies in cutting the cost of periodically re-training a classifier. One of the contributions of this work is the idea of using online clustering to implement family identification and malware classification simultaneously. Table 1 shows an overall comparison between existing approaches for malware classification and family identification.

Approach	**Malware classification**	**Family identification**
Classifier-only	Classifier trained on available ground truth	No family identification, it needs period retraining
Re-clustering	Classifier trained on available ground truth	Ground truth updated by periodic clustering of the whole dataset
Incremental clustering	Classifier trained on available ground truth	Ground truth updated by periodic clustering of malware whose family is still unsure
Online clustering	*Done simultaneously*	

Table 1: Existing approaches for malware classification and family identification.

MalFamAware uses BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [6] as online clustering algorithm. BIRCH is one of the best clustering algorithms for family identification [7] and it can be used in an online fashion. We present experimental results on a dataset of publicly available Windows malware, analyzed with the Cuckoo Sandox[2] to extract static and dynamic features. Contrary to our previous work [7], this paper focuses on both *family identification* and *malware family classification*, presenting a solution to perform them in the same phase. Indeed, MalFamAware classifies new incoming malware into the correct family, if already known, or creates a new one if no suitable existing family is found. We show that MalFamAware outperforms approaches based on re-clustering or incremental clustering, both in terms of accuracy and execution time. In particular, MalFamAware requires three orders of magnitude less time than other clustering-based approaches to analyse samples, while providing slightly better accuracy. We also compare the accuracy of MalFamAware against that achieved by using different standard classifiers trained in two distinct ways: only once at the beginning or periodically on all the previous samples. Results show that the accuracy of MalFamAware is comparable to that of other classifiers when samples belong to known families, and it is larger for samples of new families.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 explains how BIRCH works. Section 4 details MalFamAware and Section 5 extensively describes the experimental evaluations we carried out. Finally, Section 6 draws conclusions and discusses future work.

2 Related Work

The problem of classifying malware in families has been extensively addressed in literature, as widely reported by some recent surveys on the topic, such as those by Ye *et al.* [8] and Ucci *et al.* [9]. Family identification has been investigated as well, although less thoroughly than malware classification. Pitolli *et al.* [7] propose to use BIRCH to cluster malware in families, but these families were fixed. Kinable and Kostakis [10] investigate call graph clustering, where samples are modelled as call graphs and clusters are computed on the basis of the graph edit distance. DUET [11] is a system based on cluster ensemble to combine and get the best out of clusterings computed by different algorithms. FIRMA [12] relies on network traffic features to determine the clusters. Bayer *et al.* [13] and BitShred [14] focus on improving the scalability of malware clustering.

To the best of our knowledge, only a few works address malware classification and family identification in an integrated way. As already mentioned in section 1, Malheur [5] is an open-source[3] system which identifies malware families showing similar behaviour and assigns new samples to identified families. It extracts features from textual malware analysis reports, such as those produced by Cuckoo Sandbox. Besides supporting the standard approach based on total re-clustering, it also allows for incremental clustering (i.e., *Incremental Malheur*). Incremental Malheur uses knowledge on families identified during the latest clustering to train a classifier for quickly classifying novel samples. For each new batch of malware, Incremental Malheur (i) clusters only samples that have not been classified with a sufficiently high confidence, (ii) adds resulting new families to those already known, and (iii) re-trains the classifier accordingly. An accuracy and performance comparison between our solution and Incremental Malheur is reported in section 5.4.2.

Zhong *et al.* [15] propose ARIGUMA, which implements the same approach of Malheur. The reported experimental results, stemming from a small-scale dataset containing 626 samples, outline an accuracy of 61.6%. We select Malheur for comparison because ARIGUMA is not available for download and use.

Malware classification into known families has been also investigated for Android apps, e.g. by Deshotels *et al.* [16] and by Garcia *et al.* [17]. The first work focuses on creating automatisms to generate signatures for family recognition. The second one aims to classify obfuscated Android malware into known families. Both works present the same limitation of other malware classification approaches, i.e. they cannot classify correctly malware belonging to families that were not in the training set. Family identification for Android malware has been explored as well. Li *et al.* [18] iteratively mine sample payload to remove legitimate libraries and verify whether different malware use the same payload version, then use clustering to group malware in families. Aresu *et al.* [19] analyse the logs of HTTP traffic generated by malicious apps to extract statistical information, which are then used to cluster Android malware in families by using BIRCH algorithm. Both works rely on clustering algorithms to identify families but address neither malware classification nor scalability issues.

A common problem of approaches based on machine learning is their potential vulnerability to attacks where

[2] `https://www.cuckoosandbox.org/`

[3] `http://www.mlsec.org/malheur/`

an adversary prepares samples in such a way to fool classification or clustering algorithms, with the aim to make them malfunction, e.g. to achieve the misclassification of a sample. The relatively recent research field that addresses this problem is referred to as *Adversarial Machine Learning* [20].

Adversarial machine learning has been investigated in the field of malware analysis as well. EvadeML [21] shows that classifiers used for PDF malware detection can be evaded by performing stochastic manipulations of samples to generate new variants. IagoDroid [22] implements an effective attack against malware classification techniques used in triage stages. IagoDroid takes as input a malware and a family, different from the one that malware belongs to, and generates a variant that is classified into that family, hence achieving evasion.

Adversarial learning for clustering in the field of malware analysis has been explored as well. Biggio *et al.* [23] present a methodology to assess the security of clustering algorithms against this type of attacks and give evidence of a successful attack against malware behavioural clustering in families. A similar, effective attack against Malheur clustering is shown in [24]. Referenced literature also propose countermeasures to make classifiers and clustering algorithms robust against those attacks. In our work we focus on the feasibility and effectiveness of performing malware classification and family identification together and, although MalFamAware can be vulnerable to similar attacks, we leave the corresponding adversarial analysis as future work.

3 Balanced Iterative Reducing and Clustering using Hierarchies

Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) is a clustering algorithm optimized to perform hierarchical clustering over particularly large data-sets. It exploits the concepts of how the space of data is not usually uniformly occupied and how data points are not equally important. Based on these assumptions it can make each clustering decision without scanning all data points and currently existing clusters. One great advantage of BIRCH is that it makes full use of available memory to derive the finest possible sub-clusters while minimizing I/O costs.

BIRCH is built around the concept of *Clustering Feature [CF]* and *CF Tree*. It tries to minimise the memory requirements of large datasets by summarising the information contained in dense regions as *Clustering Feature* (CF) entries. Given N d-dimensional data points in a cluster $\overrightarrow{X}_i$, where i=1,2,...,N, the CF entry of the cluster is defined as a triple: $CF = (N, \overrightarrow{LS}, SS)$, where N is the number of data points in the cluster, $\overrightarrow{LS}$ is the linear sum of the N data points, i.e. $\sum_{i=1}^{N} \overrightarrow{X}_i$, and SS is the square sum of the N data points $\sum_{i=1}^{N} \overrightarrow{X}_i^2$ On the basis of this definition, it is possible to have CFs composed by other CFs. In this case, the sub-cluster is equal to the sum of the CFs.

BIRCH authors formalised the *CF Additive Theorem*: assume that $CF_1 = (N_1, \overrightarrow{LS}_1, SS_1)$ and $CF_2 = (N_2, \overrightarrow{LS}_2, SS_2)$ are the CF entries of two disjoint sub-clusters; then, the CF entry of the sub-cluster that is formed by merging the two disjoint sub-clusters is $CF_1 + CF_2 = (N_1 + N_2, \overrightarrow{LS}_1 + \overrightarrow{LS}_2, SS_1 + SS_2)$.

