EC2: Ensemble Clustering and Classification for Predicting Android Malware Families

Tanmoy Chakraborty, Fabio Pierazzi and V.S. Subrahmanian

Abstract—As the most widely used mobile platform, Android is also the biggest target for mobile malware. Given the increasing number of Android malware variants, detecting *malware families* is crucial so that security analysts can identify situations where signatures of a known malware family can be adapted as opposed to manually inspecting behavior of all samples. We present `EC2` (**E**nsemble **C**lustering and **C**lassification), a novel algorithm for discovering Android malware families of *varying sizes* – ranging from very large to very small families (even if previously unseen). We present a performance comparison of several traditional classification and clustering algorithms for Android malware family identification on DREBIN, the largest public Android malware dataset with labeled families. We use the output of both supervised classifiers and unsupervised clustering to design `EC2`. Experimental results on both the DREBIN and the more recent Koodous malware datasets show that `EC2` accurately detects both small and large families, outperforming several comparative baselines. Furthermore, we show how to automatically characterize and explain unique behaviors of specific malware families, such as `FakeInstaller`, `MobileTx`, `Geinimi`. In short, `EC2` presents an early warning system for emerging new malware families, as well as a robust predictor of the family (when it is not new) to which a new malware sample belongs, and the design of novel strategies for data-driven understanding of malware behaviors.

Index Terms—Android, malware, ensemble, classification, clustering.

1 Introduction

With over 1.4 billion active phones worldwide [1] and over 290,000 phones sold in Q1 2016 alone with an 84.1% market share [2], Android dominates the mobile market. But this success has a dark side - more than 97% of mobile malware target Android devices [3] in order to steal money or private data. Examples include illegally sending SMSs to premium-rate numbers, stealing calendars, emails, texts, contact lists, social network accounts, documents and banking credentials. To elude detection, malicious developers use *obfuscation* techniques (e.g., polymorphism and metamorphism) [19] to automatically generate multiple variants of the same malware, thus creating a new *family* [17] of malware samples having the same purpose but slightly different characteristics (e.g., different file hash, names of functions and variables). In fact, almost 9,000 new Android malware samples were found daily in 2016, an increase of 40% over 2015 [8].

Given the increasing number of new malware samples, manual investigation is impractical and might lead to significant delays in the release of detection signatures for anti-malware tools. The only viable way to manage such huge volumes of malware is to design novel algorithms that can automatically group similar samples into families. This has several major benefits – (i) if a sample belongs to a known family, the same removal techniques can be re-used; (ii) security analysts can focus their manual investigation on the few new samples that do not belong to any known family, thus optimizing their limited time and resources; (iii) understanding the characteristics of each family helps to detect more robust signatures for anti-malware tools.

Most literature on mobile malware analysis [11], [14], [21], [48], [58] is focused on *detection* (i.e., distinguishing malware from benign software). Several efforts [28], [29], [30], [55], [56], [58], [59] have been proposed in the context of detecting Android malware families. Some of these works consider outdated datasets (e.g., [30], [59]) and/or do not characterize malware families through feature explanation (e.g., [28]), mostly because they use features that are low level and hard to interpret, such as API call sequences. However, the major limitation of all existing works [28], [29], [30], [55], [56], [58], [59] is that they focus only on large families (e.g., size > 10) for which training data is available, and ignore small families which represent the novel malware variants on which security analysts should focus their attention.

We propose `EC2`, the first algorithm that effectively classifies a malware sample into both large and small families (even if previously unseen). We begin by presenting a thorough performance comparison on the classification of Android malware families[1] by using DREBIN [4], [14], the largest publicly available Android malware dataset with labeled families. Several state-of-the-art classification and clustering algorithms are evaluated for the task of family classification by considering different combinations of *static*

• *T. Chakraborty is with Dept. of Computer Science and Engineering, Indraprastha Institute of Information Technology, Delhi (IIIT-D), India. F. Pierazzi is with the Department of Engineering "Enzo Ferrari", University of Modena and Reggio Emilia, 41125 Modena, Italy. V.S. Subrahmanian is with Department of Computer Science, Dartmouth College, Hanover, NH 03755, USA. E-mail: tanmoy@iiitd.ac.in, fabio.pierazzi@unimore.it, vs@dartmouth.edu*

Manuscript received November, 2016; revised July, 2017.

1. It is important to observe that in some cases the concept of "malware family" may be fuzzy, as a security analyst may consider that a sample belongs to more than one family. For this reason, all results in this paper refer to the publicly available and labeled DREBIN [4] and Koodous [9] datasets.

and *dynamic* features. We then design EC2 as an ensemble that combines the best of classification and clustering, and evaluate its performance both on DREBIN [4] and on the more recent Koodous academic dataset [9], outperforming several comparative baselines [30], [42], [52], [60].

The paper makes five major contributions.

(i) We thoroughly assess performance of several state-of-the-art supervised classification and unsupervised clustering algorithms for Android malware family identification. The best supervised classifier (Random Forest) has accuracy[2] of 0.93 for large families (size ≥ 10), which decreases to 0.73 for small families (size < 10) due to lack of training data. In contrast, the best unsupervised clustering method (DBSCAN) achieves accuracy of 0.91 for small families and 0.86 for large families (see Table 5).

(ii) While past research has looked at clustering and classification separately for traditional (mostly PC) malware, EC2 combines the results of both clustering and classification to overcome the drawbacks of each. EC2 delivers significant performance across all malware families (including families with just one sample), achieving an overall accuracy of 0.97, and outperforming several comparative baselines on DREBIN.

(iii) Unlike all related works [28], [29], [30], [55], [56], [58], [59], EC2 is the first algorithm that can classify malware samples into small or even *previously unseen* Android malware families. Given the huge number of malware released in the wild everyday [8], the capability of detecting small and previously unseen malware families is critical for prioritizing manual inspection of new threats. We show how we identify samples belonging to very small malware families - in fact showing 0.43 Precision and 0.31 Recall for detecting families of size just 1.

(iv) We show how to characterize different malware families by extracting family-specific features that distinguish one family from others. We chose to use the DREBIN dataset as each sample is labeled with a ground truth family to which the sample belongs [4]; our analysis reveals that the most important features for malware family classification are: re-using signatures for signing malware variants, requesting network permissions, requesting permissions to read/send SMS messages (used to send texts to premium-rate numbers) and use of encryption (often used for string decryption or repacking of malicious code to avoid static analysis). We also show that some malware families are better identified through static features, while others require dynamic analysis (e.g., because they adopt string decryption at runtime of CnC server URLs and CnC commands to avoid trivial detection via static code analysis). We observe that more recent malware families may show different behaviors trends [55] as obfuscation strategies become more sophisticated and adapt to anti-virus strategies; however, the approach we propose for family characterization is general and can also be applied to more recent malware samples.

(v) To show that EC2 is equally efficient in detecting recently released malware samples, we use Koodous, a dataset (Dec 2015 to May 2016) of recent Android malware samples. Here too, EC2 outperforms other baselines with an overall F-Score of 0.74.

The rest of the paper is organized as follows. Section 2 presents a thorough literature review on malware detection and classification. Section 3 presents the DREBIN dataset. Section 4 describes static and dynamic features for malware samples. Section 5 shows how classification and clustering leads to our new EC2 algorithm. A detailed evaluation of the classification and clustering algorithms are separately presented in Section 6 and Section 8 respectively. Section 7 shows how to use classification to automatically characterize malware families. Finally, Section 9 compares EC2 with several existing baselines, including the best classification and clustering algorithms and shows that EC2 outperforms all. Section 10 presents conclusions and directions for future works.

2 Related work

We identify and discuss three main areas of related work: (i) malware spread and characterization, (ii) malware analysis and detection, (iii) prediction of malware families.

Malware spread and characterization. Some broadly related works predict malware spread [40], [62], evaluate infection rates and risk indicators [57], characterize of malware on different third-party Android markets [45] etc. However, these works do not propose ways to detect/classify malware into into families.

Malware analysis and detection. Several efforts ([14], [15], [41], [44], [54], [61]) focus on *malware detection*, i.e., given a new sample, predict whether it is benign or malicious. Kolter et al. [41] propose an n-gram approach based on the bytes of malware binaries. Baldangombo et al. [15] propose an ensemble classifier relying on static features from Windows malware. Siddiqui et al. [54] compare some supervised classifiers using static features and achieve best performance with decision trees. Ye et al. [61] combine file content and file relations for malware detection. While these works focus on PC malware, malware detection for *mobile* devices [53] has gained much recent attention. Some works [22], [36], [46], [51] study the effectiveness of existing antivirus solutions against obfuscation techniques applied to malware, whereas our main focus in this paper is related to family prediction algorithms. Some papers [35], [50] proposed distillation and re-training on adversarially crafted samples to increase classifier robustness against obfuscation; but this approach has been later shown to be not effective [23] through a quantitative evaluation. Other works like DREBIN [14], Crowdroid [21], DroidAPIMiner [11] perform malware detection through static features of Android applications, whereas TaintDroid [32] and DroidScope [58] use dynamic features extracted from execution traces. DroidSIFT [63] performs semantics-aware classification based on control flows. MARVIN [44] proposes a binary classification method to discriminate between benign and malware applications using both static and dynamic features. MaMaDroid [49] builds behavioral models from dynamic logs through HMM to distinguish between benign and malicious applications. Unlike malware detection, our paper focuses on the problem of identifying malware families. Moreover unlike most past work we present a thorough analysis

2. We report accuracy in terms of Macro F-Score (MaF*), formally defined in Section 8.

of several supervised classification and unsupervised clustering methods with different combinations of static and dynamic features.

