R-PackDroid: Practical On-Device Detection of Android Ransomware

Michele Scalas, Davide Maiorca, *Member, IEEE,* Francesco Mercaldo, Corrado Aaron Visaggio, Fabio Martinelli, *Senior Member, IEEE,* and Giorgio Giacinto, *Senior Member, IEEE,*

Abstract—Ransomware constitutes a major threat for the Android operating system. It can either lock or encrypt the target devices, and victims may be forced to pay ransoms to restore their data. Despite previous works on malware detection, little has been done to specifically identify Android malware as ransomware. This is crucial, as ransomware requires immediate countermeasures to avoid data being entirely compromised. In this paper, we propose R-PackDroid, a machine learning-based application (which directly runs on Android phones) for the detection of Android ransomware. R-PackDroid is a lightweight approach that leverages a methodology based on extracting information from system API packages. We demonstrate its effectiveness by testing it on a wide number of legitimate, malicious and ransomware-based applications. Our analyses pointed out three major results: first, R-PackDroid can distinguish ransomware from malware and legitimate applications with very high accuracy; second, R-PackDroid guarantees resilience against heavy obfuscation attempts, such as class encryption; third, R-PackDroid can be used to effectively predict and detect novel ransomware samples that are released after the ones used to train the system. R-Packdroid is available on the Google Play Store, and it is the first, academic ransomware-oriented detector available for Android.

Index Terms—Malware; Android; Ransomware; Machine Learning.

1 Introduction

The term `ransomware` usually refers to attacks that lock the victim's device or encrypt its data, by asking a sum of money to restore the compromised functionality. Such attacks are particularly devastating, as they could entirely destroy sensitive data of private users and companies. According to Symantec, the number of ransomware variants increased in 2017 by 46%, with massive outbreaks such as the one concerning Ukrainian companies (`Petya/NotPetya`). Hence, it is not surprising to see that the same trend applied to mobile ransomware, with more than $42,000$ samples blocked in 2017 [18].

Mobile ransomware typically features different characteristics in comparison to its X86 counterpart. As performing data encryption typically requires high-level privileges (especially to write on areas that are directly managed by the kernel), most attacks only lock the target device by making victims believe that their data are encrypted, or by warning them that they are currently controlled by the police for their actions (a strategy directly inspired by *scareware*-based approaches).

To counteract mobile malware, Machine Learning has been increasingly used both by researchers and anti-malware companies, either to perform direct detection or to generate signatures. Such ML-based algorithms normally discriminate between legitimate and malicious applications by analyzing static information that belongs to different categories, such as permissions, API calls, IP Addresses, and so forth. `Drebin` and `StormDroid` are popular implementations of such algorithms, but more have been developed during these years [1], [5], [9], [11]. While such approaches have proved to be extremely effective at detecting malicious files in the wild, they feature two major limitations: *(i)* They can only discriminate malicious and benign files, without distinguishing the various categories of attacks (for example ransomware, spyware, etc.). This raises a critical issue to establish the user reaction time to the threat. Differently to spyware and other generic malware, ransomware attacks entirely compromise the device data, thus requiring immediate mitigations. *(ii)* They extract huge amounts of information from multiple parts of the file (AndroidManifest, executable DexCode) to perform detection. In particular, `Drebin` itself uses hundreds of thousands of features. While this approach increases the overall accuracy of the system, it extends the degrees of freedom of a skilled attacker to perform targeted attacks against the learning algorithm. For example, it would be quite easy to mask a certain IP address, if the attacker understood that this has an important role for detection [11].

To overcome such limitations, researchers introduced *ransomware-oriented* detection approaches, which essentially attempted to specifically recognize ransomware attacks by exploiting the differences between their *modus operandi* and the one employed by generic malware. Normally, ransomware attacks are characterized by screens that are constantly visualized to the user, containing messages (in various languages) that ask for ransoms. For this reason, previous works based their detection on the presence of ransom-related strings, on the abuse of specific admin API,

- *M. Scalas (michele.scalas@diee.unica.it) D. Maiorca (davide.maiorca@diee.unica.it), and G. Giacinto (giacinto@diee.unica.it) are with the Department of Electrical and Electronic Engineering, University of Cagliari, Piazza d'Armi, 09123 Cagliari, Italy.*
- *C. A. Visaggio (visaggio@unisannio.it) is with the Department of Engineering, University of Sannio, Benevento, Italy.*
- *F. Mercaldo (francesco.mercaldo@iit.cnr.it) and F. Martinelli (fabio.martinelli@iit.cnr.it) are with the Institute of Informatics and Telematics, National Research of Council, Pisa, Italy.*

etc. [3], [21] Albeit effective, these approaches are limited by the fact that certain features are extremely prone to obfuscation (for example, all string-related ones), and that a linguistic dictionary is actually required to detect attacks in languages that are different to English. Moreover, *none of the ransomware-oriented approaches have been publicly ported to Android and can be directly used on mobile phones*.

In this paper (which greatly extends our previous work in [14]), we address the aforementioned limitations by proposing R-PackDroid, a ransomware-oriented, Machine Learning-based detector that *discriminates ransomware, goodware and generic malware directly on Android devices*. More specifically, our detector features the following characteristics:

- **Portability**. R-PackDroid is available on the Google Play Store[1], runs on any Android device, and has been optimized to perform *on-device detection*, without requiring external connections.
- **Compactness**. The rationale behind R-PackDroid is showing that it is possible to develop an accurate, multi-functional detector by using a *very compact set* of features, i.e., *system API packages*. The key aspect of this methodology is that, albeit such information may seem quite generic at a first glance, it is actually employed very differently by ransomware, generic malware and goodware.
- **Accuracy and Flexibility**. R-PackDroid has been mostly designed to discriminate between ransomware and goodware with very high accuracy, but it can also be applied to detect generic malware attacks with a good detection rate, thus distinguishing them from their ransomware counterparts.
- **Resilience**. The choice of API-packages to perform detection is also related to the resilience of such information against obfuscation. In this paper, we in fact demonstrate that even complex, static-obfuscation techniques such as class encryption do not defeat our detection approach: this is because the information about package API is much more difficult to remove or replace than a single API call.

We discuss and validate our methodology by proposing the results of various insights and experiments. First, we show that R-PackDroid is able to detect ransomware with very high accuracy and low false positives, while keeping a good detection rate for generic malware as well. Second, we discuss the performances attained by R-PackDroid against a consistent number of obfuscated samples, showing that our approach is robust even against heavy static-obfuscation techniques such as class encryption. Third, we show how R-PackDroid can predict possible novel attacks while being trained on previously released data. Finally, we propose a comparison between R-PackDroid and other state-of-the-art approaches, showing that it can be used as a good alternative to detect attacks that may evade other systems. The attained results show that R-PackDroid can be used as a valid aid, also in combination with other anti-malware solutions (it can be installed together with other security apps), to protect mobile devices against ransomware and malware. We also claim that effective detection can be carried out by using a limited, yet discriminant number of features, and we encourage research that is aimed to design discriminant features that are truly useful for detection.

1. https://play.google.com/store/apps/details?id=it.michelescalas.rpackdroid

Paper structure. Section 2 provides the basic concepts of Android apps; Section 3 discusses the basic characteristics of Android ransomware, and describes the key-intuition R-PackDroid is based on; Section 4 provides a description of the related work in the field; Section 5 describes the employed methodology and implementation to build our system; Section 6 illustrates the experimental results and some insights about the packages used by the analyzed data; Section 7 discusses the limitations of our work, which is finally concluded by Section 8.