On top of these definitions, [6] describes the *CF-tree*: a compact representation of the dataset where each entry in a leaf node represents a sub-cluster. Thus, a single entry contains pointer to a child node and a CF made up of the sum of the CFs in the children (i.e., sub-clusters of sub-clusters). Each leaf node is a CF as well, i.e. a sub-clusters of data points. All entries in the leaf nodes should satisfy a *Threshold* requirement, i.e. the diameter of each leaf must be lower than this value.

3.1 Algorithm details

BIRCH operates through four phases, with the last being optional:

1. scans data and loads it into memory by building a CF tree. If memory is exhausted, it rebuilds the tree from the leaf nodes;
2. resizes data by building a smaller CF tree. BIRCH tries to remove more outliers and *condenses* the data;
3. directly applies an agglomerative hierarchical clustering algorithm to the sub-clusters represented by their CF vectors;
4. fixes problems with CF trees where same valued data points may be assigned to different leaf entries.

Differently from other offline clustering algorithms, BIRCH has the great advantage of clustering incoming feature vectors incrementally and dynamically. It uses CF vectors to summarise the information in each identified cluster. In this way, it can overcome a common limitation of other hierarchical clustering approaches: it can undo clustering choices performed in previous steps to correct the clusters according to newly analysed samples [25]. When a new feature vector is fed to BIRCH, it is assigned to either an existing cluster or to a novel cluster created on purpose for that sample.

4 MalFamAware Approach

The approach we propose in MalFamAware leverages online clustering to simultaneously perform family identification and malware classification. As already explained in Section 1, the former consists in discovering if the analysed malware belongs to a completely new family, while the latter aims to find which known family the analysed malicious software belongs to.

Our approach can be divided in two phases: a *tuning phase* (subsection 4.1) where required parameters are setup, and a *family identification and malware classification phase* (subsection 4.2) where available malware are analysed to determine their families and spot new families.

4.1 Tuning Phase

The tuning phase works in two consecutive steps. First, there is the computation of features normalisation values (subsection 4.1.1). Second, the BIRCH algorithm is configured on the basis of available ground truth (subsection 4.1.2).

4.1.1 Feature normalisation

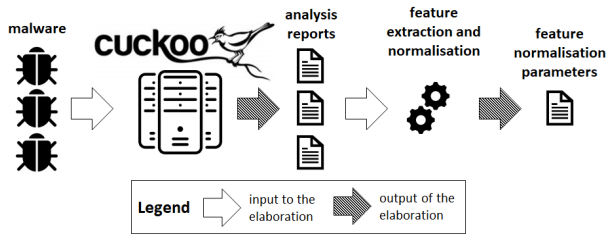

Fig. 1: MalFamAware workflow for feature normalisation.

A dataset of non-labelled malware is used for feature normalisation, Figure 1 shows the corresponding workflow. Each sample is executed for at most five minutes inside the Cuckoo Sandbox, which produces a textual analysis report containing information extracted both statically (e.g., header fields, strings, etc.) and dynamically (e.g., file system access, system calls, etc.) from the sample currently under analysis. A hybrid approach like this allows to mitigate the issues deriving from using either static analysis alone (e.g. evasion techniques based on obfuscation) or dynamic analysis alone (e.g. evasion techniques based on environment awareness). A feature vector with 241 numerical values for each sample is generated from the analysis report. String information are converted to numeric features by using locality-sensitive hashing, so that similarity among strings is preserved as much as possible. A summary of used features is reported in table 2, the complete list of used features is detailed in appendix A.

Category	Feature count
Static features	58
File system operations	53
Registry operations	36
Process operations	68
Network activities	26

Table 2: Categories of used features

To improve robustness against outliers, feature values are normalised with respect to the median and the interquartile range (IRQ) of values present in dataset. For each feature f_i, let m_i be the median of f_i values over the dataset, and let irq_i be the corresponding interquartile range. For each value x_j of f_i, the normalised value $\bar{x_i}$ is computed as follows.

$$\bar{x_i} = \frac{x_i - m_i}{irq_i}$$

4.1.2 BIRCH Tuning

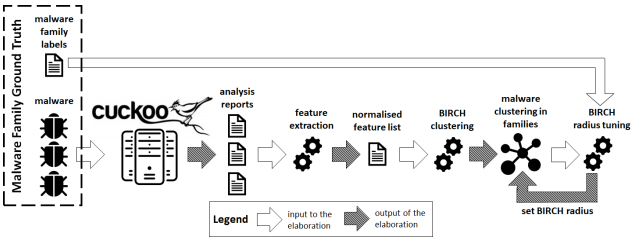

Fig. 2: MalFamAware workflow for the tuning of BIRCH radius.

During the tuning phase, the BIRCH radius is configured on the basis of an available ground truth. The workflow of this tuning is depicted in Figure 2. The tuning works iteratively. At each iteration, a different radius is used and the accuracy of the correspondent clustering is measured on the basis of the ground truth. The radius that maximizes the accuracy is chosen.

Note that in this paper we consider a setting of BIRCH where a single scan of the input dataset is run, which enables to analyse incoming samples in an online

fashion. BIRCH can be also configured to include additional three phases that rescan input data to refine and improve the clustering, which however would prevent the algorithm to work online.

4.2 Family Identification and Malware Classification Phase

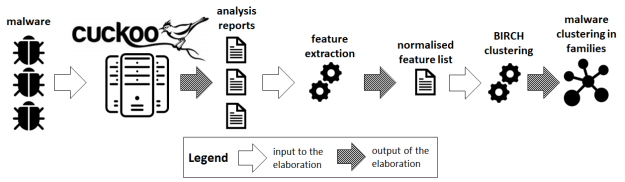

Fig. 3: MalFamAware workflow for family identification and malware classification. The final output, i.e. *malware clustering in families*, may have been updated (*family identification*) and includes the malware just analyzed in the proper cluster (*malware classification*).

After the tuning phase, samples are analyzed as they arrive following the workflow shown in Figure 3. Cuckoo analyses statically and dynamically the sample and produces an analysis report. Required features are extracted and normalised as described in section 4.1.1. The corresponding feature vector is fed to BIRCH algorithm, configured to work online and with a specific radius threshold, as explained in section 4.1.2.

BIRCH revises its clustering for each elaborated feature vector. A new malware family is identified if a new cluster is added, which takes place when the radius of an existing cluster exceeds the threshold. The cluster where the feature vector is put determines what family the corresponding malware has been classified in.

Malware classification is achieved by assigning samples to clusters, while family identification takes place whenever a new cluster is created. As sample clustering and the possible addition of a new cluster are carried out by BIRCH for each sample it receives, malware classification and family identification basically occur simultaneously.

5 Experimental evaluation

In this section we report the experimental results on the effectiveness of MalFamAware, in terms of accuracy and performance. We first describe the testbed we use (§ 5.1) and what ground truth we consider (§ 5.2), then we detail experiments and results on family identification (§ 5.3) and malware classification (§ 5.4).

5.1 Testbed

We test MalFamAware on a single machine equipped with an Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz, using 4 cores and 8 GB of memory. The operating system is Ubuntu 16.04.2 with kernel GNU/Linux 4.4.0. For the implementation we use well-known publicly available libraries[4] that provide machine learning algorithm tools.