Prediction of malware families. Several efforts ([17], [38], [52], [60]) consider the problem of *malware classification*, i.e., given a set of malware samples, identify which samples are variants of the same malware *family*. Ye et al. [60] propose an ensemble based on k-medoids and hierarchical clustering. Bayer et al. [17] propose a scalable method combining hierarchical clustering and locality-sensitive hashing on malware execution traces. Rieck et al. [52] propose a supervised SVM-based approach that clusters dynamic features from execution traces. All these works are focused on traditional PC malware, whereas our focus is on Android [53], an ecosystem that differs from traditional PC malware in terms of application, sandboxing, resource access, and system calls (more details in Section 4.1). Some recent work tackles the problem of Android malware classification. [16] proposes a very preliminary model checking approach to classify malware samples from just 2 families (`OpFake`, `DroidKungFu`) with 100 samples each. We consider 156 families including 47 singleton families with about 5000 samples. Dendroid [56] proposes a text-mining based method to find similarities in malware families by extracting code chunks and deriving static features from them. DroidLegacy [30] proposes an approach to detect malicious code modules, but their focus is explicitly on piggybacked and repackaged applications. However, these works [30], [56] have some major shortcomings – they evaluate their approach only on the outdated MalGenome dataset [6], [64] (which is only a small subset of DREBIN, consisting of about 20% of the DREBIN dataset samples), and results obtained with their approach are hard to interpret for a human security analyst, whereas we automatically identify specific characteristics that distinguish malware families and which are easy to interpret. Moreover, past works do not consider small families, which is one of the major strengths of `EC2`. In [30] the authors remove all families with ≤ 10 samples, while in [56] the authors remove all singletons (i.e., families with just one sample) from the analysis. There are other works related to mobile malware classification. DroidMiner [59] analyzes fine-grained behaviors of Android malware applications. DroidScribe [28] considers only dynamic features and classifies Android malware into families to provide a baseline against proposals that focus on static features. DroidSieve [55] considers only static features – in particular it proposes a set of features that are resilient to modern obfuscations strategies. Prescience [29] builds on DroidSieve and studies periodic retraining of classifiers over time to adapt to novel obfuscation techniques. However, in this paper we also consider small and previously unseen families which are not the main focus of [28], [29], [55], [59], [63]; in particular, our major contribution is the `EC2` algorithm which is very efficient in detecting both very small (including new malware families) and large families. Andronio et al. [12] propose a method for discriminating between goodware, scareware and ransomware by generalizing three key insights of common behaviors in mobile ransomware: threatening text, device locking, and encryption. Aresu et al. [13] propose an approach that by extracting features from HTTP traffic is focused on classificaton of mobile botnets variants (e.g., `Zitmo`). However, these works are not generic since they are targeted to specific objectives (identify which malware is scareware/ransomware, and identify variants of mobile botnets samples), whereas we propose a generic approach that can be applied to any malware family and is also able to detect singleton families.

As a final remark, it is important to observe that in some cases the concept of "malware family" may be fuzzy, as a security analyst may consider that a sample belongs to more than one family. For example, consider the `Petya` ransomware and its more recent variant `NotPetya` [10], which antivirus vendors eventually decided to consider as two distinct families due to their differences in propagation and operativity. Nevertheless, the detection of small and/or previously unseen families can also be helpful in the detection of very different variants of a known family which may represent an entirely new strain. For the sake of fairness, all results in this paper refer to the publicly available and labeled DREBIN [4] and Koodous [9] academic datasets.

3 DREBIN Dataset

We first analyze the DREBIN dataset [4] as it represents the largest public, labeled mobile malware dataset, as of 2016. All DREBIN samples (even ones in the same family) are characterized by different *hash* values, indicating that some obfuscation technique [19] has been applied in order to prevent easy detection of variants. As described in Section 4.3.1, we execute the malware samples in a controlled environment to extract dynamic features. We only kept dynamic logs for samples executed for 120 seconds without failures. In particular, the final dataset used in this paper consists of $4,845$ malware samples: Figure 1(a) reports the distribution of malware family sizes, and Figure 1(b) reports the top 15 malware families by size, among which: `FakeInstaller`, that simulates an installer of Android applications but in reality sends SMSs to a premium-rate service; `DroidKungFu` that tries to perform privilege escalation and steals sensitive data from the device; `Opfake` that is another malware family sending SMSs to premium-rate numbers.

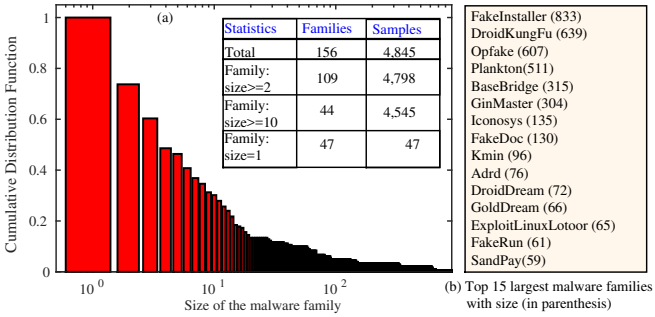
The size of Android malware families follows a heavy-tailed skewed distribution (cf. Figure 1(a)): 112 of 156 families have under 10 samples. This suggests that supervised classification algorithms might not work well due to lack of training samples for small families. Hence, we leverage unsupervised clustering approaches and propose a novel ensemble method that leverages both classification and clustering.

4 Feature extraction

We first present some fundamentals about the Android environment, and then describe how we design and extract static and dynamic features from the samples.

4.1 Android Fundamentals

Android applications (or `apk` files) are *jar*-like compressed files. Android users can install applications from both official (*Google Play Store*) and third-party marketplaces. The *components* of each Android application must be declared in the `Manifest` file which is a header in each apk. The main `apk` components are:



(b) Top 15 largest malware families with size (in parenthesis)

Fig. 1: (a) Cumulative distribution of the size of malware families present in the DREBIN dataset (see Supplementary for the non-cumulative family distribution), (b) top-15 malware families by decreasing size.

- *Activities* model the screens and windows that constitute the user interface of an application.
- *Services* are background processes that remain active even when the application is out-of-focus (e.g., playing music), and may be used for malicious acts (e.g., aggressive advertisement from Adware).
- *Content Providers* are used to define and regulate data sharing between applications.
- *Broadcast Receivers* are used to monitor system-level events (such as `BOOT_COMPLETED`, commonly monitored by malware to trigger malicious behavior on phone startup [14]).
- *Intents* are messaging objects that can be used for interactions and communication between components. They are used to invoke *actions*, e.g., start another activity, start a service, or deliver a broadcast message that can be captured by broadcast receivers.

Each Android application runs in an *isolated* environment that is separated from other applications, and by default it can only write on a separate memory space and cannot access phone resources (e.g., Camera). To perform more advanced interactions, Android relies on a fine-grained *permissions system* for which an application has to explicitly request the user, at installation time, for permission to access specific *software* (e.g., access call log) and/or *hardware* (e.g., camera, vibration) resources. This guarantees transparency about the data that can be accessed by the application (e.g., access and modify Calendar or Contact List), although it is often overlooked by users since the number of permissions may be high. Moreover, it is not trivial to determine what an application actually does with a permission (e.g., when/where an app exactly uses `READ_SMS` permission to read text messages inbox). A full set of standard permissions is available in Android documentation[3], but program developers can also define a set of *custom permissions* depending on their needs (e.g., to communicate with other applications of the same developer).

4.2 Static Features

Several static features have been proposed in the literature (e.g., [14], [44], [55], [63]). We wanted to develop easy to extract and explainable static features by leveraging those proposed in [14]. Despite [14] extracts static features both from the Manifest and from the source code (e.g., API calls), we decided to consider only features from the Android Manifest[4] for three main reasons: (i) features from the code may introduce overly detailed and noisy information [31], [36], whereas the Android `Manifest` already has rich information about an application and its structure; (ii) the use of encryption or reflection [46] may easily introduce much noise in the code, whereas the Android Manifest must be declared in plaintext and also contains many specifications about requested permissions and interfaces; (iii) finally, we consider features that are easy to interpret, so that we can automatically extract meaningful characteristics that distinguish malware families (cf. Section 7 and supplementary materials).

TABLE 1: Static features derived from the analysis of code and `Manifest` in the malware samples in DREBIN.

Group	Feature	Description
Author	`author`	Derived from SHA256 hash that digitally signs the Android sample
Structure	`filesize`	The size of the file in bytes
	`n_activities`	Num. Activities
	`n_intents`	Num. Filtered Intents
	`n_providers`	Num. Content Providers
	`n_services`	Num. Services
	`n_receivers`	Num. Broadcast Receivers
Permissions	`n_std_sw_perm`	Num. standard sw permissions required to install the application
	`n_std_sw_ _perm_dangerous`	Num. std sw permissions marked as dangerous in Android documentation
	`n_hw_perm`	Num. std hw permissions required to install the application
	`n_custom_perm`	Num. custom permissions (i.e., non-std) defined by the application developer
	$\texttt{sw\_permission}_1$	1 if $\texttt{sw\_permission}_1$ is required by the application, 0 otherwise
	...	1 if $\texttt{sw\_permission}_i$ is required by the application, 0 otherwise
	$\texttt{sw\_permission}_N$	1 if $\texttt{sw\_permission}_N$ is required by the application, 0 otherwise
	$\texttt{hw\_permission}_1$	1 if $\texttt{hw\_permission}_1$ is required by the application, 0 otherwise
	...	1 if $\texttt{hw\_permission}_j$ is required by the application, 0 otherwise
	$\texttt{hw\_permission}_K$	1 if $\texttt{hw\_permission}_K$ is required by the application, 0 otherwise

Depending on their meaning, our static features fall into three main groups: *author*, *structure* and *permissions*. Table 1 reports the complete list of static features, where the leftmost column reports feature groups.

Author. Malware of the same family may be developed by the same *author*. Developer information is usually published by marketplaces (e.g., Google Play). However, the DREBIN dataset also contains applications that have been removed from the store or that have been retrieved from third-party marketplaces. We assume that if two applications are digitally signed with the same certificate, then they have been developed by the same author. This feature has not been considered by [14], but in some other related research (e.g., [55]). Section 7 shows that some families (e.g.,

3. https://developer.android.com/guide/topics/security/permissions.html

4. Our static feature set corresponds to the features proposed in [14] related to the Android Manifest with some modifications: unlike [14], we also consider the `filesize` and `author` of an app (which have also been considered in [55]); unlike [14], we only consider the *number* of application components (e.g., Activities, Services), instead of their specific class *names* (e.g., `HomePageActivity.class`)– this is because names can be very easily obfuscated, whereas number of components may not be modified arbitrarily by the malware developer without raising suspiciousness in malware detectors. We also observe that all static features in [14] are binary, whereas we also have continuous static features: author, filesize, number of components, and number of permissions.

MobileTx and Opfake) have many samples developed by the same author – while the author feature is insignificant for others (e.g., Geinimi, FakeInstaller, JiFake) as the malware developers were more careful.

Application components. Applications sharing similar structure may be variants of the same family. We describe app structure via the number of components found in the Manifest: filesize, n_activities, n_intents, n_providers, n_receivers, n_services. As an example, the feature n_activities is very significant to discriminate the family Opfake (cf. supplementary materials), and also for discriminating the various families in the multi-class classification task (cf. Figure 8). Unlike the feature set presented in [14], we choose not to consider the *names* of the Android components, but rather their counts because component names can be altered very easily (as they are just the names of Java classes defined by the application developer). On the other hand, the number and organization of components is harder to change significantly without also improving chances of detection of a sample as malware.

Permissions. The Android permissions system provides a rich source of features as malware from the same family may need the same permissions that may help distinguish them from other families. For example, MobileTx is one of the few families requiring the RESTART_PACKAGES permission, that is used to kill antivirus and application monitoring systems. Sometimes, even the absence of a certain permission requested by the malware can be useful to discriminate its family. For instance, FakeInstaller does not need the ACCESS_NETWORK_STATE permission— this absence is a strong indicator that a sample belongs to this family. A predefined set of *standard permissions*[5] label some features *dangerous* as they provide access to sensitive resources such as call logs or contact lists. Our feature set includes binary vectors as in [14], containing possible standard software and hardware permissions from the Android official documentation. We also count the number of customized permissions that can be defined by programmers. As a final remark, although permission-related features are very important, they may not be sufficient by themselves for detecting/classifying some types of Android malware (e.g., in case of privilege escalation attacks [20]); hence it is fundamental to consider the other kinds of features presented in this section as well.