2 Background on Android

Android applications are zipped .apk (i.e., Android application package) archives that contain the following elements: *(i)* The AndroidManifest.xml file, which provides the application package name, and lists its basic components, along with the permissions that are required for specific operations; *(ii)* One or more classes.dex files, which are the true executable of the application, and which contain all the implemented classes and methods (in the form of Dalvik bytecode) that are executed by the app. This file can be disassembled to a simplified format called smali; *(iii)* Various .xml files that characterize the application layout; *(iv)* External resources that include, among others, images and native libraries.

Although Android applications are typically written in Java, they are compiled to an intermediate byte-code format called Dalvik (which be further referred to as DexCode), whose instructions are contained in the classes.dex file. Such format is essentially obtained by converting compiled Java .class files to a more compact, register-based byte-code representation. The classes.dex files belonging to the apps are then further parsed at install time, and converted to native ARM code that is executed by the Android RunTime (ART). This technique allows to greatly speed up execution in comparison to the previous runtime (dalvikvm, available till Android 4.4), in which applications were executed with a just-in-time approach (during installation, the classes.dex file was only slightly optimized, but not converted to native code).

3 Android Ransomware

To better understand the core intuition R-PackDroid is based on, it is important to provide a general description of the actions performed by ransomware. The majority of ransomware-based attacks for Android are based on the goal of *locking* the device screen while asking the victim for money in order to unlock it. According to the taxonomy proposed by [8], there are multiple ways to do so: *(i)* by resorting to a *hijacking* activity (i.e., a screen that the user visualizes and with which she can interact) that is constantly shown; *(ii)* by setting up specific parameters of certain API calls; *(iii)* by disabling certain buttons, such home or back.

Locking is generally preferred to other data encryption strategies because it does not require to operate on high-privileged data. Indeed, accessing specific areas of the Android internal memory would only be possible with root permissions. Conversely, locking the device does not require particularly high privileges, and would allow the attacker to ensure his goal (i.e., scaring the victim) with minimum effort. The majority of locking screens show the victim writings and images related to police activities, or alternatively pornographic material. There are, however, samples that also perform data encryption. According to [8], only four ransomware families possess the ability of encrypting data: `Simplocker`, `Koler`, `Cokri` and `Fobus`. In particular, some of these families employ a customized encryption algorithm, while others resort to standard algorithms.

The whole `R-PackDroid` project revolves around *how locking and encryption* actions are performed. As such actions require the usage of multiple functions that involve core functionalities of the system (e.g., managing entire arrays of bytes, displaying activities, manipulating buttons and so on), *attackers tend to use functions that directly belong to the Android system API*. This is because it would be extremely time consuming (and probably also inefficient) to build new APIs that perform the same actions as the original ones.

As an example of this behavior, consider the `DexCode` snippet provided by Listing 1, belonging to a *locker-type* ransomware[2]. In this example, it is possible to observe that the two function calls (expressed by `invoke-virtual` instructions) that are actually used to lock the screen (`lockNow`) and reset the password (`resetPassword`) are system-API calls, belonging to the class `DevicePolicyManager` and to the package `android/app/admin`. The same behavior is provided by Listing 2, which shows the encryption function employed by a *crypto-type* ransomware sample[3]. Again, the functions to manipulate the bytes to encrypt belong to the system-API (`read` and `close`, belonging to the `FileInputStream` class of the `java/io` package; `flush` and `close`, belonging to the `CipherOutputStream` class of the `javax/crypto` package).

As system API calls can be quite numerous, the key idea is therefore finding a compact, yet effective representation to provide information about this behavior. For this reason, we decided to leverage `API-packages` that, despite being less informative than API-methods, can represent in a compact way the actions perform by ransomware, as well as the *differences in terms of API between ransomware, generic malware, and goodware*.

```
1
2 invoke-virtual {v9},
      Landroid/app/admin/DevicePolicyManager;->lockNow()V
3 move-object v9, v0
4 move-object v10, v1
5
6 ...
7
8 move-result-object v9
9 move-object v10, v7
10 const/4 v11, 0x0
11 invoke-virtual {v9, v10, v11},
      Landroid/app/admin/DevicePolicyManager;->resetPassword
      (Ljava/lang/String;I)Z
```

Listing 1: Part of the onPasswordChanged() method belonging to a locker-type ransomware sample.

```
1
2 Ljava/io/FileInputStream;->read([B)I
3 move-result v0
4 const/4 v5, -0x1
5 if-ne v0, v5, :cond_0
6 invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->
      flush()V
7 invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->
      close()V
8 invoke-virtual {v3}, Ljava/io/FileInputStream;->close()V
```

Listing 2: Parts of the encrypt() method belonging to an encryption-type ransomware sample.

4 Related work

Most of Android malware detectors typically discriminate between malicious and benign apps, and we refer to them as *generic malware-oriented* detectors. However, as the main goal of `R-PackDroid` is detecting ransomware, this Section will be mainly focused on describing systems that aim to specifically detect such attacks (*ransomware-oriented* detectors). A brief description of the other detectors will be provided at the end of this Section.

2. MD5: `0cdb7171bcd94ab5ef8b4d461afc446c`
3. MD5: `59909615d2977e0be29b3ab8707c903a`

The most popular and publicly available *ransomware-oriented* detector is `HelDroid`, proposed by Andronio et al. [3]. This tool includes a text classifier based on NLP features, a lightweight `smali` emulation technique to detect locking strategies, and the application of taint tracking for detecting file-encrypting flows.

The system has then been further expanded by Zheng et al. with the new name of `GreatEatlon`, and features significant speed improvements, a multiple classifier system that combines the information extracted by text- and taint-analysis, and so forth [21]. However, the final label provided by the system for detection is only *malicious* or *benign*, with no final decision on the sample being ransomware or not. Furthermore, the system is still computationally demanding and it still strongly depends on a text classifier: the authors trained it on generic threatening phrases, similar to those that typically appear in ransomware or scareware. This strategy can be easily thwarted by means of, e.g., string encryption [13]. Moreover, it strongly depends on the presence of a language dictionary for that specific ransomware campaign.

Yang et al. proposed a tool to monitor the activity of ransomware by dumping the system messages log, including stack traces. Sadly, no implementation has been released for public usage [20].

Song et al. proposed a method that aims to discriminate between ransomware and goodware by means of process monitoring [17]. In particular, they considered system-related features representing the I/O rate, as well as the CPU and memory usage. The system has only been evaluated with only one ransomware sample developed by the authors, and no implementation is publicly available.

Cimitille et al. introduced an approach to detect ransomware that is based on formal methods (by using a tool called `Talos`), which help the analyst identify malicious

TABLE 1: An overview of the current state-of-the-art, ransomware-oriented approaches.

Work	Year	Static	Dynamic	Machine-Learning	Available
Chen et al. (RansomProber) [8]	2018		✓		
Cimitille et al. (Talos) [10]	2017	✓			
Gharib et al. (Dna-Droid) [12]	2017	✓	✓	✓	
Song et al. [17]	2016		✓		
Zheng et al. (GreatEatlon) [21]	2016	✓		✓	✓
Yang et al. [20]	2015		✓		
Andronio et al. (HelDroid) [3]	2015	✓		✓	✓

sections in the app code [10], [15]. In particular, starting from the definition of payload behavior, the authors manually formulated logic rules that were later applied to detect ransomware. Unfortunately, such procedure can become extremely time-consuming, as such rules should be manually expressed by an expert.