5.2 Ground Truth

To the best of our knowledge, no large-scale Windows malware dataset containing samples grouped in families is publicly available. The lack of this ground truth cannot be easily overcome with manual labelling [3] because it is an extremely time-consuming task. Following the approach proposed in several previous works [14, 26, 5, 27, 7], we build a ground truth by using labels assigned by antivirus software.

This approach has well-known issues that mainly derive from the inconsistent label sets used by different security firms [28, 29, 30]. To address this issue, we use the *AVclass* tool [31], which analyses the labels assigned by antiviruses and chooses the one picked by the majority of them. When a majority is not available, AVclass assigns a unique label to the sample, i.e. the malware is placed in a *singleton* cluster.

We collected 5351 malware samples for the Windows operating system from VirusTotal[5]. In order to enable the repeatability of our experiments, the list of MD5 hashes of all the used samples is available online [6]. With the aim of building a malware family ground truth without under-represented families, we select only the samples of families with at least 10 malware, in line with other works [32, 33] that use the same approach of discarding families with a few samples. After this selection, we had 4613 samples divided in 18 families; table 3 reports the distribution of malware over these families. To properly simulate how samples are analyzed over time, we consider their temporal distribution on the basis of the timestamps reported in their header. Although some samples have clearly forged timestamps (e.g. in the future), in the large majority of cases timestamps seem to be realistic. The distribution of samples over time is shown in Figure 4. We include in our reference dataset only samples between 2006 and 2016, which amount to around 4100 samples. To enable a fair

[4] scikit-learn (`http://scikit-learn.org/`) and SciPy (`https://www.scipy.org/`)

[5] VirusTotal, `https://www.virustotal.com/`

[6] `http://users.diag.uniroma1.it/aniello/malware_dataset/malware_dataset_md5_list.txt`

Family Name	Samples
allaple	55
cosmu	99
domaiq	33
downloadguide	82
flystudio	417
gator	39
hotbar	21
installcore	12
yantai	130
installerex	47
loadmoney	16
mira	53
multiplug	85
phishbank	1859
ramnit	1115
soft32downloader	13
sytro	412
vobfus	125

Table 3: Distribution of samples over families containing at least 10 elements.

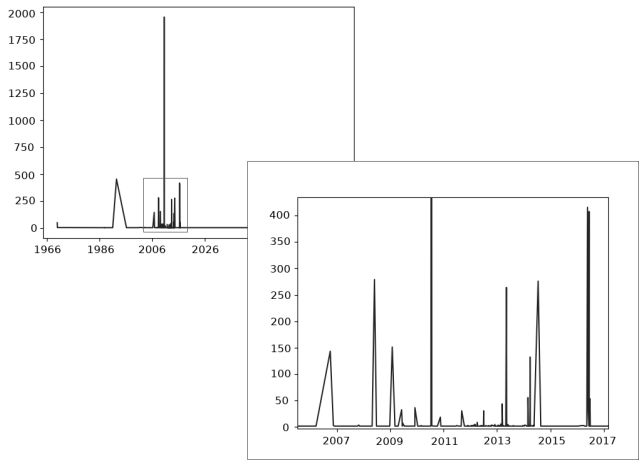

Fig. 4: Temporal distribution of samples.

comparison with other incremental approaches where malware are analyzed periodically in blocks, we organise samples in batches according to their timestamps, obtaining 10 blocks with approximately 400 samples each. Figure 5 reports for each block the percentage of samples belonging to unknown families, i.e. those families for which no sample is present in any of the previous blocks. This information is useful to evaluate the

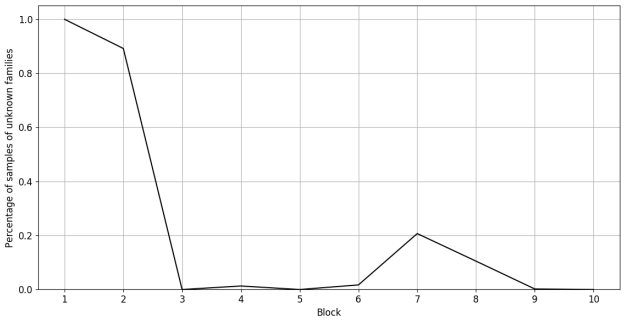

Fig. 5: Percentage of samples of unknown families in each block.

accuracy of approaches that deal with malware of new families.

5.3 Family Identification

We assess the capability of MalFamAware to identify families by evaluating how it clusters malware with BIRCH against the ground truth (§ 5.2), and comparing clustering accuracy with respect to a number of well-known algorithms. We first introduce the metrics used to compare clustering accuracy (§ 5.3.1) and the comparison algorithms (§ 5.3.2), then we present family identification results on accuracy (§ 5.3.3) and performance (§ 5.3.4).

5.3.1 Clustering Accuracy Metrics

To evaluate the accuracy of an algorithm, we compare the resulting clustering to the ground truth (§ 5.2), which can be considered as a clustering having one cluster for each family, containing all the samples of the family itself. In our experiments, we consider the following three accuracy metrics to measure how similar two clusterings are: ARI, AMI and FMI.

– *Adjusted Rand Index* (**ARI**). The Rand index [34] is defined as the number of pairs of samples that are either in the same family or in different families in both clusterings, divided by the total number of pairs of samples, hence it lies in the nominal range $[0, 1]$. In practice, the Rand index frequently has value greater than 0.5 and its baseline, i.e. the average between random partitions, is not constant. Assuming the generalized hyper-geometric distribution as a model of randomness, the Rand index is *adjusted for chance* to take into account its expected value. The resulting Adjusted Rand Index (ARI) [35] lies in the range $[-1, 1]$, where 1 still

means a perfect matching and 0 represents the fact that the Rand index equals its expected value.
- *Adjusted Mutual Information* (**AMI**). The mutual information measures the mutual dependence between two random variables. It can be used to compare two clusterings by considering their associated entropies as random variables. The entropy of a clustering is computed on the basis of the probability that a sample belong to a specific cluster [36]. Analogously to the way the Rand index is adjusted through the ARI, the Adjusted Mutual Information (AMI) [37] adjusts for chance the mutual information by considering the difference with respect to its expected value, which also in this case is computed by assuming a generalized hyper-geometric distribution as model of randomness. The AMI takes a value of 0 when the mutual information between the two clusterings is the same as its expected value, and it is 1 when the two clusterings are identical.
- *Fowlkes-Mallows Index* (**FMI**). The Fowlkes-Mallows Index (FMI) [38] is yet another clustering similarity metric, which can be defined as the geometric mean of precision and recall, i.e. $\sqrt{\frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}}$. The latter are computed on the basis of the number of samples that are in common or not in the two clusterings, as follows. TP represents how many pair of samples are in the same cluster in both clusterings. FP is the number of sample pairs that are in the same cluster in the first clustering but not in the second one, while FN counts how many pairs belong to the same cluster in the second clustering but not in the first one. Finally, TN represent the number of pairs of samples that are in different clusters in both clusterings. FMI takes values closer to 1 as the similarity of the clusterings increases, while the nearer FMI is to 0 the more clusterings are different.

5.3.2 Clustering Algorithms for Comparison

This section briefly describes the clustering algorithm chosen for comparison with BIRCH.