We extract static features with official *Android SDK*[6] tools and *Androguard*[7], a Python library for extracting metadata from apk files and their Manifest. We have 190 static features in all (Author: 1, Application components: 6, Permissions: 183).

4.3 Dynamic Features

Since static features alone may not be enough because of obfuscation techniques [19], we run malware samples in a sandbox in order to find common behaviors exhibited by families [52]. We first describe generation of dynamic logs, then design and motivate extraction of dynamic features.

4.3.1 Generation of Malware Dynamic Logs

We installed and configured the official Android emulator[8], a fully-functional Android system that runs applications with a graphical user interface. We also used inetsim[9], a tool that simulates many network services, and also provides realistic data in response to malware requests (e.g., if a malware tries to download an executable file from an external website). This allows us to log the most relevant malware Internet requests of the malware, despite absence of real Internet connectivity. To run the malware samples, we use DroidBox 4.1.1[10], an open-source sandbox especially tailored for dynamic malware analysis of Android applications. DroidBox can install and execute apk applications in the Android emulator. For each malware sample in DREBIN/Koodous, we execute the malware for *120 seconds* and log all its activities through DroidBox, which include: file system activities (read/write), network activity (send-net/recvnet), usage of cryptographic primitives, dynamic loading of classes, start of new services (background processes), generation of system events. We did not use simulated user input because: (i) a *random* input simulator would prevent deterministic code execution, which is important for malware classification; (ii) designing a *deterministic* input simulator to be run on all applications would be a huge challenge as many malware applications *crash* often.

```
fdaccess: {
1.9002759456634521: {
data: 541545301efbfbd7a2a57,
id: 1590266696,
operation: write,
path: /data/com.app/stats.log,
type: file write},...
}
```

(a)

```
sendsms: {
12.9806270599365234: {
message: "Thanks for
downloading Xmas walls!",
number: <phone-number>,
type: sms },
...
}
```

(b)

Fig. 2: Example of (a) write-log and (b) SMS activity log generated by DroidBox.

Figure 2(a) shows a sample Droidbox log of a filesystem *write* operation, where about $1.9s$ after starting the execution of the malware, some data is written in /data/com.apsp/stats.log. The "data" field in Figure 2(a) contains the data written by the application in hexadecimal format. Figure 2(b) shows another sample log related to SMS sending activity. Many malware try to deceptively send SMSs to premium-rate numbers owned by the malware developer in order to get money from the user. Sometimes, SMSs are also used to spread the malware by sending a text to users from a contact list to deceive them into clicking a malicious URL. Details of the dynamic log generation process can be found in the supplementary materials.

4.3.2 Dynamic Feature Extraction

Our goal is to design features that capture similarities in malware behaviors in order to classify samples into families. Dynamic logs allow us to even design very low-level features, such as read operations on specific data from files

5. https://developer.android.com/guide/topics/security/permissions.html
6. https://developer.android.com/studio/
7. https://github.com/androguard
8. https://developer.android.com/studio/run/emulator.html
9. http://www.inetsim.org
10. https://github.com/pjlantz/droidbox

TABLE 2: Dynamic features derived from the execution of the Android malware samples using bag of words.

Group	Feature	Description
RW	`read [[<basepath>] [<filename>]]`	N-gram counts about `read` operations on the Android filesystem. The `basepath` is just the name of the first folder after the root
	`write [[<basepath>] [<filename>]]`	N-gram counts about write operations on the Android filesystem. The `basepath` is just the name of the first folder after the root
System	`servicestart [<servicename>]`	N-gram counts about started background processes (i.e., `services`)
	`load [[<classpath>][<classname>]]`	N-gram counts about `dex` Android classes loaded during execution
	`crypto [[<algorithm>] [<encryptionkey>]]`	N-gram counts about `crypto` operations performed during execution
	`recvsaction [[<path>] [<name>]]`	N-gram counts about event listeners (e.g., `BOOT_COMPLETED`)
Network	`sendnet [[<protocol>] [<port>]]`	N-gram counts about outgoing network activity
	`recvnet [[<protocol>] [<port>]]`	N-gram counts about incoming network activity
SMS	`sendsms [[<phonenumber>] [<message>]]`	N-gram counts about `sms` sent by the application

and folders at particular timestamps. However, past work shows that overly detailed dynamic features rarely improve malware classification [28], [31] as they inject noise. We therefore leverage an n-gram representation commonly used in malware analysis and classification (e.g., [18], [31], [33], [52]). Using n-grams, we *count* the following operations captured by the sandbox: `RW` (read and write operations), `System` (e.g., start of a new background process), `Network` (Internet requests), `SMS` (texts sent by the device). For each dynamic log operation, we extract several possible sets of words to limit the impact of possible obfuscation strategies adopted by the attacker (e.g., changing the filename of a written file). For this purpose, we consider a *bag of word* approach in which we extract n-gram counts. In particular, we have experimentally verified that considering n-grams with $n > 3$ does not yield any performance improvement (because the level of details increases and features become too specific, as discussed in [28], [31]), so we consider unigrams, bigrams and trigrams. Each word is a feature, where the value is represented by the number of occurrences of that keyword in the logs [52]. The idea is that similar malware execute similar types of operations.

To clarify how the bag of words for dynamic features are extracted, consider an example based on the `write` log of Figure 2(a). Since we consider up to 3-grams, we can extract the following four words from this log: (i) `write`, (ii) `write /data/`, (iii) `write stats.log`, (iv) `write /data/stats.log`. The first word corresponds just to the name of the executed operation. In the second word, we extract only the *basepath* `/data/` instead of the *fullpath* `/data/com.app/` (i.e., the first folder name in the path, in addition to the root folder) because most Android folders are dependent on Application names (e.g., `com.app` in the example) or process ids, but the basepaths are unique [31]. We do not include the content written – we verified that it is not useful for malware classification, as many malware read files in different sequences (e.g., a 256 byte file read 1 or 2 bytes at a time), and also use encryption strategies to hide the real read/written content. Note that feature values represent *number of occurrences* of a word in the dynamic log of the malware sample.

We can then define a generalized version of a write operation feature set as follows:

$$\texttt{write}\left[\,[<\texttt{basepath}>]\,[<\texttt{filename}>]\,\right] \qquad (1)$$

where items between squared brackets are optional (hence, if one considers all the possible combinations she can obtain four words).

Table 2 reports the full list of dynamic feature types extracted with the bag of words approach. The rationale behind the design of these features is to capture similarities in malware behaviors by counting the number of high-level actions of each sample [31]. Section 7 shows that dynamic features are effective in characterizing different malware families (cf. supplementary materials): for example, `Opfake` loads a malicious `apk` in memory at runtime through the `dexclass` loading function; `MobileTx` sends SMS to premium-rate numbers; `Geinimi` starts a fake background process called `GoogleKeyboard` that collects user information that are then sent outside.

Note that since we always consider 1-gram counts consisting of just the operation (e.g., `write`, `read`), our approach also captures the total number of dynamic operations of each kind executed by a malware.

By executing DREBIN malware samples, we collect $6,875$ dynamic features corresponding to the set of possible words of Table 2. We then drop all the features that have value 0 (i.e., never occurred) for all malware samples except one. In other words, we drop all n-grams executed by just a single malware sample in the dataset – as they do not contribute to the malware classification task; thus we are left with $2,048$ dynamic features.

5 Discovering Android Malware Families

5.1 Supervised Classification

We consider six standard supervised classifiers – Decision Tree (DT), K-Nearest Neighbors (K-NN), Logistic Regression (LR), Naive Bayes (NB), Support Vector Machine (SVM), and Random Forest (RF). We perform hyper-parameter optimization in order to find the parameters that generate the best results. For instance, we use CART with Gini gain criteria for DT; K-NN method with $K = 5$; multinomial logistic regression and SVM with linear kernel. Section 6 shows that Random Forest turns out to be the best classifier in general, buts fails to capture small families. This motivated us to run unsupervised clustering algorithms with the hope that interdependency between malware samples captured via clustering might help in detecting small families.

5.2 Unsupervised Clustering

We consider five well known clustering methods previously used in malware analysis: [17], [26]: DBSCAN, Hierarchical with complete linkage and Euclidean distance, Affinity, K-Means and MeanShift. The value of K in K-Means was determined by the Silhouette Method. Other parameters were systematically tuned to get the best performance.

Section 8 shows that DBSCAN turns out to be the best clustering algorithm. Although it accurately captures small families, its overall performance is significantly worse than

the best classification algorithm. This further motivates us to combine both the results of classification and clustering in a systematic way to improve the overall performance as well as to detect small families.

5.3 EC2: Combining Classification and Clustering

Section 6 will show that while classification methods perform well on families with at least 10 samples, they fail to predict small families effectively. On the other hand, clustering methods efficiently capture small families. To get the best of both worlds, we propose EC2 (**E**nsemble **C**lustering and **C**lassification) which combines the results of classification and clustering in a systematic way (see Algorithm 1).

In the initial steps, EC2 uses an unsupervised clustering method A_{uc} to cluster all the samples (Step 1). In parallel, it trains a supervised classifier A_{sc} on training set TR (Step 2) and measures the membership probability of each test sample for each family (Step 4). If a test sample achieves a maximum membership probability greater than a certain threshold δ_1, it is assigned to the corresponding family (Step 8); otherwise it is marked as "unlabeled" (Step 10). In Section 9.2, we vary the value of δ_1 and observe that the highest accuracy is obtained with $\delta_1 = 0.6$. For each such unlabeled sample s, EC2 first checks the cluster C_s where it is assigned by A_{uc} (Step 13). If more than δ_2 fraction of the constituent members in C_s have already been labeled with *a single family* f, then s is labeled with f (Step 15); otherwise s is labeled with a new family C_s. Note that in this step if there is a tie in the size of the classes inside the cluster C_s, we by randomly assign the sample into one of the majority classes[11]. Moreover, Step 17 enables us to create a completely new family which may not be present in the training set, thus allowing us to identify completely unobserved families.