Gharib et al. proposed `Dna-Droid`, a static and dynamic approach in which applications are first statically analyzed, and then dynamically inspected if the first part of the analysis returned a suspicious result. The system uses Deep Learning to provide a classification label [12]. The static part is based on textual and image classification, as well as on API calls and application permissions. The dynamic part relies on sequences of API calls that are compared to malicious sequences, which are related to malware families. Obviously, this approach has the drawback that heavily obfuscated apps can escape the static filter, thus avoiding to be dynamically analyzed.

Finally, Chen et al. proposed `RansomProber`, a purely dynamic ransomware detector which employs a set of rules to monitor different aspects of the app execution, such as the presence of encryption or anomalous layout structures. The attained results report a very high accuracy, but sadly the system has not been publicly released yet (to the best of our knowledge).

Table 1 shows a comparison between the state-of-the-art methods for specifically detecting or analyzing Android ransomware. It is possible to observe that there is a certain balance between static- and dynamic-based methods. Some of them also resort to Machine-Learning to perform classification. Notably, only `HelDroid` and `GreatEatlon` are currently publicly available.

With respect to *generic malware-oriented* detectors, Arp et al. proposed `Drebin`, a machine learning system that uses static analysis to discriminate between generic malware and trusted files. They extracted various features from both the Manifest file and the Android executable, including IP addresses, suspicious API calls, permissions, etc. [5]. Tam et al. introduced a system to perform dynamic analysis and detection of Android malware by analyzing the system calls performed by the application [19].

Aresu et al. clustered Android malware by using the network HTTP traffic generated by those applications [4]. Such clusters can be used to generate signatures that allow to discriminate between malware and legitimate applications. Canfora et al. experimentally evaluated two techniques for detecting Android malware: the first one is based on Hidden Markov Model (HMM), and the second one exploits Structural Entropy [7]. The attained results showed that both techniques could be suitable to Android malware detection.

Chen et al. proposed `StormDroid`, a static and dynamic machine-learning based system that extracts information from API-calls, permissions and behavioral features [9]. Finally, Ahmadi et al. proposed `IntelliAV`, a *generic malware-oriented* detector that is publicly available on Android. Differently to `R-PackDroid`, such detector provides a level of dangerousness for each app, but does not directly specify the family nor the type of attack [1].

5 Methodology and Implementation

After having clarified the key-idea behind our project, we now provide a detailed description of the methodology employed by `R-PackDroid`. R-PackDroid is a *ransomware-oriented*, Machine-Learning-based system for the detection of Android ransomware. With the term *ransomware-oriented*, we mean that even if the system is able to also detect generic malware, *its main scope is specifically to detect ransomware*.

The general structure of R-PackDroid essentially reflects the typical scheme of a Machine Learning system for Android malware detection. It takes as input an Android application, analyzes it and returns three possible outputs: **ransomware**, **generic malware** or **trusted**. The analysis is performed in three steps:

- **Pre-Processing**. In this phase, the application is analyzed to extract its `DexCode`. `R-PackDroid` only analyzes the executable code and does not perform any kind of analysis on other elements, such as the application Manifest. Only specific lines of code, which will be described later in this Section, will be sent to the next module.
- **Feature Extraction**. In this phase, the code lines received from the previous phase are further analyzed to extract the related *system API packages*. The *occurrence* of such packages is then counted, thus producing a vector of numbers (*feature vector*) that is sent to a classifier.
- **Classifier**. It can be typically depicted as a mathematical function that takes as input a feature vector and returns a label associated to it. Such label can be represented as *ransomware*, *malware* and *trusted*. Its parameters are tuned through a phase called *training*.

The aforementioned structure is graphically represented in Figure 1. In the following, we provide more details about each phase of the analysis, as well as an insight into the implementation of the system as an Android app.

5.1 Preprocessing and Feature Extraction

The general idea of the first two phases is performing static analysis of the Dalvik bytecode contained in the **classes.dex** file. The goal is retrieving the *system API packages* employed

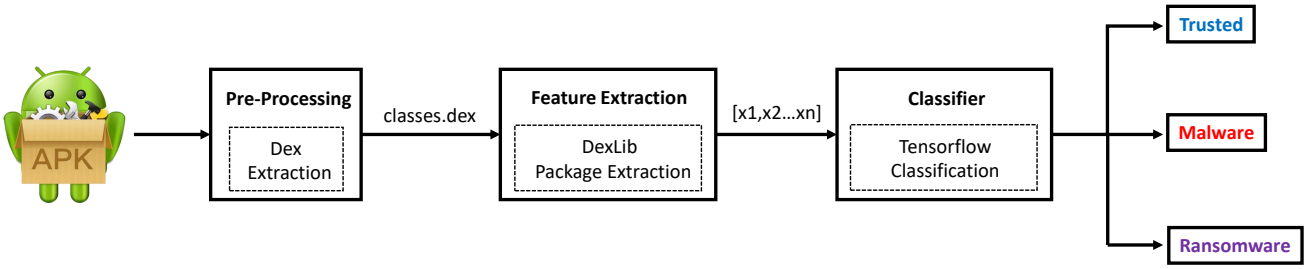

Fig. 1: General Structure, among with some implementation details, of the R-PackDroid Android App.

by method calls inside the application. Such approach revolve around four basic concepts:

- **Coherence with actions**. Most ransomware writers resort to system APIs to carry out memory- or kernel-related actions (for example, file encryption or memory management). This is why system API packages possess a consistent discriminant power: focusing on user-implemented APIs or packages (like it happens, for example, with `Drebin` [5]) exposes the system to a risk of being evaded by simply employing different packages to perform actions.
- **Small feature set**. Using System API packages significantly reduces the number of features that are used by the system. This is particularly helpful, as system designers can easily visualize the behavior of each feature that is employed by the application in the dataset. This is also the reason why we did not employ system API classes or methods: this would have led to thousands of features that would have been quite hard to be properly managed and understood.
- **Independence from Training**. Using system API packages makes the choice of the features independent from the training data that are used. This means that it is less likely that applications are not correctly analyzed only because they employ never-seen before packages.
- **Resilience against obfuscation**. Using heavy obfuscation routines typically lead to inject system API-based code in the executable. This means that, even if some APIs can be concealed by the usage of heavy obfuscation routines, the file can still be detected.

The pre-processing is hence easily performed by directly extracting the **classes.dex** file from the `.apk` app. Since `.apk` files are essentially zipped archives, such operation is rather straight-forward.