- Density-Based Spatial Clustering of Applications with Noise (**DBSCAN**) [39] is a density-based algorithm that can discover clusters of arbitrary shapes and is efficient for large spatial datasets. The algorithm looks for clusters by searching the neighbourhood of each data point in the dataset.
- **Hierarchical algorithms** [40] construct a tree of clusters, also known as a *dendrogram*. Every cluster node contains child clusters, thus allowing the exploration of data at different levels of granularity. These algorithms need a *linkage criterion* to determine the distance between sets of observations as a function of the pairwise distances between single observations. Most common linkage criteria are **Ward's link**, **Single-link**, **Complete-link** and **Average-link**.
- **K-Means** [41] is a well known center-based algorithm. The center of each cluster, called centroid, is the mean of all the instances of that cluster. The number of clusters k is assumed to be fixed.
- **Mini-Batch K-Means** [42] is a variant of K-Means which uses mini-batches to reduce computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration.
- Expectation Maximization (**EM**) [43] is a general-purpose statistical iterative method of maximum likelihood estimation in the presence of incomplete data, which can be used for the purpose of clustering.
- Clustering Using Representative (**CURE**) [44] is a scalable and efficient algorithm for large datasets that is robust to outliers and is able to identify clusters of arbitrary shapes and sizes. In this algorithm, each cluster is represented by a constant number of points that are well scattered in the cluster itself.

All these algorithms rely on specific parameters which affect their behaviour. BIRCH depends on a threshold which defines the maximum radius of clusters (§ 4.1.2). K-Means needs the setup of the number of clusters to generate. The two main parameters of DBSCAN are (i) the maximum distance between any two samples to be considered in the same neighbourhood, and (ii) the minimum amount of data points in a neighbourhood to be considered a cluster. Hierarchical algorithms need a threshold to decide when to cut the tree. EM has several parameters, including the number of mixture components. Also CURE needs to setup in advance the number of clusters, as well as the number of representative points for each cluster. For each algorithm, we iteratively tune parameters to maximise the accuracy. In particular, we choose to maximise the FMI. When multiple parameters have to be configured, we tune them one by one. The maximisation approach is based on numerical approximation with direct method: the same algorithm is executed a large number of times on the same input data, each time with a different value of the parameters, in order to cover as much as possible a given range of values. For example, for BIRCH, we vary the value of the threshold between 0 and 10 with a step of 0.1. Figure 6 shows how the FMI for BIRCH evolves by varying the threshold value.

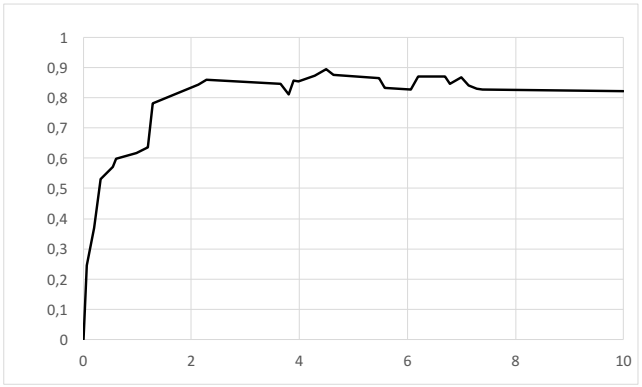

Fig. 6: BIRCH FMI accuracy by varying radius treshold.

5.3.3 Family Identification Accuracy

Clustering algorithms can generate different clusterings depending on the order of input samples, thus we execute each algorithm 40 times with malware sorted differently. Gigures 7, 8 and 9 show the comparison between BIRCH and the other clustering algorithms (§ 5.3.2), in terms of ARI, AMI and FMI, respectively (§ 5.3.1). The accuracy values are averaged over the 40 executions, and error bars are included to show how the accuracy varies by changing the order of input malware. Table 4 reports all the average accuracy scores.

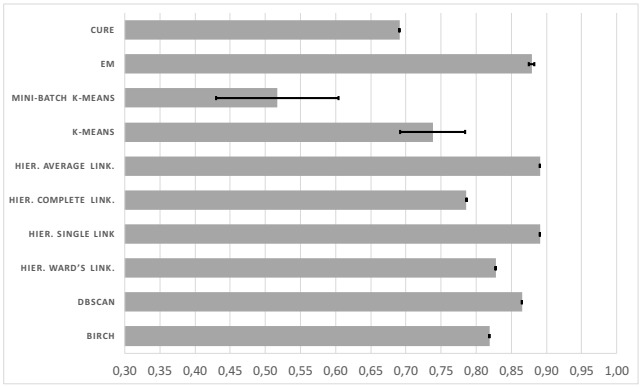

Fig. 7: ARI achieved by clustering algorithms.

These results show that BIRCH achieves an accuracy, in terms of ARI and FMI, lower than some other hierarchical algorithms and than EM and DBSCAN. It is to highlight that (i) we use a setting of BIRCH purposely aimed at favouring performance over accuracy (§ 4.1.2), as can be proved in section 5.3.4, and (ii) EM and DBSCAN requires as input the number of clusters to generate, while BIRCH does not.

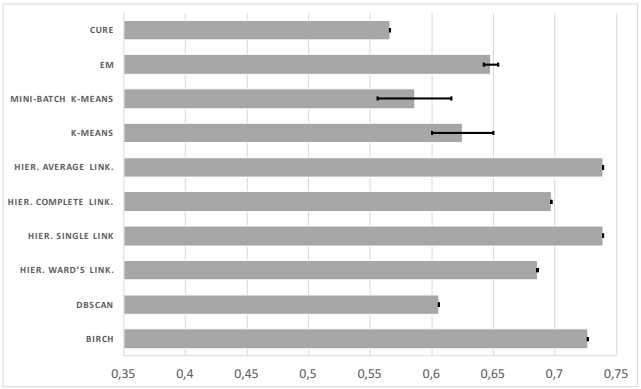

Fig. 8: AMI achieved by clustering algorithms.

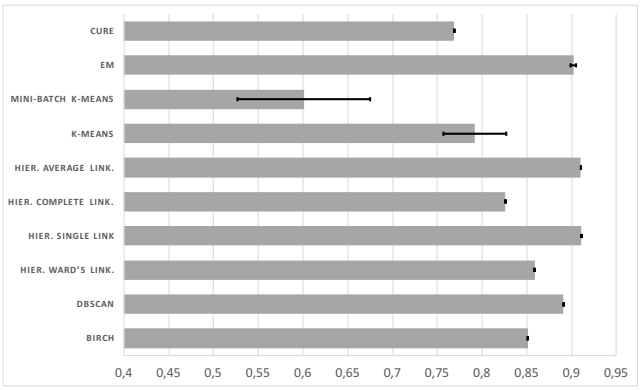

Fig. 9: FMI achieved by clustering algorithms.

5.3.4 Family Identification Performance

Figure 10 shows average and standard deviation of the execution times of each algorithm. These results clearly prove that BIRCH is much faster than all the other algorithms, except for the Mini-Batch K-means, which, however, turns out be the least accurate. The most accurate algorithms (i.e. hierarchical algorithms) perform significantly worse than BIRCH.

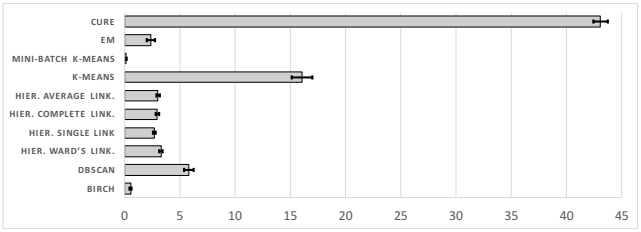

Fig. 10: Execution times in seconds of clustering algorithms.