Illustrative Example: Figure 3 shows an example of EC2 in action. Suppose there are six samples $O1, \cdots, O6$. A classification algorithm A_{sc} classifies the samples with a probability distribution as shown in Figure 3(a). If $\delta_1 = 0.7$, then samples $O1$ and $O3$ are assigned to class $C1$ because the membership probability of both these samples is above 0.7 (Figure 3(b)). No other samples are assigned to a class due to the lack of high confidence. Therefore, we run a clustering algorithm A_{uc} that may group them into three clusters $G1, G2, G3$ (Figure 3(c)). For the unassigned samples we use the clustering results as follows. For instance, in the case of $O2$ we see that it belongs to cluster $G1$, and there are two other samples $O1$ and $O3$ which also belong to $G1$. Now, out of 3 samples in $G1$, two are already labeled as $C1$ from the classification result, i.e., more than δ_2 (which is set as 0.6) fraction of samples in $G1$ are labeled as $C1$. Therefore, $O2$ is also assigned to $C1$. However for $O5$ and $O6$ which belong to $G2$ and there is no other member labeled earlier, we assign them separately in a class $G2$. Similarly $O4$ is assigned to a singleton class $G3$.

Time Complexity: Assume that there are n malware samples and f families in the test set. Once we obtain the results of the classification and clustering algorithms, for each sample it takes $\mathcal{O}(f \log f)$ time to obtain the maximum membership probability. Therefore, the total time required to execute Steps 3-10 is $\mathcal{O}(nf \log f)$. Further if we assume there are on average n_c samples per cluster, the worst case total cost incurred by EC2 in Steps 11-17 is $\mathcal{O}(n.n_c) \sim \mathcal{O}(n^2)$. However, in practice $n_c \ll n$, which implies that $\mathcal{O}(n.n_c) \sim \mathcal{O}(n)$. So the overall worst-case time complexity is $\mathcal{O}(n + nf \log f) \sim \mathcal{O}(nf \log f)$.

11. Note that if δ_2 is greater than 0.5, there is no possibility that a tie is encountered.

Input: Training set: TR, test set: TS, thresholds: δ_1 and δ_2, Classification algo: A_{sc}, Clustering algo: A_{uc}
Output: Labeled families of TS

1 Run A_{uc} on $TR + TS$ and obtain set of clusters $\mathbb{C}$;
2 Train A_{sc} on TR;
3 **for** *each* $s \in TS$ **do**
4 $\quad MP[s, f] \leftarrow$ Membership probability of s in family f obtained from A_{sc};
5 $\quad mp^* \leftarrow \max_{\forall f} MP[s, f]$;
6 $\quad f^* \leftarrow \underset{f}{argmax}\ MP[s, f]$;
7 $\quad$ **if** $mp^* \geq \delta_1$ **then**
8 $\quad\quad L[s] \leftarrow f^*$; ▷ Assigning family label of s
9 $\quad$ **else**
10 $\quad\quad Push(UnlabeledS, s)$ ▷ Push s into $UnlabeledS$
11 **while** $UnlabeledS$ *is not empty* **do**
12 $\quad s \leftarrow Pop(UnlabeledS)$;
13 $\quad C_s \leftarrow$ Cluster of s (where $C_s \in \mathbb{C}$);
14 $\quad$ **if** *More than* δ_2 *fraction of the samples in* C_s *are labeled with a family* f **then**
15 $\quad\quad L[s] \leftarrow f$;
16 $\quad$ **else**
17 $\quad\quad L[s] \leftarrow C_s$;
18 **return** L;

Algorithm 1: EC2 algorithm

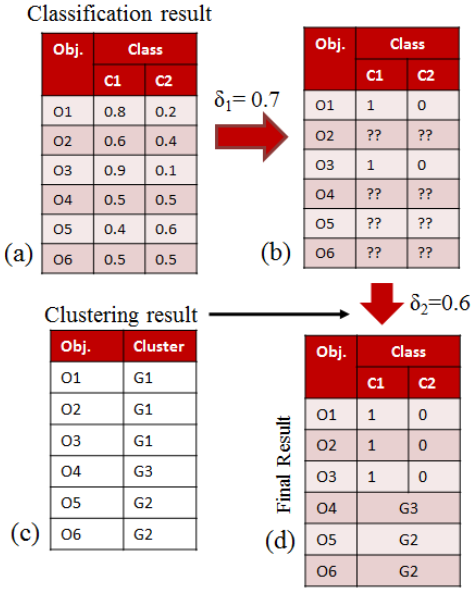


Fig. 3: An illustrative example EC2. Six samples $O1, \cdots, O6$ are finally grouped into three clusters $C1, G2, G3$.

6 Performance of Supervised Classifiers

In this section, we provide the performance of the classifiers. We start with the metrics used to evaluate the performance of the classifiers, followed by a comparative evaluation.

6.1 Evaluation Metrics

As the size distribution of malware families is highly skewed (cf. Figure 1(a)), we examine the performance of the classifiers at Macro (Ma) and Micro (Mi) levels. At each level, we consider Precision (P), Recall (R), F-Score (F) and Area under the ROC curve (AUC). At micro (*resp.* macro) level, Precision, Recall and F-Score are denoted as MiP, MiR and MiF (*resp.* MaP, MaR and MaF) respectively. Each of such metrics ranges from 0 (no match) to 1 (exactly similar).

We recall how Micro and Macro statistics are computed for evaluating multi-class classifiers. Given a classifier and a malware family i, we let TP_i, FP_i and FN_i indicate True Positive, False Positive and False Negative samples, respectively. Then, for n different families, Micro Precision (MiP) and Macro Precision (MaP) are:

$$MiP = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} TP_i + FP_i} \tag{2}$$

$$MaP = \frac{\sum_{i=1}^{n} \frac{TP_i}{TP_i+FP_i}}{n} \tag{3}$$

We also observe that micro statistics suppress the results of small clusters over those of large clusters.

6.2 Performance Analysis

We observe that Random Forest (RF) achieves the highest overall classification accuracy after 5-fold cross-validation[12] (see Supplementary Materials for the performance of the other classifiers). RF yields a MaF=0.65 and MaAUC=0.83, and MiF=0.93 and MiAUC=0.97 averaged over 50 iterations for *all families* (including the ones with less than 10 samples). Small families lead to lower values for macro statistics because these are difficult to predict due to the lack of training data. Performance of small families will be separately discussed in Section 6.4. For now, we consider only families with size ≥ 10 (44 families, 4545 samples).

Composite Performance: Figure 4(c) shows performance of all classifiers on families of size ≥ 10 considering both static and dynamic features. For better visualization we adopt the setup used in [25] – for each evaluation metric (such as MiF, MaF, MiAUC, MaAUC), we separately scale the scores of the methods so that the best performing method has a score of 1. The composite performance of a method is the sum of the four normalized scores. If a method outperforms all other methods, then its composite performance is 4 (See Supplementary Materials for the actual performance value of the classifiers).

Figure 4(a) shows the composite performance of all the classifiers for different feature sets. Considering both static and dynamic features, Random Forest outperforms all others (with composite performance of 4), followed by DT (3.92), K-NN (2.24), SVM (2.15), NB (1.99) and LR (1.94). The superior performance of DT corroborates the results in [15], [41] for separating malware from benign applications and might be due to the categorical features such as author, structure and permissions. As described in [24], K-NN and DT are extremely useful when the features are categorical and/or a mixture of continuous and categorical. This might also explain the poor performance of SVM. Figure 4(b) presents the relative performance of the classifiers by individually considering dynamic features and both static and dynamic features together w.r.t. the performance with only static features (which were used in most previous mobile malware classification research [30], [34], [56]).

Figure 4(b) reports that for all classifiers the relative performance exceeds one in most cases, showing that dynamic features improve classification accuracy compared to static ones. Although the relative performance of K-NN, LR, NB and SVM is higher with only dynamic features, among all classifiers the best absolute performance is achieved by Random Forest with both static and dynamic features (Figure 4(c)). Moreover, Section 7 will show that adopting both kinds of features is also extremely useful in distinguishing characteristics of each malware family. Therefore, unless otherwise mentioned, we will present results that include both static and dynamic features and that refer to the best classifier (RF).

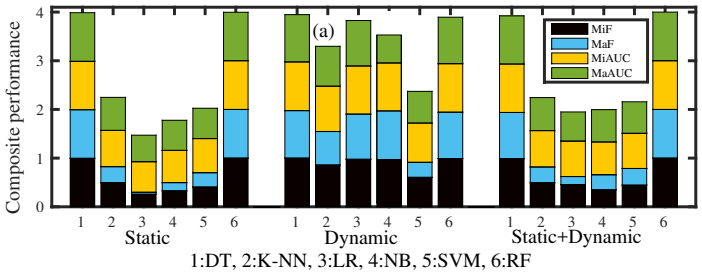
Family-wise Performance: To understand the detailed results of the classifier, Figure 5 shows the family-wise accuracy of Random Forest, the best classifier. We consider all the detected (*resp.* ground-truth) families, sort them by descending order of size, and plot the Precision (*resp.* Recall) for the families with at least 10 samples in Figure 5(a) (*resp.* Figure 5(b)). We observe high Precision irrespective of family size, whereas Recall drops slightly for smaller families. The small performance drop at bin 5 for both Precision and Recall is caused by two malware families, `Boxer` and `SMSreg`, that have some hard-to-detect variants.

6.3 Prediction of Unknown Families

Thus far, out of total $4,845$ samples we have considered $4,545$ samples belonging to 44 large families (families with at least 10 samples). However, there are 300 other malware samples present in a total of 112 small families. In this experiment, we treat these 300 malware samples as "unknown malware samples". In particular, out of $4,545$ samples (with size greater than 10) we randomly choose $4,245$ samples for our training set (set I). Two test suites are prepared for this experiment: (i) Set II: the remaining 300 samples from large families out of $4,545$ sample set, (ii) Set III: 300 unknown malware samples (belonging to 112 small families) whose representatives are absent in the training set. The classifiers are trained on Set I to predict samples in Set II and Set III. The experiment is repeated 50 times to see how the classifier performs especially for Set III. We wish to show that a standard classifier should identify the family of a sample with *high confidence* if the malware belongs to one of the given families. However, if it does not belong to a known family, the classifier should assign it to a known family with *low confidence*.

We use Random Forest with all (static and dynamic) features to estimate the probability of each test sample belonging to different families. Figure 6(a) shows that Random Forest with the proposed feature set is very confident in labeling Set II (the maximum probability is greater than 0.80 for the great majority of test samples; and MaF and MiF

12. We consider 5-fold over 10-fold because the former handles folds with less than 10 samples in our training data. It is required because in our dataset small families are the prevalent families.



Algo	MiF	MaF	MiAUC	MaAUC
DT	**0.94**	**0.85**	**0.97**	**0.92**
K-NN	0.47	0.28	0.73	0.63
LR	0.43	0.14	0.71	0.56
NB	0.33	0.27	0.65	0.62
SVM	0.42	0.30	0.70	0.60
RF	**0.95**	**0.89**	**0.97**	**0.93**

(c)

Fig. 4: Performance for families having ≥ 10 samples averaged over 50 iterations with 5-fold cross-validation on the DREBIN dataset. (a) Composite performance of the classifiers; (b) relative improvement (in terms of Micro AUC) of the classifiers with respect to their performance with only static features (the horizontal dotted red line indicates that both the performance are equal); (c) absolute values of the metrics for all the classifiers considering static and dynamic features together.