Once pre-processing is complete, the `classes.dex` file is further analyzed by the feature extraction module, which inspects the executable file for all `invoke`-type instructions (i.e., all instructions related to invocations) contained in the `classes.dex` code. Then, each invoke is further inspected for the presence of system-API packages (without considering parameters and return types). If a specific API package is found, its occurrence value is increased by one. As an example, consider Listing 3, in which the feature extractor module parses the same code of Listing 2. The four `invoke` instructions are related to the `javax/crypto` and `java/io` packages, which are counted respectively twice.

```
1 Feature List
2
3 Ljavax/crypto/
4 Ljava/io/
5 Ljava/lang/
6
7 Code
8
9 Ljava/io/FileInputStream;->read([B)I
10 move-result v0
11 const/4 v5, -0x1
12 if-ne v0, v5, :cond_0
13 invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->
      flush()V
14 invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->
      close()V
15 invoke-virtual {v3}, Ljava/io/FileInputStream;->close()V
16
17 Feature Vector
18
19 [2 2 0]
```

Listing 3: An example of feature extraction by considering a small number of features.

The `java/lang` package is never used in this snippet, hence its value is zero.[4]

As API reference, we chose Android Nougat (API 25), which is now widely used by most new-generation devices. This leads to 270 API packages (and therefore, 270 features).

5.2 Classification

`R-PackDroid` features a *supervised approach*, in which the system is trained with samples whose label (i.e., benign, generic malware or ransomware) is known. Such technique has been used in previous works with very good results [5], [11]. In particular, our application employs Random Forest classifiers, which are especially good to handle multi-class problems, and which are widely used for malware detection.

It is worth noting that we also considered unsupervised approaches such as clustering, but we did not obtain satisfactory results with respect to the type of features we chose.

5.3 Implementation

`R-PackDroid` has been designed to work on the largest amount of devices possible. Hence, during its development, we focused on optimizing its speed and battery consumption. For this reason, we avoided any textual parsing of bytecode lines (which can be attained by transforming the `.dex` file to a number of `.smali` files with `ApkTool`). We

4. For this example, we considered a very small subset of features.

therefore resorted to DexLib, a powerful parsing library part of the baksmali[5] disassembler (and used by ApkTool itself), to directly extract method calls and their related packages. This allows to obtain a very high precision at analyzing method calls, and significantly reduces the presence of bugs or wrong textual parsing in the analysis phase.

The classification model has been implemented by using Tensorflow[6], an open source, machine learning framework which has been designed to be also used in mobile phones. In particular, we adapted its Random Forest implementation (TensorForest) to the Android operating system. Notably, our Android application only performs classification by using a previously trained classifier. The training phase is carried out separately, on standard X86 architectures. This choice was made to ensure the maximum easiness of use to the final user, thus reducing the risk of invalidating the existing model.

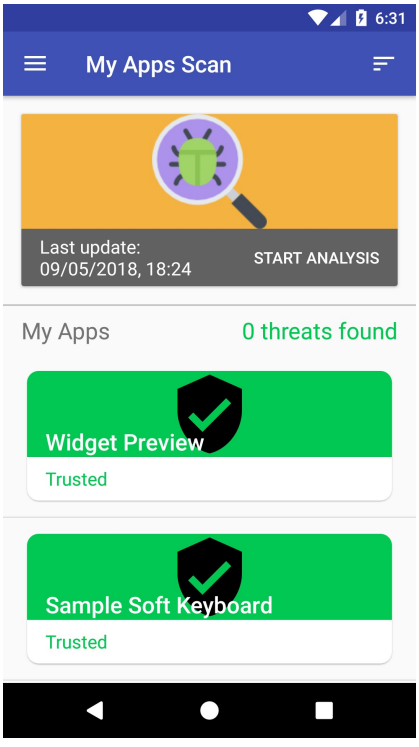

Fig. 2: An example of the Android R-PackDroid screen.

Figure 2 shows an example of the main screen of R-PackDroid. The application is parsed either when it is downloaded from any store, or when the user decides to scan it (or to scan the whole file system). Each application is identified by a box, whose color is associated to the application label (green for trusted, red for malware and violet for ransomware). After getting the result, by clicking on each box related to the scanned application, it is possible to read more details about the packages that it employs, as well as general information such as the app hash and size. Moreover, if the user believes that the result reported by R-PackDroid is wrong, she can report it by simply tapping a button (a privacy policy to accept is obviously also included). To this scope, we resort to the popular service FireBase[7].

6 Experimental Evaluation

In this Section, we report the experimental results attained from the evaluation of R-PackDroid. Note that, for the sake of simplicity and speed (and also to effectively use multiple models trained in a different way), we did not run the experiments on the Android application itself, but on a X86 machine. The trained model that is released with the Android version of R-PackDroid has been trained with all the samples used in this evaluation.

The rest of this Section is organized as follows: we start by providing an overview of the dataset employed in our experiments. Then, we describe the results attained by three evaluations. The first one aimed to establish the general performances of R-PackDroid by considering random distributions of training and test samples. The second one had the goal of checking whether R-PackDroid was resilient against applications that were obfuscated in multiple ways. The third one aimed to show how R-PackDroid behaved when samples released after its training data were actually analyzed. Finally, we provide an overview of the R-PackDroid computational performances and a comparison to other state-of-the-art approaches.

6.1 Dataset

In the following, we describe the dataset employed in our experiments. Without considering obfuscated applications (which are going to be discussed in Section 6.3), we obtained and analyzed 19714 apps, which are organized in the three categories we mentioned in Section 5.

6.1.1 Ransomware

The 3017 samples used for our ransomware dataset were retrieved from the VirusTotal service[8] (which aggregates the detection of multiple anti-malware solutions) and from the HelDroid dataset[9] [3]. With respect to the samples obtained from VirusTotal, we used the following procedure to obtain the samples: *(i)* we looked and downloaded the Android samples whose anti-malware label included the word *ransom*; *(ii)* for each downloaded sample, we extracted its family by using the AVClass tool, which essentially combines the various labels provided by anti-malware solutions to create a unique label that identifies the sample itself [16]; *(iii)* we considered only those samples whose family was coherent to ransomware behaviors, or was known to belong to ransomware.

In general, our goal was obtaining a representative corpus of ransomware to ascertain the prediction capabilities of R-Packdroid. For this reason, the dataset includes families that perform both device locking (such as Svpeng and LockScreen) and encryption (such as Koler and SLocker). For a better description of the aforementioned families, please see Section 3.

5. https://github.com/JesusFreke/smali
6. https://www.tensorflow.org/
7. https://firebase.google.com/
8. http://www.virustotal.com
9. https://github.com/necst/heldroid

TABLE 2: Ransomware families included in the employed dataset.

Family	Samples
Locker	752
Koler	601
Svpeng	364
SLocker	281
Simplocker	201
LockScreen	122
Fusob	120
Lockerpin	120
Congur	90
Jisut	86
Other	280

6.1.2 Malware and Trusted

We considered a dataset composed of 8304 Android malware samples taken from the following sources: *(i)* `Drebin` dataset, one of the most recent, publicly available datasets of malicious Android applications[10] (which also contains the samples from the Genome dataset [22]); *(ii)* `Contagio`, a popular free source of malware for X86 and mobile; *(iii)* `VirusTotal`.

In order to download trusted applications, we resorted to two data sources: *(i)* we crawled the Google Play market using an open-source crawler[11] *(ii)* we extracted a number of applications from the `AndroZoo` dataset [2], which features a snapshot of the `Google Play` store, allowing to easily access applications without crawling the Google services. We obtained 8393 applications that belong to all the different categories available on the market. We chose to focus on the most popular apps to increase the probability of downloading malware-free apps.