Algorithm	k	ARI	AMI	FMI
BIRCH	NO	0.819	0.727	0.851
DBSCAN	NO	0.865	0.606	0.891
Hierarchical Ward's linkage	NO	0.828	0.686	0.859
Hierarchical single linkage	NO	0.891	0.739	0.911
Hierarchical complete linkage	NO	0.786	0.697	0.826
Hierarchical average linkage	NO	0.891	0.739	0.91
K-Means	YES	0.738	0.625	0.792
Mini-Batch K-Means	YES	0.517	0.586	0.601
EM	YES	0.879	0.648	0.902
CURE	YES	0.691	0.566	0.769

Table 4: Comparison of considered algorithms in terms of clustering accuracy. Average accuracy scores are reported, measured as ARI, AMI and FMI. The column **k** indicates whether the algorithm requires to know in advance the number of clusters.

5.4 Malware Classification

We evaluate the effectiveness of MalFamAware in classifying malware in families over time (i.e. batch by batch) by measuring its accuracy against the available ground truth (§ 5.2) and comparing it with other well-known classifiers. In all these experiments we use the first batch to learn the BIRCH radius threshold (§ 4.1.2). We first define the metrics used to measure classification accuracy (§ 5.4.1), then we present a number of experiments on accuracy comparison: between MalFamAware and Malheur (§ 5.4.2), and between MalFamAware and other classifiers (§ 5.4.3). Finally, we also show comparative results on classification time (§ 5.4.4).

5.4.1 Classification Accuracy Metrics

We have to evaluate to what extent a clustering is similar to the available ground truth, in the same way we can compare the output of a classifier. It is to note that this problem is different from the one we addressed to assess family identification accuracy (§ 5.3.1), indeed for malware classification we want to evaluate the accuracy after every batch, not only when all the samples have been analyzed. The labels of our ground truth, produced by AVclass [31], are different from those generated by clustering algorithms, which assign anonymous labels such as "cluster1". We have to choose comparison metrics that are independent from the naming space of labels, thus we decided to use precision, recall and accuracy (i.e. the F1 score) as defined in [45].

Given the set of labels $\mathcal{L}^{GT} = \{l_i^{gt}\}$ of the classes of the ground truth, with $i = 1 \dots N^{GT}$, let n_i^{gt} be the number of samples labelled with l_i^{gt}. Given the set of labels $\mathcal{L}^{C} = \{l_i^{c}\}$ produced by a clustering algorithm, with $i = 1 \dots N^{C}$, let n_i^{c} be the number of samples labelled with l_i^{c}. Let $n_{i,j}$ be the number of samples of class l_i^{gt} (according to the ground truth) that have been assigned to cluster l_j^{c} by the clustering algorithm. We first define precision and recall for a class l_i^{gt} of the ground truth with respect to a cluster l_j^{c} identified by the clustering algorithm:

$$Precision(l_i^{gt}, l_j^{c}) = n_{i,j}/n_j^{c}$$
$$Recall(l_i^{gt}, l_j^{c}) = n_{i,j}/n_i^{gt}$$

Given the total number n of samples that have been clustered, we then define the overall precision, recall and F1 score (i.e. the accuracy):

$$Precision = \sum_{l_i^{gt} \in \mathcal{L}^{GT}} \frac{n_i^{gt}}{n} \max_{l_j^{c} \in \mathcal{L}^{C}} \left\{ Precision(l_i^{gt}, l_j^{c}) \right\}$$

$$Recall = \sum_{l_i^{gt} \in \mathcal{L}^{GT}} \frac{n_i^{gt}}{n} \max_{l_j^{c} \in \mathcal{L}^{C}} \left\{ Recall(l_i^{gt}, l_j^{c}) \right\}$$

$$F1 = \frac{2 \cdot Recall \cdot Precision}{(Precision + Recall)}$$

We use these metrics also to compare the accuracy of classifiers, which have the same labels of the ground truth because they are trained on it. To make such comparison as fair as possible, we do not take into account this fact and assume the classifiers output labels in a different namespace.

Another key aspect to investigate and measure is the capability to deal with samples belonging to families that are still not known. At this regard, we introduce another metric, referred to as *weighted accuracy*, computed as the product of the accuracy obtained for the samples of a certain batch and the percentage of malware in that batch that belong to families still unknown.

5.4.2 Comparison between MalFamAware and Malheur

Figure 11 reports the accuracy comparison between BIRCH, Malheur and Incremental Malheur (§ 2). Figure 12 shows the same comparison using the weighted accuracy as metric (§ 5.4.1). BIRCH outperforms the other two algorithms in the large majority of batches. Incremental Malheur performs quite badly for blocks 8 and 9. In

both cases, this is due to the presence in the previous block of several malware samples that do not belong to any known family.Incremental Malheur does not update the clusters identified previously, thus over time it may end up with an overall clustering that does not result as accurate as that of BIRCH or Malheur.

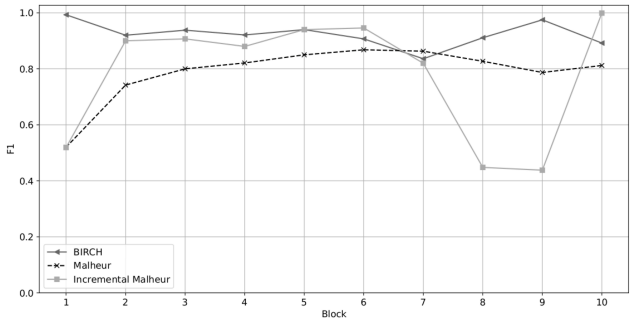

Fig. 11: Accuracy comparison between BIRCH, Malheur and Incremental Malheur.

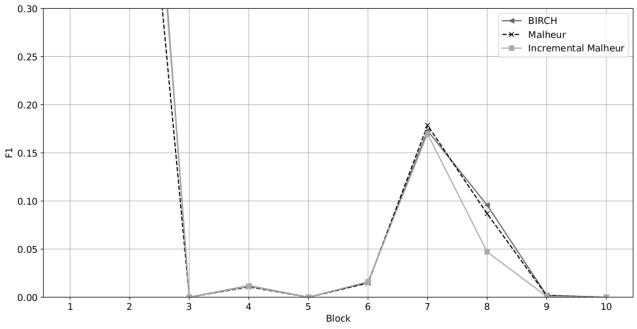

Fig. 12: Weighted accuracy comparison between BIRCH, Malheur and Incremental Malheur.

5.4.3 Comparison between MalFamAware and Classifiers

We compare the accuracy of BIRCH against that of several standard classification algorithms: Random Forest, Support Vector Machines (SVM), Gaussian Naive-Bayes and a Multi-layer Perceptron. We train these classifiers in two distinct ways: (i) once at the beginning using only the samples contained in first block as training set (see Figure 13), and (ii) at every batch, where the training set used to analyse the i-th batch includes all the samples contained in blocks 1 to $i-1$ (see Figure 14).

In both cases, BIRCH shows an accuracy comparable with the most accurate classifiers. The evident accuracy degradation for block 7 is justified as this block is the one having the largest number of samples belonging to unknown families. This degradation affects all the algorithms, but BIRCH proves to be the least affected. Surprisingly, all the standard classification algorithms perform better when they are trained only on the samples contained in the first block.