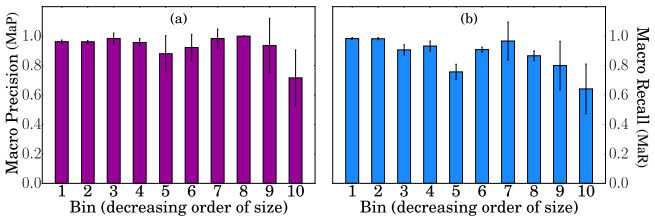


Fig. 5: Family-wise accuracy (mean values of (a) Macro Precision (MaP) and (b) Macro Recall (MaR) and their variance) of Random Forest (RF) with static and dynamic features on the DREBIN dataset. The families are grouped in bins and sorted by descending order of their size. The results are obtained with 5-fold cross-validation and averaged over 50 iterations. The small drop in bin 5 is caused by `Boxer` and `SMSreg` families, that have variants harder to detect.

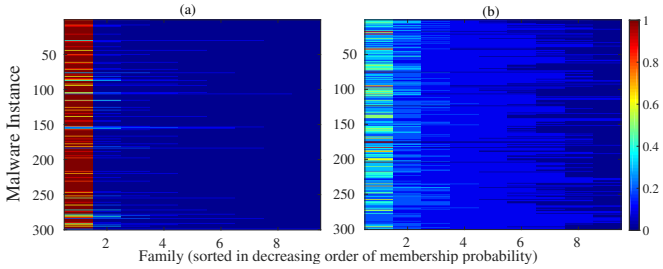


Fig. 6: Family membership probability of test samples (300 samples each from (a) known (Set II) and (b) unknown families (Set III)) obtained from Random Forest with static and dynamic features on the DREBIN dataset. For better visualization, we plot top 10 most probable families per test sample and sort families based on descending order of membership probability.

are 0.96 and 0.91); whereas in Figure 6(b) it is extremely erratic in predicting the labels of Set III. As we have seen in Section 5.3, this behavior of the classifier has been leveraged to design the `EC2` algorithm.

6.4 Performance for Small Families

Thus far, we have primarily considered 44 families with size ≥ 10. However, there are 109 families that have size ≥ 2,

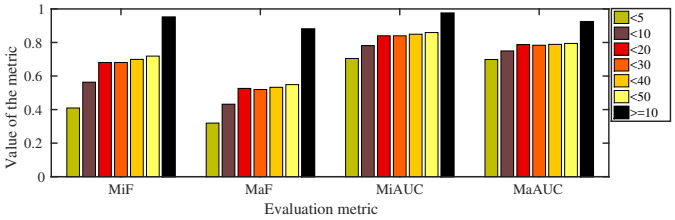


Fig. 7: Performance of Random Forest (RF) for families with different sizes, averaged over 50 iterations with 2-fold cross-validation on the DREBIN dataset. We compare the performance of RF considering only small families (having size less than x, where $5 \leq x \leq 50$) with large families (having size ≥ 10).

and each of the remaining 47 families has only one sample (singleton families).

We now consider all the families of size ≥ 2 and adopt a 2-fold cross validation using stratified sampling so that at least one sample per family is present in the training set. The experiment is repeated 50 times and the average accuracy is reported. We separately measure class-wise performance. Figure 7 shows the performance of Random Forest for families of size less than x ($5 \leq x \leq 50$) in order to see how the classifier performs on such heavily skewed families.

We compare this performance with the performance obtained earlier in Figure 4 for families with at least 10 malware samples. Figure 7 shows that the performance of the classifier drops significantly if we consider only small families in the test set - not surprising due to the small size of the training set. Moreover, although Section 6.3 shows that for samples belonging to unknown families Random Forest is erratic in assigning them into known families, it is not possible for a classifier to identify completely unknown families. This means that singleton families cannot be identified by classifiers. However, because of the huge daily growth in malware variants, it is critical to identify malware families as soon as an application has been detected as malware. This motivates us to study unsupervised clustering algorithms, as presented in Section 8.

7 Characterizing Some Malware Families

A security analyst needs both an accurate predictor and to understand which features discriminate between different

malware families. These findings are useful in engineering robust signatures for detection of new malware variants.

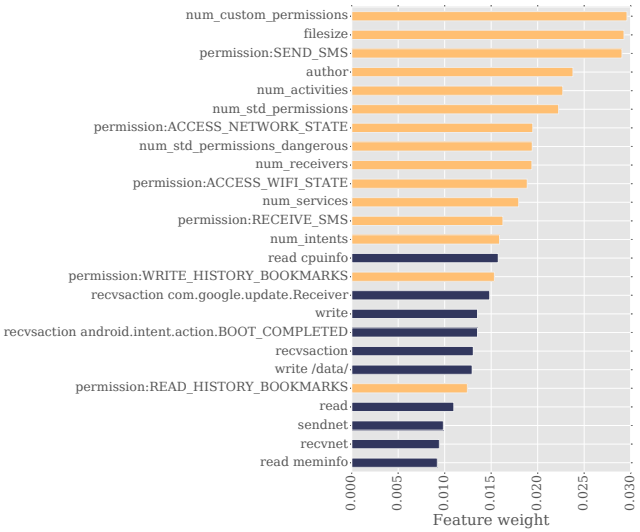


Fig. 8: Top 25 discriminative features of Random Forest multi-class classifier for the DREBIN dataset. Static and dynamic features are represented as *yellow* and *blue* bars, respectively.

Figure 8 reports the top 25 features by decreasing classification weight for the Random Forest multi-class classifier, with static features in *yellow* and dynamic features in *blue* (see Table 1 and Table 2 for detailed features description). We see that static features are more importang than dynamic ones, probably because Android applications have a rich set of information in their `Manifest` (i.e., XML-header), making it more challenging for malware authors to create hugely different variants. Figure 8 shows that the most relevant static features are about *custom permissions*, that can be used for interactions and data sharing between applications with the same author/signature. `filesize` is also relevant, suggesting that variants of the same malware often share similar size distributions. Another important feature is `author`, because malware authors might reuse the same signature to publish many variants of their malicious applications on multiple markets. The most important *dynamic features* include the number of `write` operations on the filesystem and `read cpuinfo`, i.e. the number of read operations of CPU characteristics (to learn details of the device hardware to determine possible exploits). The receiver for `BOOT_COMPLETED` signal is also a top feature – it is often used by malware developers to determine when the device is turned on in order to trigger malicious behavior [14].

Malware family characterization: Apart from the overall discriminative features obtained from the multi-class classification, it is interesting to identify the *specific* features that characterize and distinguish *each* malware family from the rest. To this end, we design the following experimental setup – for each of the 44 large malware families, we learn the best *binary* classifier (Random Forest) to distinguish each family from the rest, such that there exist 44 set of classification rules that separate the behavior of one malware family from all others. In this way, for each family we can sort the features by decreasing *weight* learned by the classifier, thus ranking the features that are more relevant for classification.

Figure 9 shows relevant examples of the top 3 features for families of different sizes.[13] Each row in Figure 9 corresponds to a different family. Each histogram reports feature values on the X-axis, and the percent of malware samples having that feature value on the Y-axis. We report results about the following families[14]:

- `FakeInstaller` pretends to be an app that installs (or uninstalls) other apps from the system, whereas its primary purpose is to send premium-rate SMS without user consent.
- `MobileTx` is a Trojan that both steals information and also tries to send premium-rate SMS.
- `Geinimi` is a Trojan that opens a backdoor to send information from the device to a remote server.

Though some of these families share similar gs (e.g., sending premium-rate SMS), each of them is characterized differently through the top 3 features shown in Figure 9.

FakeInstaller. (discovered in May 2012 [5]) The top 3 features in `FakeInstaller` are static (see first row of Figure 9). The first feature is related to a permission accessing the network state (i.e., whether a connection is available or not on the device at a given time). This feature is important for discriminating `FakeInstaller` as it is one of the few malware families that does *not* require it (i.e., where permission `ACCESS_NETWORK_STATE` is set to 0, as can be observed in Figure 9). The `filesize` of `FakeInstaller` samples is usually lower than the other families. Moreover, the `recvsaction DATA_SMS_RECEIVED` is used to monitor whenever an SMS is received, and is probably used by `FakeInstaller` to remove evidences of its texts sent to premium-rate number and to abort or delete incoming SMS.

MobileTx. (discovered in May 2012 [5]) The second row of Figure 9 reports the top 3 distinguishing features of the `MobileTx` family, an SMS-fraud malware. The most important feature involves `sendsms` activity to the following phone number: `1065-71090-88877`. This turns out to correspond to a premium-rate number used to steal money from the victim [37]. We discovered 68 premium numbers used by DREBIN samples. Some of these texts also have predetermined text content which is captured by our dynamic analysis. Moreover, all variants of `MobileTx` require the `RESTART_PACKAGES` permission that allows the malware to kill all processes including monitors and antivirus software installed in the device. The histogram in Figure 9 shows that some other malware families (blue bar) also require this permission for the same purpose, but it is not common. Finally, all 21 variants of `MobileTx` have the same `author`=443 (that is the anonymized integer for the SHA256 signature), and hence this feature also helps identify this family.

Geinimi. (discovered in Dec 2010 [5]) `Geinimi` variants steal information such as fine location, device IDs (e.g., IMEI, IMSI) and the list of installed apps. Its top 3 distinguishing feature value distributions are shown in the last

13. The average F-score for the binary classification is even higher than multi-class classification, and is reported in supplementary materials.