6.2 Experiment 1: General Performances

In this experiment, we evaluated the general performances of `R-PackDroid` at detecting ransomware and generic malware. To do so, we randomly split our dataset by 50%, thus using the first half to train the system and the second half to evaluate the system. The number of trees of the random forests was evaluated by performing a 10-fold cross validation on the training data. We repeated the whole process 5 times and we averaged the results by also determining the standard deviation, in order to understand the dependence of the system on the training data.

Considering the multi-class nature of the problem, we represented the results by calculating the ROC curve for `R-PackDroid` in two different cases:

- **Ransomware against benign samples**. This was done as the main scope of `R-PackDroid` is detecting ransomware and, more importantly, *to avoid them being considered as benign files*. This is a crucial point, as it would be a critical mistake to wrongly consider a ransomware sample as benign, as this would most likely compromise the whole device by locking it or encrypting its data.
- **Generic malware against benign samples**. As `R-PackDroid` has also been built with the capability of detecting generic malware, it was of interest to see how it performed at this task. Notably, as the features of `R-PackDroid` were selected to *detect and report* ransomware attacks, it was reasonable to expect that our system could not outperform generic malware-oriented detectors such as `Drebin` or `StormDroid` [5], [9]).

Results are reported in part (a) of Figure 3. Considering the attained performances (reported on the left), we can deduce that the system exhibited the following behaviors:

1) `R-PackDroid` was able to precisely detect more than 97% of ransomware samples with only 1% of false positives. Considering the fact that our dataset included a consistent variety of families, we claim that `R-PackDroid` was able to detect the majority of ransomware families in the wild.
2) `R-PackDroid` featured good accuracy with relatively low false positives (around 85% at 2%, more than 90% at 5%) at detecting generic malware. Considering the higher variety and complexity of generic malware in the wild, we believe that this an encouraging result, given the fact that only system-based packages were used for detection.

The attained results show that `R-PackDroid` performed very well at ransomware detection. A legitimate question is *why* such approach is so effective at this task. As we employed Random Forests, a non linear classification model composed of an ensemble of classifier, it is harder to understand which features are truly discriminant for the classifier. However, it is possible to employ metrics that are typically used for decision trees to have at least an idea of which features could be considered as discriminant.

Part (b) of Figure 3 shows the top-25 most relevant features for the Random Forest classifier, according to its information gain IG. Such gain is typically represented as the following:

$$IG(T, a) = H(T) - H(T|a) \tag{1}$$

where $H(T)$ is the overall entropy for the whole dataset T and $H(T|a)$ is the average entropy obtained by splitting the set T using the attribute a. The higher is the gain, the more relevant the feature is.

The attained results depicted a very interesting scenario. Note how the information gain for each feature is not so high, meaning both that the system does not particularly overfit on specific information, and that the final decision is taken by considering a combination of multiple features. There are, obviously, some attributes that are more important than others. Some of these are related to the core `Java` functionalities (e.g., `java.util`, `java.lang.reflect`). Others are related to how Android display activities (e.g., `android.content.res` and `android.graphics`), or to actions that require privileges and that are typically employed by ransomware (e.g. `android.app.admin`).

It is also worth noting that, even though information gain provides a possible metric to understand which features can be important for the classifier, the final decision of the Random Forest ensemble is taken through a majority voting process. This means that there may be differences between the results obtained with information gain and the real role that the feature itself holds in the ensemble decision.

10. https://www.sec.cs.tu-bs.de/~danarp/drebin/
11. https://github.com/liato/android-market-API-py

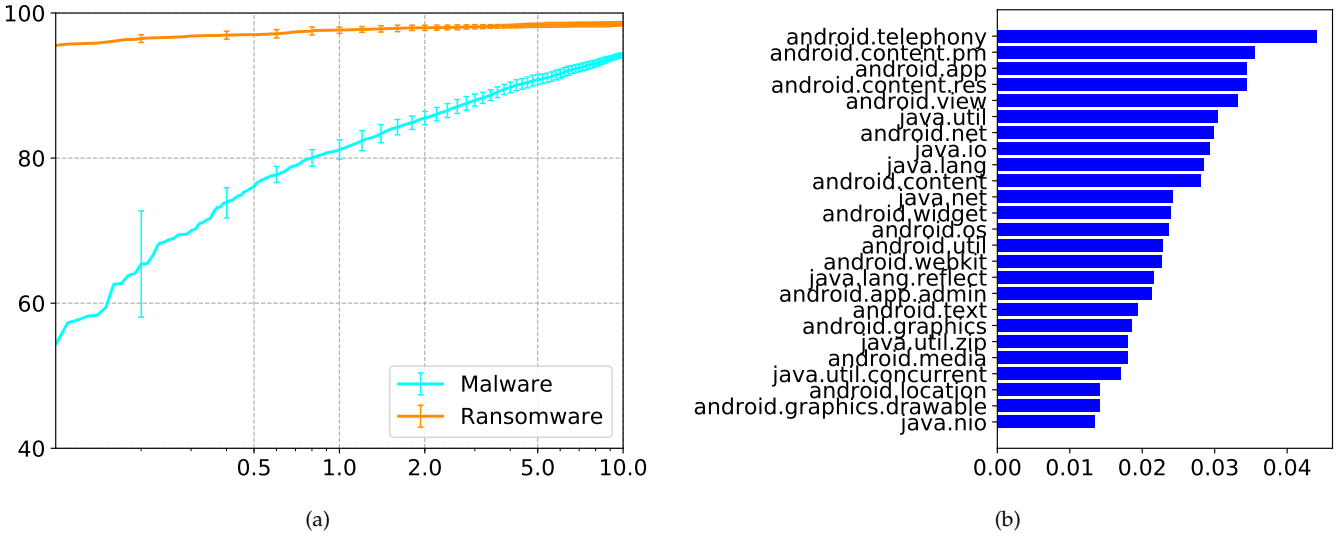

Fig. 3: Results for the experiment 1. On the left, we report the ROC curve (averaged on 5 splits) for ransomware and generic malware. On the right side, we report the top-25 features, ordered by the classifier information gain (calculated by averaging the gains, for each feature, that were obtained by training the 5 splits).

6.3 Experiment 2: Resilience against Obfuscation

The goal of this experiment was assessing the robustness of `R-PackDroid` against obfuscated samples, i.e., understanding whether the application of commercial tools to samples could influence the detection capability of `R-PackDroid`. This is important, as commercial obfuscation tools are quite popular nowadays, since they introduce good protection layers against static analysis (e.g., to avoid pieces of legitimate applications to be copied). Previous works showed that attackers could exploit this aspect by obfuscating malware samples with such tools, thus managing to bypass anti-malware detection [13].

In this experiment, we especially focused on obfuscated samples whose original (i.e., non-obfuscated) variant was already included in the training set. This is because we wanted to assess if obfuscation was enough to influence the key-features of `R-PackDroid`, thus *changing the classifier decision for a sample whose original label was malicious*.

The easiest and most complete way to perform such assessment was employing as test-bench the `Android PRAGuard` dataset [13], in which the entire Android Genome data [22] was obfuscated (with the commercial tool `DexGuard`) multiple times with different techniques of various complexity, leading to 8926 test samples. Such techniques can be summarized as follows:

- **Trivial**. It renames the name of packages, classes and methods that *are implemented by the user*.
- **String Encryption**. It encrypts strings that are related to `const-string` instructions, and injects a user-implemented method that performs decryption at runtime.
- **Reflection**. It transforms calls to *user-implemented* methods to a sequence of instructions that make massive usage of the `java.lang.reflect` API.
- **Class Encryption**. This call encrypts user-implemented classes, and injects routines that allow to perform dynamic loading of such classes.