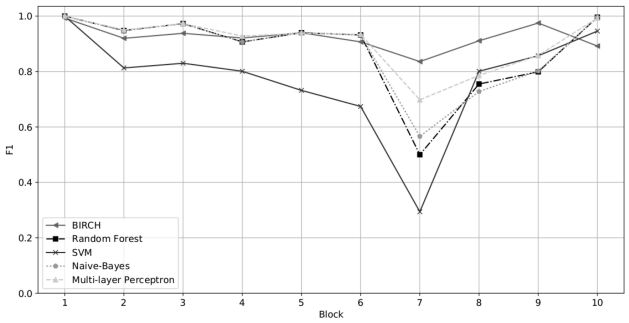

Fig. 13: Accuracy comparison between BIRCH and classifiers trained once on the first block.

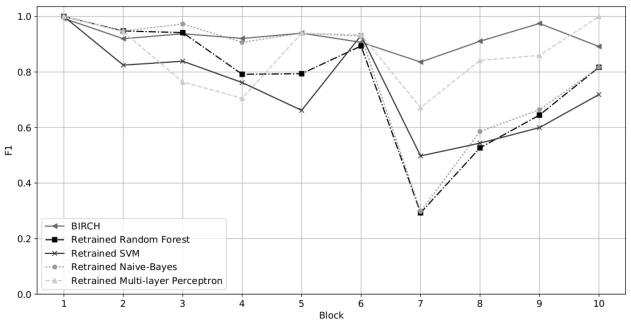

Fig. 14: Accuracy comparison between BIRCH and classifiers retrained at every block.

Figures 15 and 16 show the same comparison using the weighted accuracy (§ 5.4.1) to highlight how these algorithms behave when samples belong to yet unknown families. BIRCH outperforms standard classification algorithms, both when trained once at the beginning and when retrained at each batch.

Finally, Table 5 reports precision, recall and F1 score for the tested approaches, averaged over all the 10 blocks together with the standard deviation value. It can be observed that BIRCH provides the highest precision and F1 score.

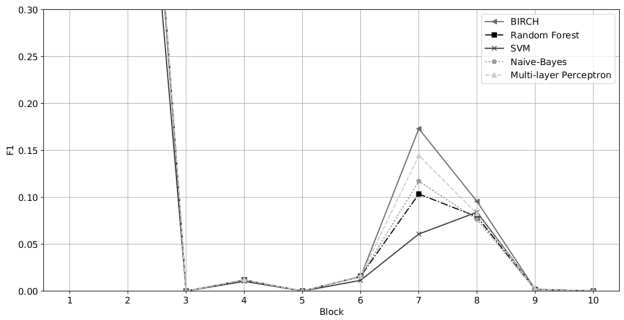

Fig. 15: Weighted accuracy comparison between BIRCH and classifiers trained once on the first block.

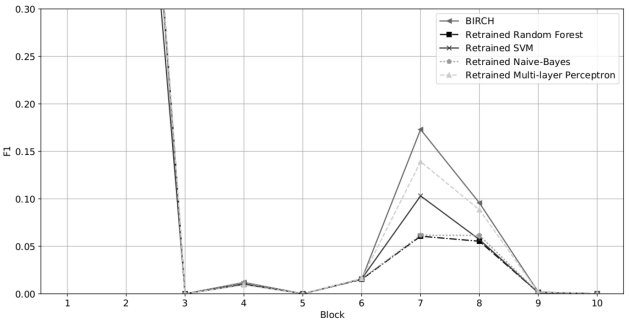

Fig. 16: Weighted accuracy comparison between BIRCH and classifiers retrained at every block.

5.4.4 Malware Classification Performance

We finally compare the execution time required for each block. Results are reported in Figure 17 in logarithmic scale. As we here consider the classifiers trained once at the beginning with the samples of the first block, we do not include their execution time for the first block. The picture shows that BIRCH exhibits execution times similar to standard classification algorithms, and at least three orders smaller than Malheur and Incremental Malheur.

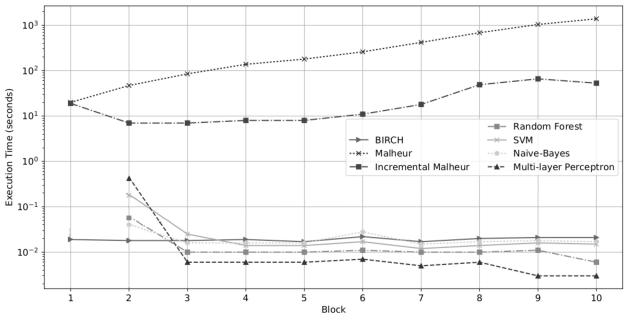

Fig. 17: Execution time comparison for all the tested algorithms.

Algorithm	Precision	Recall	F1
BIRCH	0.952 ± 0.043	0.902 ± 0.077	0.923 ± 0.041
Malheur	0.941 ± 0.018	0.693 ± 0.128	0.789 ± 0.097
Incremental Malheur	0.949 ± 0.043	0.724 ± 0.285	0.78 ± 0.209
Naive-Bayes	0.789 ± 0.181	0.986 ± 0.026	0.866 ± 0.133
Retrained N-B	0.69 ± 0.254	0.997 ± 0.003	0.785 ± 0.214
Random Forest	0.783 ± 0.196	0.995 ± 0.013	0.861 ± 0.148
Retrained R.F.	0.677 ± 0.26	0.995 ± 0.013	0.739 ± 0.204
SVM	0.746 ± 0.217	0.84 ± 0.152	0.75 ± 0.176
Retrained SVM	0.701 ± 0.227	0.821 ± 0.166	0.709 ± 0.137
Multi-layer Perceptron	0.834 ± 0.13	0.977 ± 0.038	0.895 ± 0.091
Retrained M-l P.	0.833 ± 0.131	0.901 ± 0.151	0.851 ± 0.109

Table 5: Precision, Recall and F1 score for all the tested algorithms.

6 Conclusions and future work

In this paper we propose an approach, MalFamAware, to carry out simultaneously family identification and malware classification by using an online clustering algorithm, namely BIRCH. We present an extensive experimental evaluation where we compare how MalFamAware performs in terms of family identification (against other clustering algorithms) and malware classification (against Malheur, Malheur Incremental and other classification algorithms). Results prove MalFamAware has good accuracy in family identification and high accuracy in malware classification, showing very low execution time.

As main future work we plan to carry out more extensive experiments with more and larger batches, in order to assess the reliability of our approach. Additionally, we want to study the convenience of introducing periodic total reclustering whenever an updated and reliable ground truth becomes available, in order to properly realign the clustering in families and to retrain BIRCH radius threshold. As introduced in Section 2, another topic that deserve investigation is *Adversarial Machine Learning*, we need to analyze and improve the robustness of our solution regards to this kind of attacks.