14. See supplementary materials for top 5 features and explanations of three other families.

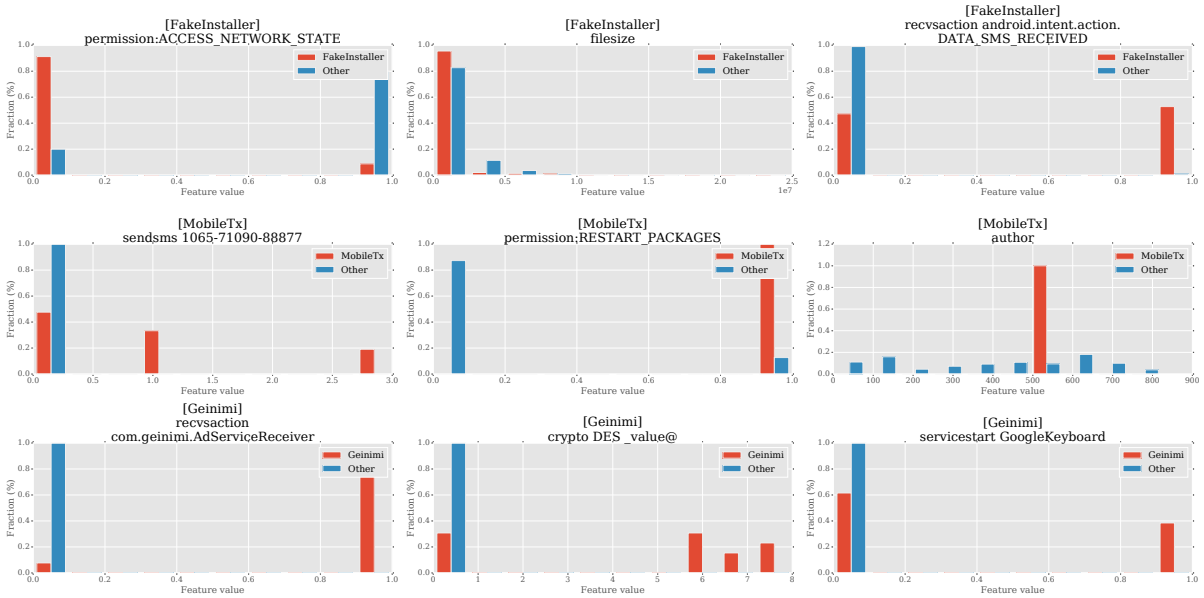


Fig. 9: Feature value comparison between malware families present in the DREBIN dataset. In particular, we report the histogram of the top-3 feature values for the following malware families (number of samples within parenthesis): FakeInstaller (883), MobileTx (21), Geinimi (13). (More families in the supplementary materials). Each row corresponds to a family, and the feature value is compared against all other malware families in the dataset.

row of Figure 9. The first feature is associated with the registration of `AdServiceReceiver`, a component used to monitor system events on which to trigger malicious behavior (e.g., to send stolen data to remove server). The `crypto` operations identified in the top 3 features use the DES algorithm to *decrypt* URLs of the CnC servers and GET commands which are encrypted to avoid static code analysis, but the malware authors recycled decryption keys and hence we detected them during dynamic analysis. The permission `MOUNT_UNMOUNT_FILESYSTEMS` is used to access data in SD card slots, as some variants of `Geinimi` use SD cards to store and load repackaged malicious applications download from the CnC servers. The `servicestart GoogleKeyboard` represents the launch of a background process named `GoogleKeyboard` (to avoid user detection by looking at the phone task manager) that performs malicious actions, steals information periodically and sends it to the CnC whose URLs are decrypted at runtime.

Finally, some families are better distinguished using static features (e.g., `FakeInstaller`), whereas others are better distinguished through dynamic features (e.g., `MobileTx`). These statistics can be used by security analysts to define new signatures for malware classification as they automatically reveal some internals of the code obfuscation techniques adopted for a certain malware family.

8 Performance of Clustering Algorithms

In this section, we start by explaining the metrics used to evaluate clustering algorithms followed by a detailed performance evaluation.

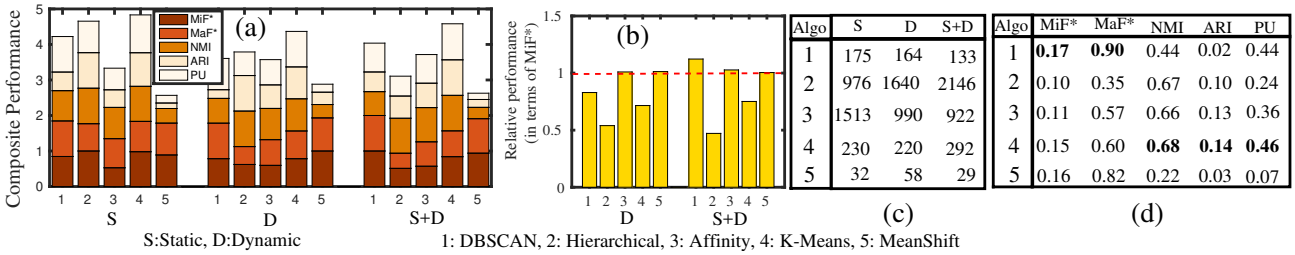
8.1 Evaluation Metrics

We compare the detected clusters with the original families in terms of 5 well-known metrics: Micro F-Score (MiF*), Macro F-Score (MaF*), Normalized Mutual Information (NMI) [27], Adjusted Rand Index (ARI) [39] and Purity (PU) [47]. For MiF*, MaF*, NMI and PU (*resp.* ARI), the value ranges between 0 (*resp.* -1) and 1 where 0 (*resp.* -1) refers to no match with the ground-truth and 1 refers to a perfect match. Note, the definitions of Micro and Macro F-Scores for clustering are slightly different from the ones for the classification. They are defined as follows.

Given N samples and a detected cluster i with size n_i, True Positive (TP_i) denotes the number of pairs (out of ${}^{n_i}C_2$) in which both samples belong to the same ground truth family; if not, they are counted as False Positive (FP_i). Similarly, True Negative (TN_i) counts the number of pairs that are outside cluster i, and that belong to different detected clusters j and k (i, j, k are mutually different), for which both samples belong to different ground-truth families. Once FP_i,TP_i,TN_i are computed for all detected clusters i, the calculation of MiF* and MaF* is the same as MiF and MaF (see Section 6.1).

8.2 Performance Analysis

We consider all malware samples in 156 families. Since clustering methods may produce different outputs depending upon the parameter settings and the initial seed selection, we run each clustering method 50 times, and the average performance is reported. Figure 10 shows the composite performance (as discussed in Section 6.2) of all the clustering methods (see Supplementary Materials for the actual performance values). Among all, K-Means performs the best. Figure 10(b) shows the relative performance of the 5 clustering methods using both static and dynamic features as compared to static features alone. Unlike classification, dynamic features alone do not improve performance. In 3 out of 5 cases, the combination of static and dynamic



Algo	S	D	S+D
1	175	164	133
2	976	1640	2146
3	1513	990	922
4	230	220	292
5	32	58	29

(c)

Algo	MiF*	MaF*	NMI	ARI	PU
1	**0.17**	**0.90**	0.44	0.02	0.44
2	0.10	0.35	0.67	0.10	0.24
3	0.11	0.57	0.66	0.13	0.36
4	0.15	0.60	**0.68**	**0.14**	**0.46**
5	0.16	0.82	0.22	0.03	0.07

(d)

Fig. 10: (a) Composite performance of the clustering methods; (b) relative improvement of the clustering methods w.r.t. their performance with only static features; (c) number of clusters detected for different feature sets; (d) absolute value of the metrics for the clustering methods after considering both static and dynamic features together on the DREBIN dataset.

features leads to an improvement over using static features alone. This suggests that static features are more important than dynamic features in clustering malware. Figure 10(c) shows the number of clusters generated by each method using both static and dynamic features. Figure 10(d) summarizes the accuracy of different methods using both static and dynamic features. Overall, the performance of K-Means averaged over all the evaluation metrics is 0.41, followed by DBSCAN (0.39), Affinity clustering (0.37), Hierarchical (0.29) and MeanShift (0.26). This is in sharp contrast with the results reported in [17] where hierarchical clustering yielded the best results for PC malware. A closer inspection of Figure 10 reveals that DBSCAN dominates others in terms of Micro and Macro F-Score values. The reason might be that DBSCAN correctly identifies True Positives for each cluster/family which information theoretic metrics (such as NMI, ARI) often ignore as pointed in [43]. Moreover, the latter metrics do not penalize clusters with large cardinality and tend to prefer a small number of large clusters (which K-Means produces as shown in Figure 12(a) later). Moreover, K-Means does not capture small families. We now discuss how the clustering methods detect small families.

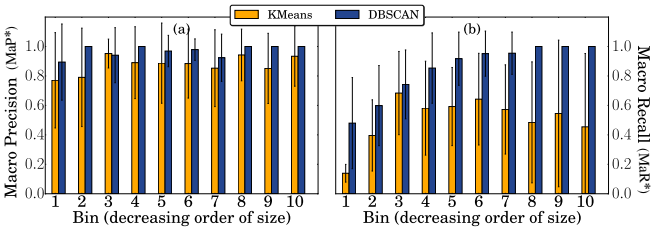


Fig. 11: Family-wise accuracy (mean values of (a) Precision and (b) Recall and their variance) of DBSCAN with static and dynamic features for the DREBIN dataset. The families are sorted in descending order of the size. The results are averaged over 50 iterations.

Family-wise Performance: Figure 11(a) shows that irrespective of the size of the clusters detected, DBSCAN achieves better Precision than K-Means. Moreover, Figure 11(b) shows that the DBSCAN's recall is almost double that of K-Means. These results emphasize the fact that DBSCAN accurately captures the skewed size distribution of Android malware families.

Performance for Small Families: Figure 12(a) shows the cumulative distribution of the cluster size obtained from

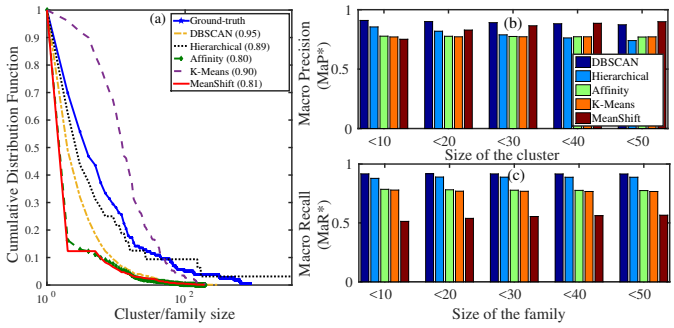


Fig. 12: (a) Cumulative distribution function (CDF) of the size of the clusters detected by different methods on the DREBIN dataset. The Chi-square correlation of the CDF obtained from each method with that of ground-truth is reported in the figure legend; (b) performance of the clustering methods for the small clusters/families having size less than x ($10 \leq x \leq 50$).

different methods as well as that obtained from the ground-truth families. DBSCAN produces the distribution which is closest to ground-truth in terms of Chi-square correlation (0.95). To identify which method best detects small families, we measure the performance for those clusters/families having size less than x (x varies from 10 to 50). Figures 12(b) and 12(c) show that DBSCAN is the best. Interestingly, all clustering methods seem to be quite consistent in the small zones. These results highlight the necessity of adopting clustering methods for identifying small malware families.

Singleton Families: For singleton families, DBSCAN and K-Means seem to be the best and the worst methods respectively. The Precision (Recall) of the clustering methods for detecting only singleton families are: DBSCAN: 0.33 (0.13), Hierarchical: 0.22 (0.07), Affinity: 0.06 (0.11), MeanShift: 0.02 (0.05) and K-Means: 0.002 (0.004). The detection of singleton families is very important and hugely challenging (due to lack of training data) because these identify potentially new types of malware that may need new anti-malware signatures.