We used as training set the whole dataset of malware (including the original Genome set), ransomware and goodware samples (the false positives threshold was set to 1%), and reported the attained results in Figure 4. The first thing to point out is that almost all obfuscation strategies, with the exception of Class Encryption, did not perform any changes to system API packages. This was expected, as obfuscation is in general used to conceal what the author of the application directly developed. Hence, `R-PackDroid` was able to detect all samples that were obfuscated with all the employed techniques with the exception of Class Encryption. However, the accuracy at detecting samples obfuscated with encryption only decreased by 7%. This showed that, despite the fact that most method calls (even related to system API packages) were concealed, *some package-related features were still extracted by our system*.

To demonstrate this, Figure 5 reports the distribution (in terms of occurrence) of the top-5 packages in Figure 3 for regular malware belonging to the Genome dataset, and for the same set obfuscated with class encryption. The Figure shows how class encryption made the occurrence of such packages decrease. However, this effect did not occur in all malware samples, and features such as `android.view`, `android.app` and `java.util` were still present in the majority of malware samples. This is also due to the fact that, as reported by [13], entry-point classes (i.e., the ones referred by the `AndroidManifest.xml` file) did not get encrypted by the obfuscator. This is because such changes would require modifying the `AndroidManifest.xml` accordingly, leading to a much more complicated obfuscation process. Moreover, introducing encryption of classes may add other system packages that can be related to suspicious or malicious activities. For these reasons, some of the API packages could still be detected by `R-PackDroid` and treated as features.

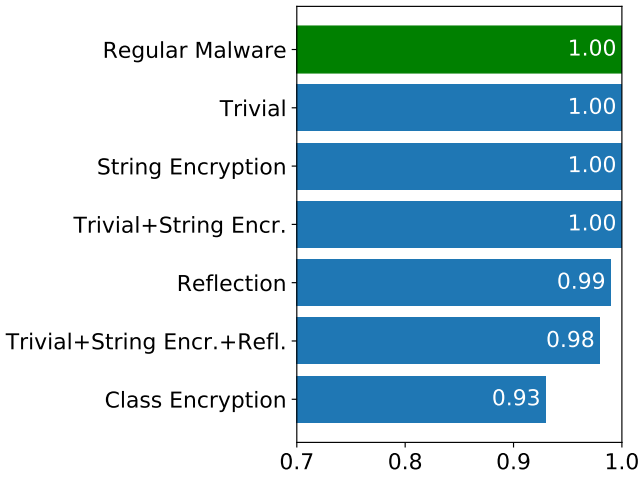

Fig. 4: Accuracy comparison for the obfuscated files of the Android PRAGuard dataset.

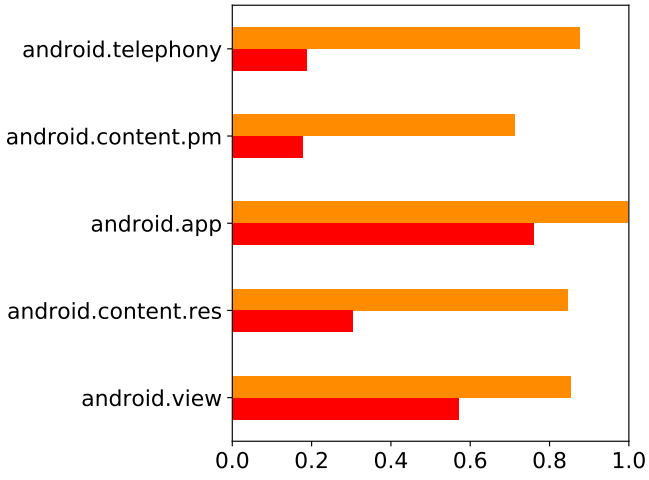

Fig. 5: Top-5 feature distribution, ordered by the classifier information gain, for regular malware samples (orange) and obfuscated ones with class encryption (red).

6.4 Experiment 3: Temporal Performances

In this experiment, we assessed the capabilities of `R-PackDroid` at detecting ransomware samples that were first seen (according to the `VirusTotal` service) *after* the data that were used as training set. This assessment is useful to understand if, without constant upgrades to its training set, `R-PackDroid` would be able to detect novel, unseen ransomware samples.

For this assessment, we included in the training set (along with *all generic malware and trusted samples*) ransomware samples that were first seen before a date D_{tr}, and we tested our system on a number of ransomware samples that were released on a date D_{te} for which $D_{te} > D_{tr}$ (the false positive threshold was set to 1%). We performed our tests by choosing different values of D_{tr} and D_{te}. Results are provided in Table 3.

Before discussing the attained results, it is worth noting that we were able to retrieve only a little amount of samples whose first release date was between January and September 2017. Conversely, we were able to retrieve a consistent amount of samples whose D_{te} was November 2017. Hence, we considered three main ranges for D_{te}: *(i)* January to September 2017; *(ii)* October 2017; *(iii)* November 2017. According to these values, we set the corresponding value of D_{tr} to December 2016 and September 2017.

TABLE 3: `R-PackDroid` predicted ransomware for different training (D_{tr}) and test (D_{te}) temporal intervals. The number of the samples for each test is reported in brackets.

Train Date (D_{tr})	**Test Date (D_{te})**		
	01 ÷ 09-2017	**10-2017**	**11-2017**
Up to Dec. 2016	60 (118)	93 (137)	685 (1055)
Up to Sept. 2017	/	130 (137)	928 (1055)

Table 3 shows that, by training the system with data retrieved in 2016, we were able to detect more than 60% of new ransomware samples. Worth noting, the percentage related to November 2017 is more significant than the ones related to the other months, due to the consistent difference in the number of samples. Although the attained accuracy might seem to be quite limited, it is important to remember that the training data was referred to the previous years, and therefore they might be significant differences in the way test samples were structured. This is further confirmed by the fact that, by training the system with data till September 2017 (meaning that we only add a moderate quantity of samples) the accuracy for the files released in November 2017 greatly increased.

To sum up, this experiment proved that `R-PackDroid` was able to predict, with good accuracy, samples that were publicly released after the ones employed by the training set.

6.5 Computational Performances

We analyzed the computational performances of `R-PackDroid` by running it both on X86 and Android environments. In particular, we focused on extracting the time interval between the `.apk` loading and the generation of the feature vector for 100 benign samples (grouped by their `.apk` size)[12]. The choice of benign samples was due to the fact that they are typically more complex to be analyzed in comparison with generic malware and ransomware. We first ran our experiments on a 24-core Xeon machine with 64 GB of RAM. The attained results, showed in Figure 6, show that our system can analyze even very big applications in less than 0.2 seconds.