7 Compliance with Ethical Standards

Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. G. Laurenza, D. Ucci, L. Aniello, R. Baldoni, An architecture for semi-automatic collaborative malware analysis for cis, in: 3rd International Workshop on Reliability and Security Aspects for Critical Infrastructure, 2016.
2. G. Laurenza, L. Aniello, R. Lazzeretti, R. Baldoni, Malware triage based on static features and public APT reports, in: Proceedings of the First International Conference on Cyber Security Cryptography and Machine Learning, CSCML, 2017, pp. 288–305.
3. A. Mohaisen, O. Alrawi, M. Mohaisen, Amal: High-fidelity, behavior-based automated malware analysis and classification.
4. L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, R. Baldoni, Android malware family classification based on resource consumption over time, in: 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), 2017, pp. 31–38.
5. K. Rieck, P. Trinius, C. Willems, T. Holz, Automatic analysis of malware behavior using machine learning 19 (4) 639–668.
6. T. Zhang, R. Ramakrishnan, M. Livny, BIRCH: an efficient data clustering method for very large databases, in: ACM Sigmod Record, Vol. 25, ACM, pp. 103–114.
7. G. Pitolli, L. Aniello, G. Laurenza, L. Querzoni, R. Baldoni, Malware family identification with BIRCH clustering, in: Proceedings of the 51st Annual International Carnahan Conference on Security Technology (ICCST), ICCST '17, 2017.
8. Y. Ye, T. Li, D. Adjeroh, S. S. Iyengar, A survey on malware detection using data mining techniques, ACM Comput. Surv. 50 (3) (2017) 41:1–41:40.
9. D. Ucci, L. Aniello, R. Baldoni, Survey of machine learning techniques for malware analysis, Computers & Security (2018).
10. J. Kinable, O. Kostakis, Malware classification based on call graph clustering, Journal in computer virology 7 (4) (2011) 233–245.
11. X. Hu, K. G. Shin, Duet: integration of dynamic and static analyses for malware clustering with cluster ensembles, in: Proceedings of the 29th annual computer security applications conference, ACM, 2013, pp. 79–88.
12. M. Z. Rafique, J. Caballero, Firma: Malware clustering and network signature generation with mixed network behaviors, in: International Workshop on Recent Advances in Intrusion Detection, Springer, pp. 144–163.
13. U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, E. Kirda, Scalable, behavior-based malware clustering., in: NDSS, Vol. 9, Citeseer, pp. 8–11.
14. J. Jang, D. Brumley, S. Venkataraman, Bitshred: feature hashing malware for scalable triage and semantic analysis, in: Proceedings of the 18th ACM conference on Computer and communications security, ACM, pp. 309–320.
15. Y. Zhong, H. Yamaki, Y. Yamaguchi, H. Takakura, Ariguma code analyzer: Efficient variant detection by identifying common instruction sequences in malware families, in: Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual, IEEE, 2013, pp. 11–20.
16. L. Deshotels, V. Notani, A. Lakhotia, Droidlegacy: Automated familial classification of android malware, in: Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014, ACM, 2014, p. 3.
17. J. Garcia, M. Hammad, S. Malek, Lightweight, obfuscation-resilient detection and family identification of android malware, ACM Transactions on Software Engineering and Methodology (TOSEM) 26 (3) (2018) 11.
18. Y. Li, J. Jang, X. Hu, X. Ou, Android malware clustering through malicious payload mining, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2017, pp. 192–214.
19. M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, G. Giacinto, Clustering android malware families by http traffic, in: Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on, IEEE, 2015, pp. 128–135.
20. L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, J. D. Tygar, Adversarial machine learning, in: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, AISec '11, ACM, New York, NY, USA, 2011, pp. 43–58. doi:10.1145/2046684.2046692.
URL http://doi.acm.org/10.1145/2046684.2046692
21. W. Xu, Y. Qi, D. Evans, Automatically evading classifiers, in: Proceedings of the 2016 Network and Distributed Systems Symposium, 2016, pp. 21–24.
22. A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador, D. Clark, Picking on the family: Disrupting android malware triage by forcing misclassification, Expert Systems with Applications 95 (2018) 113–126.
23. B. Biggio, I. Pillai, S. Rota Bulò, D. Ariu, M. Pelillo, F. Roli, Is data clustering in adversarial settings secure?, in: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, AISec '13, ACM, New York, NY, USA, 2013, pp. 87–98. doi:10.1145/2517312.2517321.
URL http://doi.acm.org/10.1145/2517312.2517321
24. B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, F. Roli, Poisoning behavioral malware clustering, in: Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop, AISec '14, ACM, New York, NY, USA, 2014, pp. 27–36. doi:10.1145/2666652.2666666.
URL http://doi.acm.org/10.1145/2666652.2666666
25. J. Han, J. Pei, M. Kamber, Data mining: concepts and techniques, Elsevier.
26. M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, J. Nazario, Automated classification and analysis of internet malware, in: International Workshop on Recent Advances in Intrusion Detection, Springer, pp. 178–197.
27. G. Wicherski, pehash: A novel approach to fast malware clustering., LEET 9 (2009) 8.
28. P. Li, L. Liu, D. Gao, M. K. Reiter, On challenges in evaluating malware clustering., in: RAID, Vol. 6307, Springer, 2010, pp. 238–255.
29. A. Mohaisen, O. Alrawi, Av-meter: An evaluation of antivirus scans and labels, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, pp. 112–131.
30. D. Harley, The game of the name malware naming, shape shifters and sympathetic magic, in: CEET 3rd Intl. Conf. on Cybercrime Forensics Education & Training, San Diego, CA.
31. M. Sebastián, R. Rivera, P. Kotzias, J. Caballero, AVclass: A tool for massive malware labeling, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, pp. 230–253.
32. P. Kotzias, L. Bilge, J. Caballero, Measuring pup prevalence and pup distribution through pay-per-install services., in: USENIX Security Symposium, 2016, pp. 739–756.
33. M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, S. Zanero, Measuring and defeating anti-instrumentation-equipped malware, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2017, pp. 73–96.

34. W. M. Rand, Objective criteria for the evaluation of clustering methods 66 (336) 846–850.
35. L. Hubert, P. Arabie, Comparing partitions 2 (1) 193–218.
36. A. Strehl, J. Ghosh, Cluster ensembles–a knowledge reuse framework for combining multiple partitions, Journal of machine learning research 3 (Dec) (2002) 583–617.
37. N. X. Vinh, J. Epps, J. Bailey, Information theoretic measures for clusterings comparison: Is a correction for chance necessary?, in: Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, ACM, New York, NY, USA, 2009, pp. 1073–1080.
38. E. B. Fowlkes, C. L. Mallows, A method for comparing two hierarchical clusterings, Journal of the American Statistical Association 78 (383) (1983) 553–569.
39. M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al., A density-based algorithm for discovering clusters in large spatial databases with noise., in: Kdd, Vol. 96, 1996, pp. 226–231.
40. O. Maimon, L. Rokach, Data mining and knowledge discovery handbook, Vol. 2, Springer, 2005.
41. J. MacQueen, et al., Some methods for classification and analysis of multivariate observations, in: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Vol. 1, Oakland, CA, USA., 1967, pp. 281–297.
42. D. Sculley, Web-scale k-means clustering, in: Proceedings of the 19th international conference on World wide web, ACM, 2010.
43. C. B. Do, S. Batzoglou, What is the expectation maximization algorithm?, Nature biotechnology 26 (8) (2008) 897–899.
44. S. Guha, R. Rastogi, K. Shim, Cure: an efficient clustering algorithm for large databases, in: ACM Sigmod Record, Vol. 27, ACM, 1998, pp. 73–84.
45. M. Steinbach, G. Karypis, V. Kumar, et al., A comparison of document clustering techniques, in: KDD workshop on text mining, Vol. 400, Boston, 2000, pp. 525–526.

A Feature List

This appendix includes the complete list of used features in tables 6,7,8,10,9.