8.3 Feature Importance

We identify the most important features in clustering by measuring the percentage decrease in accuracy (in terms of MaF*) when each feature is dropped. Figure 13 reports results for DBSCAN, where the most important feature groups are D:System, S:Author and S:Permissions. Recall

from Figure 10(b) that combining static and dynamic features performs well, but dynamic features alone perform worse than static features. This differs from the classification results shown in Figure 4(b) where dynamic features are extremely important. Thus both types of features are important.

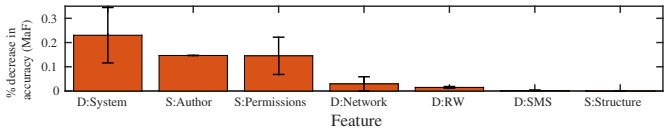


Fig. 13: Average performance (MaF*) decrease per group of static (S) and dynamic (D) features (DBSCAN) on the DREBIN dataset.

To better understand the results in Figure 13, let us look at the top 5 static and dynamic features in Table 3, along with their overall ranking (obtained by sorting for decreasing performance loss). Most of the top 5 dynamic features are related to `dexclass` operations, that correspond to the loading of particular Android classes during execution. Further inspection reveals that the first dynamic feature corresponds to a class loaded by 10 samples (out of 17) of the `FakePlayer` malware family, which is a Trojan that tries to send premium-rate SMSs. Table 3 suggests that very specific dynamic features (such as `dexclass <class-name>`) are used by DBSCAN to discriminate clusters. Therefore, samples with *very* similar features are grouped into *reasonably small* clusters, which are not allowed to grow further, resulting in high Precision and low Recall. The most relevant static features are related to Android permissions, such as those to access GPS location, Internet, send SMS or write to external storage. The `author` is fifth in terms of importance, meaning that it plays a significant role in unsupervised clustering approaches to grouping malware.

TABLE 3: Top 5 features each from static and dynamic set with their overall ranks (OR), sorted by decreasing performance loss of the DBSCAN method for the DREBIN dataset.

OR	Static Feature	OR	Dynamic Feature
2	perm: ACCESS_COARSE_LOCATION	1	dexclass org.me. androidapplication1-1.apk
8	perm:INTERNET	3	dexclass ru.jabox.android. smsbox.lovebox-1.apk
10	perm:SEND_SMS	4	servicestart PlayerBindService
11	perm:READ_PHONE_STATE	5	dexclass ru.jabox.android. smsbox.jokebox-1.apk
12	author	6	dexclass com.agewap.soft-1.apk

9 Performance of EC2

In this section, we present the performance of `EC2`. We start by explaining the baseline methods (Section 9.1), followed by a detailed comparative evaluation: Section 9.2 reports results on DREBIN dataset [4], Section 9.3 reports results on the more recent Koodous academic dataset [9].

9.1 Baseline Methods

We consider the following five methods as baselines: (i) best standalone classifier (Random Forest), (ii) `RHWDL`: an SVM based classification of malware behavior [52], (iii) `HHC`: an ensemble-based hybrid hierarchical clustering [60], (iv) `DroidLegacy`: an existing algorithm specifically designed for Android malware family classification [30], (v) `K2`: an ensemble approach where clustering results are used as features for classification [42]. While methods (ii), (iii) and (iv) were applied on malware datasets, method (v) was used for text classification. K2 turned out to be the best baseline.

TABLE 4: Performance of the baseline methods and `EC2` on the DREBIN dataset. We consider Random Forest (RF) and different clustering methods separately with $\delta_1 = 0.6$ and $\delta_1 = 0.5$. The best baseline and the best combination for `EC2` are highlighted.

	Method	MiF*	MaF*	NMI	ARI	PU
Baselines	RF (Standalone)	0.63	0.84	0.63	0.13	0.41
	`RHWDL` [52]	0.53	0.81	0.60	0.09	0.41
	`HHC` [60]	0.68	0.79	0.59	0.08	0.39
	`DroidLegacy` [30]	0.67	0.82	0.62	0.09	0.35
	`K2` [42]	0.72	0.89	0.63	0.11	0.42
EC2	RF+Hierarchical	0.71	0.94	0.69	0.14	0.47
	RF+Affinity	0.69	0.91	0.69	0.69	0.46
	RF+K-Means	0.67	0.90	0.70	0.14	0.47
	RF+MeanShift	0.73	0.93	0.68	0.14	0.46
	RF+DBSCAN	**0.76**	**0.97**	**0.67**	**0.14**	**0.48**

9.2 Experimental Results on DREBIN

We consider the entire DREBIN dataset, randomly divide it into 5 folds and predict the families for each fold separately. The following experiment is repeated 50 times and results are averaged.

We assume that each test sample $s_i \in TS$ appears one by one and independently of other test samples. We always use the old and reliable training set TR to classify new test samples s_i, instead of augmenting with a newly classified sample[15]. If s_i is classified into one of the existing classes, we keep s_i aside because other representatives of this class are already present in the training set TS. If s_i is classified into a completely new class, we use this information in a systematic way. When the next test sample s_{i+1} appears and is classified into a new class by the classifier (trained on TS), there are two possibilities – (i) either s_{i+1} belongs to the same class s_i belongs to, or (ii) s_{i+1} forms another new class. We then augment s_i and s_{i+1} into TS and run the clustering algorithm on $TS \cup \{s_i, s_{i+1}\}$. If both of them are clustered together, we assign them same class label; otherwise we keep them separately into two different classes. In this way, we avoid including noise due to old misclassification into the current training set.

We report the results for the best settings of the baseline methods – standalone Random Forest is the first baseline; we use the same configurations proposed in [52] and [60] for `RHWDL` and `HHC` respectively. For `DroidLegacy` [30] we extract the Android malware features, run their algorithm using their open-source implementation [7], and report best

15. Augmenting a newly classified sample into the training set might lead to cascading of noise.

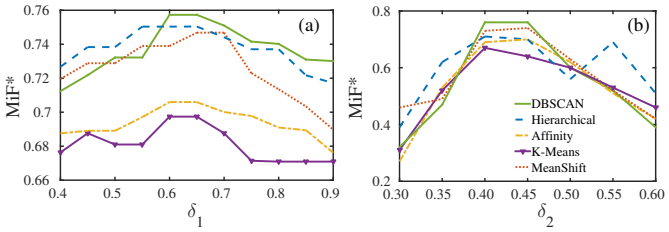


Fig. 14: Change in performance (MiF*) with the increase in (a) δ_1 and (b) δ_2 for EC2 (we consider Random Forest and different clustering methods) on the DREBIN dataset.

results obtained by tuning parameters as recommended in their paper. For K2 we report the results of Random Forest after considering outputs of *all* the clustering methods as features.

Note that the number of detected clusters and the number of ground-truth families may not be identical (c.f. Figure 3 in which there are 2 classes in the training sample; however we obtain 3 clusters in the final result). Therefore, we compare the performance of EC2 with other baselines using cluster evaluation metrics: MiF*, MaF*, NMI, ARI and PU (see Section 8 for formal definition).

Table 4 shows significant performance gains for EC2 (using Random Forest classifier and different clustering methods with $\delta_1 = 0.6$ and $\delta_2 = 0.5$) and other baselines. The values of the thresholds are selected based on Figure 14 – the performance of Random Forest is best for $\delta_1 = 0.6$ and $\delta_2 = 0.45$. The EC2 algorithm with DBSCAN turns out to be the best in this task – MiF*: 0.76, MaF*: 0.97, NMI: 0.67, ARI: 0.14 and PU: 0.48, which is 6%, 9%, 6%, 27% and 14% higher than the best baseline (K2) respectively. Most importantly, EC2 detects singleton families with average Precision (Recall) of 0.43 (0.31), which is significantly higher than the performance reported in Section 8.2. Interestingly, DroidLegacy which was specifically proposed to classify Android malware families turns out to be even worse than K2 and EC2 when considering all the families. This indicates that ensemble approaches which combine clustering and classification results may be the best choice for classifying malware families.

Finally, we evaluated competing methods separately for small and large families. Table 5 shows that for families with size $\geq$ 10, EC2 performs best, even better than Random Forest (standalone). For families with size $<$ 10, EC2 performs slightly worse than DBSCAN probably due to some classification noise propagated from the classifier. However, for small families EC2 still outperforms both RF and K2. Although we have shown the performance of EC2 for Android malware prediction, we believe it can also be used to in general for other datasets.

9.3 Experimental Results on Koodous Dataset

We now present experimental results on the more recent Koodous dataset [9]. Koodous is a community-based platform similar to VirusTotal, but it is entirely focused on static and dynamic analysis of Android malware.

The Koodous academic dataset was generously provided to us by the Koodous administrators and contains

TABLE 5: Comparison of EC2 with the best classifier (Random Forest), the best clustering (DBSCAN) and the best baseline method (K2) for small (size $<$ 10) and large (size $\geq$ 10) families in terms of MiF* and MaF* on the DREBIN dataset.

Method	Small Families ($<$ 10)		Large Families ($\geq$ 10)	
	MiF*	MaF*	MiF*	MaF*
RF (Standalone)	0.29	0.73	**0.78**	**0.93**
DBSCAN (Standalone)	**0.49**	**0.91**	0.68	0.86
K2 [42]	0.36	0.75	0.76	0.90
EC2 (RF+DBSCAN)	**0.45**	**0.89**	**0.82**	**0.95**

50,000 malware samples spanning from December 2015 to May 2016. This dataset is more recent than DREBIN, but only about 6,700 samples have been labeled with coarse-grained families (namely, *categories*). We use the labels already provided in this dataset as the ground truth because they have been closely investigated by Koodous administrators. In particular, the dataset contains the following categories (size within the parenthesis): SMS-fraud (3,257), Adware (3,069), Information-Theft (110), Ransomware (95), Fake-Installer (95), Rooting (50), Banker (31).

We use the Koodous dataset because (i) it contains more recent malware (hence it may be interesting to observe how EC2's performance changes as new obfuscation techniques are encountered), (ii) it contains more coarse-grained families which are more challenging to cluster.

We perform the same feature extraction techniques described in Section 4, and obtain static and dynamic features for the malware samples. We then consider 6,700 labeled malware samples, randomly divide it into 5 folds and predict the families for each fold separately. The following experiment is repeated 50 times and results are averaged.

TABLE 6: Performance of the baseline methods and EC2 on the Koodous dataset. We consider Random Forest (RF) and different clustering methods separately with $\delta_1 = 0.85$ and $\delta_2 = 0.5$. The best baseline and the best combination for EC2 are highlighted.