To evaluate the performances of `R-PackDroid` on a true Android phone, we ran the same analysis on a Nexus 5, a nearly 5 years old, 4-core device with 2 GB of RAM. Results are reported in Figure 7. Even if the analysis times were obviously slower than X86 machines, and even if we were using in this case the slowest version of the algorithm, the average analysis time for very large apps was slightly more than 4 seconds. This is a very encouraging result, which showed that `R-PackDroid` could be safely used even on old phones. The higher dispersion of the time values, in comparison to the ones attained in the previous picture,

12. The elapsed time to classify a sample, i.e., to read its feature vector and get the final label, is negligible.

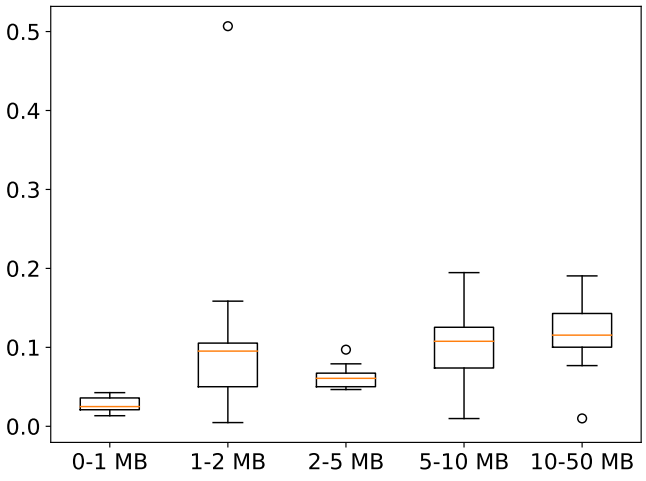

Fig. 6: Analysis performances on a X86 workstation, with the elapsed time in seconds, for different `.apk` sizes.

is possibly caused by the presence of other background processes in the device.

Finally, it is also important to observe that the analysis time is not strictly proportional to the `.apk` size, as the file may contain additional resources (e.g., images) that increase the `.apk` size, without influencing the size of the `DexCode` itself. For this reason, it was not surprising to see the attained average values did not necessarily increase with the `.apk` size.

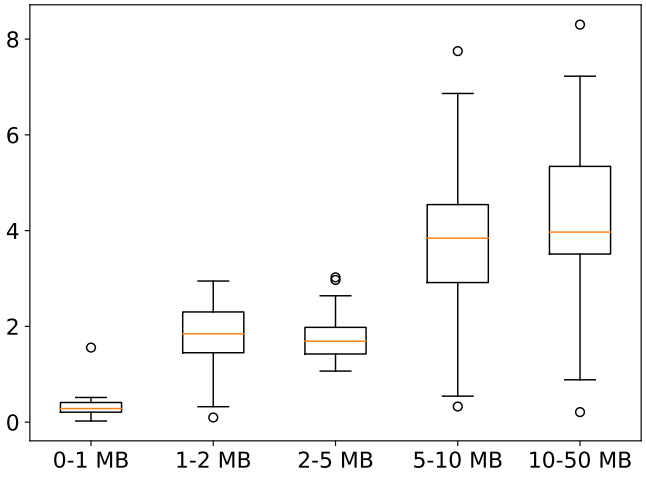

Fig. 7: Analysis performances on a real device, with the elapsed time in seconds, for different `.apk` sizes.

6.6 Comparison with Other Approaches

This section proposes a comparison between `R-PackDroid` and other state-of-the-art approaches. We are particularly interested in comparing our approach to other publicly available ones, with a special focus on those who were specifically designed to detect ransomware.

The state-of-the-art approach that is closest to what we proposed in this paper (while being publicly available[13]) is `GreatEatlon`. Notably, it was not possible for us to control the trained model of the system (it was only possible to choose among a restricted set of classifiers), or to train it with new data. For this reason, it would have been very hard to fairly define the performances of `GreatEatlon` in the same way we did in Section 6.2. Nevertheless, the system was released in 2016, meaning that data that were first seen in 2017 were for sure not included in its training set. Hence, we could make a comparison on the accuracy attained by `R-PackDroid` and `GreatEatlon` on ransomware samples released in 2017. For the sake of a fairer comparison, we trained `R-PackDroid` with data released until 2016, in the same way proposed in Section 6.4.

Before discussing the attained results, we point out that the available implementation of `GreatEatlon` provides the same labels of a *generic malware-oriented* system (malicious/benign), even if its features are specifically designed to detect ransomware. Conversely, `R-PackDroid` is a true *tri-class* (ransomware/malware/benign) system. This means that `R-PackDroid` may potentially make more mistakes than `GreatEatlon`, but it is much more precise in its detection. As classifier for `GreatEatlon`, we chose SGD, as this was the classifier that best performed on our test samples. Notably, we focused on those ransomware samples which could be properly analyzed by `GreatEatlon` (a portion of them could not be analyzed due to issues to their Manifest files), leading to 976 ransomware test samples. Results are reported in Table 4.

TABLE 4: Detection performances for `R-PackDroid` and `GreatEatlon` on 976 ransomware test files released in 2017, by using training data from 2016.

System	Benign	Generic Malware	Ransomware
R-PackDroid	123	68	785
GreatEatlon	225	751	0

The attained results were very encouraging. `R-PackDroid` was able to correctly recognize 785 out of 976 ransomware test samples, while 68 were labeled as generic malware and 123 as benign. Notably, `GreatEatlon` reported no ransomware detected because that class was not present in the system, and labeled 751 samples as generic malware. It is also important to point out that, if we had considered malware and ransomware as a unique malicious category for `R-PackDroid`, the attained detection rate would have been even higher.

Worth noting, the goal of this experiment was not to provide a definitive claim on which system was better than the other (without controlling its training data, it would be impossible to perform a fully fair comparison), but to show at least that `R-PackDroid` could be used as a valid alternative to other state-of-the-art approaches.

7 Discussion and Limitations

With the experiments provided in this paper, we demonstrated that `R-PackDroid` could be used as a valid aid to detect not only the majority of ransomware samples in the wild, but also novel, previously unseen ones. We point out that the goal of our system is not replacing commercial solutions or more general-purpose systems, but providing

13. https://github.com/necst/heldroid

something that can be used as a valid integration to current existing solutions.

Another important goal of this work was showing that using a compact set of information is often more than enough to detect a wide variety of samples. Many research works attempted to analyze the highest quantity of information possible (e.g., by extracting features from the `AndroidManifest.xml`). However, such choice would not only increase the computational complexity of the analysis (making the system less suitable to work on phones), but would also open doors to easy manipulations of the application, aimed at evading the system. In this sense, our work wants to encourage the usage of a reasonably low number of features to detect ransomware and other attacks. Of course, it is always possible that an attacker reimplements specific routines in order to avoid using system API packages. However, this may require a lot of effort from his side, and might not be feasible.

Our system also well-behaves against samples that have been obfuscated with commercial tools. However, more research should be done to understand what would happen if the application was crafted by employing strategies inherited from Adversarial Machine Learning (such as [6]). In this case, only few, limited changes to the application would suffice to attain evasion. However, this may not be so easy for a number of reasons: first, injecting packages related to system API calls in the current flow of the program (i.e., without resorting to dead code) may change (or even destroy) the whole user experience; second, performing fine-grained, automatic injection of multiple packages inside the program flow may be a non-trivial task for the attacker; finally, to make the system more robust, the feature set may be reduced by only considering those features whose increment would make the application more malicious. We plan to inspect and discuss these aspects in a future work.

Finally, it is also worth noting that, during our tests, we found samples that could not be analyzed due to crashes and bugs of the `DexLib` library, and that have therefore been excluded from our analysis. However, their percentage (with respect to the whole corpus that we analyzed) is negligible.