Name	Type
File size	Integer
Sections with virtual size / 2 >raw size	
Sections with entropy <1	
Sections with entropy >7	
Other sections count	
Resources count	
Imported libraries count	
Imported API count from Kernel*.dll	
Imported API count from User*.dll	
Imported API count from ADVAPI*.dll	
Imported API count from SHELL*.dll	
Imported API count from COMCTL*.dll	
Imported API count from CRYPT*.dll	
Imported API count from msvcr*.dll	
Imported API count from GDI*.dll	
Imported API count from SHLWAPI*.dll	
Imported API count from WS*.dll	
Imported API count from WININET.dll	
Imported API count from WINHTTP.dll	
Concatenated strings	String
Most frequent string	
Is 32 bit?	Binary
Has GUI?	
Has .rdata section?	
Has .data section?	
Has .rsrc section?	
Has .reloc section?	
Is present 'LoadLibrary*' ?	
Is present 'GetProcAddress' ?	
Is present 'MessageBox*' ?	
Is present 'ShellExecute*' ?	
Is present 'IsDebuggerPresent' ?	
Is present 'VirtualAlloc*' ?	
Is present 'CreateThread' ?	
Is present 'CreateProcess' ?	
Is present 'OpenProcess*' ?	
Is present 'RaiseException' ?	
Is present 'CreateEvent*' ?	
Is present 'GetSystemInfo' ?	
Is present 'GetComputerName*' ?	
Is present 'SetWindowsHook*' ?	
Is present 'WriteProcessMemory' ?	
Is present 'GetTickCount' ?	
Is present 'Sleep' ?	
Is present 'GetDiskFreeSpace*' ?	
Is present 'SetThreadContext' ?	
Is present 'CreateRemoteThread' ?	
Is present 'GetVersion*' ?	
Is present 'GetProcessHeap' ?	
Is present 'GetUserName' ?	
Is present 'ExitProcess' ?	
Is present '_CorExeMain' ?	
Is present 'WaitForSingleObject' ?	
Is present 'GetStartupInfo*' ?	

Is present 'GetKeyboard*' ?	
Is present 'SetUnhandledExceptionFilter' ?	
Is present 'HttpSendRequest' ?	
Is present 'HttpQueryInfo' ?	

Table 6: List of *Static Features*

Name	Type
Children processes count	Integer
Total processes count	
Main process threads count	
Main process children threads count	
Other processes threads count	
Other processes children threads count	
Executed commands count	
Started services count	
Created services count	
Number of calls to 'FlsGetValue'	
Number of calls to 'CreateWindow'	
Number of calls to 'GetSystemMetrics'	
Number of calls to 'NtMapViewOfSection'	
Number of calls to 'RtlRunDecodeUnicodeString'	
Number of calls to 'SystemParametersInfo'	
Number of calls to 'OpenService'	
Number of calls to 'RemoveDirectory*'	
Number of calls to 'NtOpenDirectory*'	
Number of calls to 'GetAdaptersAddresses'	
Number of calls to 'CopyFile'	
Number of calls to 'NtSetTimer'	
Number of calls to 'Process32Next'	
Number of calls to 'GetCursorPos'	
Number of calls to 'CryptHashData'	
Number of calls to 'CryptCreateHash'	
Number of calls to 'InternetSetOption'	
Number of calls to 'HttpQueryInfo'	
Number of calls to 'HttpAddRequestHeaders'	
Number of calls to 'InternetReadFile'	
Number of calls to 'HttpSendRequest'	
Number of calls to 'GetClipboard'	
Number of calls to 'UnhandledExceptionFilter'	
Number of calls to 'GetLastError'	
Number of calls to 'TerminateProcess'	
Number of calls to 'GetFileVersionInfo'	
Number of calls to 'HGetFolderPath'	
Number of calls to 'VirtualProtect'	
Number of calls to 'NtWaitForSingleObject'	
Number of calls to 'NtAllocateVirtualMemory'	
Number of calls to 'NtCreateThread*'	
Number of calls to 'NtQueryInformationProcess'	
Number of calls to 'NtResumeThread'	
Number of calls to 'NtTerminateProcess'	
Number of calls to 'NtCreateFile'	
Number of calls to 'NtOpenFile'	
Number of calls to 'NtQueryInformationFile'	
Number of calls to 'LdrLoadDll'	
Number of calls to 'NtCreateSection'	
Number of calls to 'NtOpenSection'	
Number of calls to 'NtQueryDirectoryFile'	
Number of calls to 'NtProtectVirtualMemory'	
Number of calls to 'NtQueryAttributesFile'	
Number of calls to 'DeviceIoControl'	
Number of calls to 'NtQuerySystemInformation'	
Number of calls to 'RegOpenKey*'	
Number of calls to 'RegCreateKey*'	
Number of calls to 'RegEnumValue*'	
Number of calls to 'RegQueryValue*'	
Number of calls to 'RegQueryInfoKey*'	
Number of calls to 'NtOpenKey'	
Number of calls to 'NtQueryValueKey'	
Number of calls to 'bind'	
Number of calls to 'connect'	
Number of calls to 'send'	
Number of calls to 'recv'	
Main process children names	String
Other processes names	
Other processes children names	

Table 7: List of *Process Features*

Name	Type
Read PIPEs count	Integer
Written PIPEs count	
Read files count from 'Fonts' folder	
Read files count from 'Assembly' folder	
Read files count from 'Chrome' folder	
Written files count in 'Chrome' folder	
Read files count from 'Python' folder	
Read files count from 'System32' folder	
Written files count in 'System32' folder	
Read files count from 'Temp' folder	
Written files count in 'Temp' folder	
Read files count from 'Device' folder	
Written files count in 'Device' folder	
Read files count from 'WindowsMail' folder	
Written files count in 'WindowsMail' folder	
Read files count from 'Common Files' folder	
Written files count in 'Common Files' folder	
Read files count from 'Program Files' folder	
Written files count in 'Program Files' folder	
Read files count from 'AppData' folder	
Written files count in 'AppData' folder	
Read files count from 'Microsoft.NET' folder	
Written files count in 'Microsoft.NET' folder	
Read files count from 'Microsoft' folder	
Written files count in 'Microsoft' folder	
Read files count from 'ProgramData' folder	
Written files count in 'ProgramData' folder	
Read files count from 'Documents and settings' folder	
Written files count in 'Documents and settings' folder	
Read files count from 'Windows' folder	
Written files count in 'Windows' folder	
Read files count from 'Users' folder	
Written files count in 'Users' folder	
Read files count from other folders	
Written files count in other folders	
Created EXE count	
Created BAT count	
Created COM count	
Created VBS count	
Created JPG count	
Created SYS count	

Created PIF count	
Created MSI count	
Created MSP count	
Created TMP count	
Created DAT count	
Read EXE count	
Read DLL count	
Read BAT count	
Created files count with multiple extensions	
Deleted files count	
Created mutexes count	

Table 8: List of *Filesystem Features*

Name	**Type**
Contacted 'com' domains count	Integer
Contacted 'org' domains count	
Contacted 'net' domains count	
Contacted other domains count	
HTTP GET requests count	
HTTP POST requests count	
Other HTTP requests count	
Contacted hosts count	
Hosts subnet	
udp packets count	
udp source ip subnet	
udp destination ip subnet	
irc packets count	
irc source ip subnet	
irc destination ip subnet	
smtp packets count	
tcp packets count	
tcp source ip subnet	
tcp destination ip subnet	
dns packets count	
icmp packets count	
udp source port average	Real
udp destination port average	
tcp source port average	
tcp destination port average	
User-Agent	String

Table 9: List of *Network Features*