	Method	MiF*	MaF*	NMI	ARI	PU
Baselines	RF (Standalone)	0.46	0.68	0.59	0.09	0.34
	RHWDL [52]	0.42	0.53	0.50	0.06	0.33
	HHC [60]	0.44	0.65	0.60	0.08	0.37
	DroidLegacy [30]	0.40	0.56	0.53	0.07	0.35
	K2 [42]	0.48	0.70	0.61	0.11	0.39
EC2	RF+Hierarchical	0.52	0.71	0.63	0.11	0.39
	RF+Affinity	0.51	0.69	0.62	0.10	0.38
	RF+K-Means	0.54	0.73	0.64	0.11	0.42
	RF+MeanShift	0.53	0.72	0.63	0.13	0.40
	RF+DBSCAN	**0.56**	**0.74**	**0.65**	**0.13**	**0.45**

Table 6 shows that although the overall prediction performance is lower than that of DREBIN, EC2 still outperforms the comparative baselines. Therefore, we may conclude that EC2 is useful for the datasets containing more recent malware samples (which are intelligently obfuscated and harder to detect).

10 Conclusions

This paper makes several major contributions. First, we proposed *a systematic analysis* of state-of-the-art classification and clustering algorithms applied to the task of grouping

Android malware into families, and by considering different combinations of static and dynamic features. Second, we proposed EC2, the first algorithm used to group malware into families that uses *an ensemble of both classification and clustering approaches*. Third, we show that the performance of EC2 is high even when considering families of all sizes, achieving an overall Micro and Macro F-Score of 0.76 and 0.97, respectively. Moreover, EC2 also *works well on small malware families* which can be very hard to identify, yielding to an F-Score of 0.36 for singleton families which is very good, given the lack of training data. Fourth, a detailed analysis of some Android malware families reveals that our approach is also effective in characterizing and explaining behavior of Android malware families. Fifth, we showed that EC2 is equally successful in detecting families of more recent malware samples, and also performs significantly well with obfuscated feature set. Future work might involve how to group malware families into more coarse-grained categories and improve data-driven understanding of malware behaviors. Moreover, since malware authors may employ increasingly sophisticated obfuscation techniques, future work may also address how to design automated feature combination strategies to make it harder for the attacker to evade classification. For instance, instead of using a base set of features, we could replace them for analysis purposes with various linear or non-linear combinations of features.

ACKNOWLEDGMENTS

The authors would like to acknowledge the suggestions of the anonymized reviewers. Part of this work was funded by ONR Grants N000141612739, N000141512007, N000141612896, and N000141512742, ARO grant W911NF1410358 and Maryland Procurement Office grant H9823014C0137.

REFERENCES

[1] https://techcrunch.com/2015/09/29/android-now-has-1-4bn-30-day-active-devices-globally/, [Oct. 2016].
[2] http://www.gartner.com/newsroom/id/3323017, [Oct. 2016].
[3] http://tinyurl.com/zqm6ufu, [Apr. 2017].
[4] https://www.sec.cs.tu-bs.de/~danarp/drebin/, Visited in Jul.
[5] https://www.virustotal.com/, [Oct. 2016].
[6] Android Genome Project (MalGenome). [Internet]. http://www.malgenomeproject.org/, Visited in Jul. 2017.
[7] DroidLegacy open-source implementation [Internet]. https://bitbucket.org/srl/droidlegacy/src, Visited in Jul. 2017.
[8] GDATA Mobile Malware Report. https://www.gdatasoftware.com/news/2017/02/threat-situation-for-mobile-devices-worsens, Visited in Jul. 2017.
[9] Koodous. Collaborative platform for Android malware analysts [Internet]. https://koodous.com/, Visited in Jul. 2017.
[10] Petya ransomware [Internet]. https://www.nytimes.com/reuters/2017/07/05/business/05reuters-cyber-attack-ukraine-backdoor.html, Visited in Jul. 2017.
[11] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in android. In *SecureComm*, pages 86–103, Sydney,AU, 2013. Springer.
[12] N. Andronio, S. Zanero, and F. Maggi. Heldroid: dissecting and detecting mobile ransomware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 382–404. Springer, 2015.
[13] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto. Clustering Android malware families by HTTP traffic. In *10th Int. Conf. on Malicious and Unwanted Softw. (MALWARE)*. IEEE, 2015.
[14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, San Diego,CA,USA, 2014.
[15] U. Baldangombo, N. Jambaljav, and S.-J. Horng. A static malware detection system using data mining methods. *CoRR*, abs/1308.2831, 2013.
[16] P. Battista, F. Mercaldo, V. Nardone, A. Santone, and C. Visaggio. Identification of android malware families with model checking. In *International Conference on Information Systems Security and Privacy. SCITEPRESS*, 2016.
[17] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, San Diego,CA,USA, 2009.
[18] B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Roli. Poisoning behavioral malware clustering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 27–36. ACM, 2014.
[19] J.-M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.
[20] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.
[21] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *SPSM*, pages 15–26, Chicago,IL,USA, 2011. ACM.
[22] Z. Cai and R. H. Yap. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In *ACM CODASPY*, 2016.
[23] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.
[24] R. Caruana and A. Niculescu-Mizil. An Empirical Comparison of Supervised Learning Algorithms. In *ICML*, pages 161–168, Pittsburgh, PA, USA, 2006. ACM.
[25] T. Chakraborty, S. Kumar, N. Ganguly, A. Mukherjee, and S. Bhowmick. Genperm: A unified method for detecting non-overlapping and overlapping communities. *IEEE Trans. Knowl. Data Eng.*, 28(8):2101–2114, 2016.
[26] M. Chandramohan, H. B. K. Tan, and L. K. Shar. Scalable malware clustering through coarse-grained behavior modeling. In *SIGSOFT/FSE*, pages 161–168, Cary, NC, USA, 2012. ACM.
[27] L. Danon, A. Diaz-Guilera, J. Duch, and A. Arenas. Comparing community structure identification. *JSTAT*, 2005(09):P09008, 2005.
[28] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. DroidScribe: Classifying android malware based on runtime behavior. In *IEEE Security and Privacy Workshops (SPW)*, May 2016.
[29] A. Deo, S. K. Dash, G. Suarez-Tangil, V. Vovk, and L. Cavallaro. Prescience: Probabilistic Guidance on the Retraining Conundrum for Malware Detection. In *AISec Workshop*. ACM, 2016.
[30] L. Deshotels, V. Notani, and A. Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, page 3. ACM, 2014.
[31] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM, 2016.
[32] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 32(2):1–29, 2014.
[33] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.
[34] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015.
[35] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.
[36] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. In *ACM CODASPY*, 2016.

[37] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing Droids: Program slicing for Smali code. In *Proc. ACM Symposium on Applied Computing*, 2013.

[38] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin. MutantX-S: Scalable malware clustering based on static features. In *USENIX ATC*, pages 187–198, San Jose, CA, USA, 2013.

[39] L. Hubert and P. Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[40] C. Kang, N. Park, B. A. Prakash, E. Serra, and V. S. Subrahmanian. Ensemble models for data-driven prediction of malware infections. In *WSDM*, San Francisco, CA, USA, 2016. ACM.

[41] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 6:2721–2744, 2006.

[42] A. Kyriakopoulou and T. Kalamboukis. Using clustering to enhance text classification. In *SIGIR*, pages 805–806, Amsterdam,NL, 2007. ACM.

[43] V. Labatut. Generalised measures for the evaluation of community detection methods. *IJSNM*, 2(1):44–63, 2015.

[44] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *IEEE COMPSAC*, 2015.

[45] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis. AndRadar: Fast discovery of android applications in alternative markets. In *DIMVA*, pages 51–71, Egham, UK, 2014. Springer.

[46] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security*, 51:16–31, 2015.

[47] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[48] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *CoRR*, abs/1612.04433, 2016.

[49] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[50] N. Papernot, P. McDaniel, A. Swami, and R. Harang. Crafting adversarial input sequences for recurrent neural networks. In *Military Communications Conference, MILCOM*, 2016.

[51] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android anti-malware against transformation attacks. In *Proc. 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[52] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA*, pages 108–125, Paris, France, 2008. Springer.

[53] A. Sadeghi, H. Bagheri, J. Garcia, et al. A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software. *IEEE Trans. Software Engineering*, 2016.

[54] M. Siddiqui, M. C. Wang, and J. Lee. Data mining methods for malware detection using instruction sequences. In *Artificial Intelligence and Applications*, pages 358–363, 2008.

[55] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. DroidSieve: Fast and accurate classification of obfuscated android malware, March 2017.

[56] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[57] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya. The company you keep: Mobile malware infection rates and inexpensive risk indicators. In *WWW*, pages 39–50, Seoul,KR, 2014. ACM.

[58] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, pages 569–584, Bellevue, WA,USA, 2012.

[59] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European Symposium on Research in Computer Security*. Springer, 2014.

[60] Y. Ye, T. Li, Y. Chen, and Q. Jiang. Automatic malware categorization using cluster ensemble. In *SIGKDD*, pages 95–104, Washington, DC, USA, 2010. ACM.

[61] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu. Combining file content and file relations for cloud based malware detection. In *SIGKDD*, pages 222–230, San Diego, CA, USA, 2011. ACM.

[62] S. Yu, G. Gu, A. Barnawi, S. Guo, and I. Stojmenovic. Malware propagation in large-scale networks. *IEEE TKDE*, 27(1):170–179, 2015.

[63] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *ACM CCS*, 2014.

[64] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.

Tanmoy Chakraborty is an Assistant Professor and a Ramanujan Fellow in the Dept of Computer Science & Engineering, Indraprastha Institute of Information Technology Delhi (IIIT-D). Prior to this, he was a postdoctoral researcher at University of Maryland, College Park, USA. He finished his Ph.D. as a Google India Ph.D fellow from IIT Kharagpur, India in 2015. His Ph.D thesis was recognized as best thesis by IBM Research India, Xerox research India and Indian National Academy of Engineering (INAE). His broad research interests include Data Mining, Social Media and Data-driven Cybersecurity. Home page: http://faculty.iiitd.ac.in/~tanmoy/

Fabio Pierazzi is a postdoctoral researcher at the University of Modena and Reggio Emilia, Modena, Italy. In the same institute, he completed the Master's Degree in Computer Engineering 2013 and the PhD in Computer Science in 2017. During 2016, he spent 10 months as a visiting research scholar at the Department of Computer Science, University of Maryland, College Park, MD, USA. His research interests focus on security analytics, network security, malware classification, intrusion detection, and all solutions that combine cybersecurity and data mining. Home page: http://weblab.ing.unimo.it/people/fpierazzi

V.S. Subrahmanian is the Dartmouth College Distinguished Professor in Cybersecurity, Technology, and Society. From 1989 to 2017, he was a Professor with the Department of Computer Science, University of Maryland, College, Park, having previously served as Director of the University of Maryland Institute for Advanced Computer Studies (UMIACS). He was with the Editorial Boards of numerous ACM and IEEE journals as well as Science, the Board of Directors of the Development Gateway Foundation (set up by the World Bank), SentiMetrix, Inc., and with the Research Advisory Board of Tata Consultancy Services. He has worked extensively at the intersection of databases and artificial intelligence. He has authored over 200 refereed publications, and is a Fellow of both AAAI and AAAS. Home page: https://www.cs.umd.edu/users/vs/