8 Conclusions

In this work, we proposed `R-PackDroid`, an application that performs ransomware and generic malware detection directly on Android devices by analyzing information extracted from system API packages. The choice of such packages allowed for obtaining a compact set of features, and for reducing the information that an attacker could easily manipulate to evade the system. We tested the effectiveness of our approach on a wide variety of benign, ransomware and generic malware samples, thus obtaining excellent results at ransomware detection, and very good performances at generic malware detection. We demonstrated that `R-PackDroid` is resilient against samples that have been obfuscated with commercial programs, and that it could predict previously unseen samples with good accuracy, by using training data that could even belong to previous years. Finally, we showed that `R-PackDroid` can be used as a valid alternative to other state-of-the-art approaches. `R-PackDroid` is available on the Google Play Store. Our hope is that publicly releasing our tool would help not only researchers, but also end-users at preventing and fighting ransomware infections.

Acknowledgements

This work was partially supported by the H2020 EU funded project *NeCS* [GA #675320], by the H2020 EU funded project *C3ISP* [GA #700294], and by the *PISDAS* project, funded by the Sardinian Regional Administration (CUP E27H14003150007).

References

[1] M. Ahmadi, A. Sotgiu, and G. Giacinto. Intelliav: Toward the feasibility of building intelligent anti-malware on android devices. In A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, editors, *Machine Learning and Knowledge Extraction*, pages 137–154, Cham, 2017. Springer International Publishing.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[3] N. Andronio, S. Zanero, and F. Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *Recent Advances in Intrusion Detection (RAID)*, pages 382–404. Springer, 2015.

[4] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto. Clustering android malware families by http traffic. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 128–135, Oct 2015.

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proc. 21st Annual Network & Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[6] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD), Part III*, volume 8190 of *LNCS*, pages 387–402. Springer Berlin Heidelberg, 2013.

[7] G. Canfora, F. Mercaldo, and C. A. Visaggio. An hmm and structural entropy based detector for android malware: An empirical study. *Computers & Security*, 61:1–18, 2016.

[8] J. Chen, C. Wang, Z. Zhao, K. Chen, R. Du, and G.-J. Ahn. Uncovering the face of android ransomware: Characterization and real-time detection. *IEEE Trans. on Information Forensics and Security (TIFS)*, 13(5):1286–1300, May 2018.

[9] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 377–388, New York, NY, USA, 2016. ACM.

[10] A. Cimitile, F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio. Talos: no more ransomware victims with formal methods. *International Journal of Information Security*, pages 1–20, 2017.

[11] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Trans. Dependable and Secure Computing*, In press.

[12] A. Gharib and A. Ghorbani. Dna-droid: A real-time android ransomware detection framework. In Z. Yan, R. Molva, W. Mazurczyk, and R. Kantola, editors, *Network and System Security: 11th International Conference, NSS 2017, Helsinki, Finland, August 21–23, 2017, Proceedings*, pages 184–198, Cham, 2017. Springer International Publishing.

[13] 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(C):16–31, June 2015.

[14] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli. R-packdroid: Api package-based characterization and detection of mobile ransomware. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1718–1723, New York, NY, USA, 2017. ACM.

[15] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio. Ransomware steals your phone. formal methods rescue it. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 212–221. Springer, 2016.

[16] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *Recent Advances in Intrusion Detection (RAID)*, volume 9854 of *Lecture Notes in Computer Science*, pages 230–253. Springer, 2016.

[17] S. Song, B. Kim, and S. Lee. The effective ransomware prevention technique using process monitoring on android platform. *Mobile Information Systems*, 2016.

[18] Symantec. Internet security threat report vol. 23, 2018.

[19] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*. The Internet Society, 2015.

[20] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao. Automated detection and analysis for android ransomware. In *2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS)*, pages 1338–1343. IEEE, 2015.

[21] C. Zheng, N. Dellarocca, N. Andronio, S. Zanero, and F. Maggi. Greateatlon: Fast, static detection of mobile ransomware. In *SecureComm*, volume 198 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 617–636. Springer, 2016.

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

Michele Scalas received the M.Sc. degree (Hons.) in Telecommunications Engineering from the University of Cagliari in July 2017, disputing a thesis entitled "Study and Development of an Android Application for Automatic Ransomware Detection". Since October 2017 he is a Ph.D. student in Electronic and Computer Science Engineering at the University of Cagliari. His research interests include machine learning, Android malware and automotive cybersecurity.

Davide Maiorca (M'16) received the M.Sc. degree (Hons.) in Electronic Engineering from the University of Cagliari, Italy, in 2012. He is a Ph.D. Student in Electronic Engineering and Computer Science at the University of Cagliari, Italy. In 2013, he visited the Systems Security group at Ruhr-Universität Bochum, guided by Prof. Dr. Thorsten Holz, and worked on advanced obfuscation of Android malware. His current research interests include adversarial machine learning, malware in documents and Flash applications, Android malware and mobile fingerprinting. He has been serving as reviewer and program committee member for various conferences and journals.

Francesco Mercaldo received his master degree in computer engineering from the University of Sannio (Benevento, Italy), with a thesis in software testing. He obtained his Ph.D. in 2015 with a dissertation on malware analysis using machine learning techniques. The research areas of Francesco are software testing, verification, and validation, with the emphasis on the application of empirical methods. Currently, he is working as post-doctoral researcher at the Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche (CNR) in Pisa (Italy). He is also involved as lecturer in Database, Web and Mobile Programming, Operating Systems (Bachelor Degree) and Software Security (Master Degree) courses at the University of Molise (Italy).

Corrado Aaron Visaggio obtained the PhD in computer engineering at University of Sannio (Italy). He is assistant professor of Software Security of the MsC in Computer Engineering at the University of Sannio, Italy. His research interests include: empirical software engineering, software security, and data privacy. He is author of more than 50 papers published on international journals, international and national conference's proceedings, and books. He serves in many program committees and editorial boards of international conferences and journals.

Fabio Martinelli received the M.Sc. degree from the University of Pisa, Pisa, Italy, in 1994 and the Ph.D. degree from the University of Siena, Siena, Italy, in 1999. He is currently a Senior Researcher with the Institute of Informatics and Telematics of the Consiglio Nazionale delle Ricerche, Pisa, Italy, where he leads the Cyber Security Project. He has co-authored more than 200 papers in international journals and conference or workshop proceedings. He is involved in several steering committees of international WGs or conferences and workshops. He manages research and development projects on information and communication security. His main research interests include security and privacy in distributed and mobile systems and foundations of security and trust.

Giorgio Giacinto (SM'10) is Professor of Computer Engineering at the University of Cagliari, Italy. Since 1995 he joined the research group on Pattern Recognition and Applications, in which he leads the Computer Security unit. His research interests are in the field of pattern recognition and machine learning for malware analysis and detection, web application security, and phishing detection. He is also the Erasmus+ academic contact point for the Faculty of Engineering and Architecture. Prof. Giacinto has been serving either as the coordinator, or as a member of the technical management board, in many R&D projects at the local, national and European level. In 2015, he co-founded the spin-off company Pluribus One that is bringing to the market the most valuable products of the activity carried out by the research group. Prof. Giacinto is author of around 150 scientific papers in international journals and conferences. He is a senior member of both the IEEE and ACM.