On the Effectiveness of System API-Related Information for Android Ransomware Detection

Michele Scalas[a,*], Davide Maiorca[a], Francesco Mercaldo[b], Corrado Aaron Visaggio[c], Fabio Martinelli[b], Giorgio Giacinto[a]

[a] *Department of Electrical and Electronic Engineering, University of Cagliari, Piazza dArmi 09123, Cagliari, Italy*
[b] *Institute of Informatics and Telematics, National Research of Council, Pisa, Italy*
[c] *Department of Engineering, University of Sannio, Benevento, Italy*

Abstract

Ransomware constitutes a significant threat to the Android operating system. It can either lock or encrypt the target devices, and victims are forced to pay ransoms to restore their data. Hence, the prompt detection of such attacks has a priority in comparison to other malicious threats. Previous works on Android malware detection mainly focused on Machine Learning-oriented approaches that were tailored to identifying malware families, without a clear focus on ransomware. More specifically, such approaches resorted to complex information types such as permissions, user-implemented API calls, and native calls. However, this led to significant drawbacks concerning complexity, resilience against obfuscation, and explainability. To overcome these issues, in this paper, we propose and discuss learning-based detection strategies that rely on System API information. These techniques leverage the fact that ransomware attacks heavily resort to System API to perform their actions, and allow distinguishing between generic malware, ransomware and goodware. We tested three different ways of employing System API information, i.e., through packages, classes, and methods, and we compared their performances to other, more complex state-of-the-art approaches. The attained results showed that systems based on System API could detect ransomware and generic malware with very good accuracy, comparable to systems that employed more complex information. Moreover, the proposed systems could accurately detect novel samples in the wild and showed resilience against static obfuscation attempts. Finally, to guarantee early on-device detection, we developed and released on the Android platform a complete ransomware and malware detector (R-PackDroid) that employed one of the methodologies proposed in this paper.

Keywords: Malware, Android, Ransomware, Machine Learning, Security

1. Introduction

The term `ransomware` refers to attacks that lock the victim's device or encrypt its data, by asking a sum of money to restore the compromised functionality. Despite the increasing diffusion of cloud-based technologies, users still store the majority of their data directly on their devices. For this reason, such attacks are particularly devastating, as they could destroy sensitive data of private users and companies (which often neglect to make backups of sensitive data). 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 [1].

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 the police currently control them for their actions (a strategy directly inspired by *scareware*-based approaches).

To counteract such attacks, Machine Learning has been increasingly used (especially combined with static analysis) both by researchers and anti-malware companies, either to perform direct detection or to generate signatures. While the goal of static detection systems is often to discriminate between generic malware and legitimate files (as in [2, 3, 4, 5, 6]), some recently-released ones focus on further identifying malware families, in particular ransomware-related ones (also known as *ransomware-oriented detection*) [7, 8, 9]. The reason for such a choice is that ransomware infections may lead to permanent data loss, making their early detection critical.

The main characteristic of systems to detect malware

*Corresponding author
Email addresses: `michele.scalas@unica.it` (Michele Scalas), `davide.maiorca@diee.unica.it` (Davide Maiorca), `francesco.mercaldo@iit.cnr.it` (Francesco Mercaldo), `visaggio@unisannio.it` (Corrado Aaron Visaggio), `fabio.martinelli@iit.cnr.it` (Fabio Martinelli), `giacinto@unica.it` (Giorgio Giacinto)

families is that they rely on different types of information extracted from multiple parts of the apps (e.g., bytecode, manifest, native libraries, and so forth [3, 7, 8, 4, 9]), which leads to use large amounts of features (even hundreds of thousands). While this approach is tempting and may seem to be effective against the majority of attacks in the wild, it features various limitations. First, it is unclear which features are essential (and needed) for classification, an aspect that worsens the overall explainability of the system (i.e., why the system makes mistakes and how to fix them). Second, increasing the types of features 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 specific IP address, if the attacker understood that this has a vital role for detection [2]. Finally, the computational complexity of such systems is enormous, which makes them unfeasible to be practically used in mobile devices, an important aspect to guarantee offline, early detection of these attacks.

In a previous work [10], we proposed a detection methodology (`R-PackDroid`) that allowed to discriminate between ransomware, generic malware, and legitimate files by focusing on a small-sized feature set, i.e., System API packages. The idea of our work was to overcome the limitations described above by showing that it was possible to solve a machine learning problem with a limited number of features of the same type. However, System API-based information does not only include packages but also classes and methods (particularly employed in other works, especially mixed with other feature types [3, 9]) that better define the behavior of APIs. Intuitively, using finer-grained information leads to better accuracy and robustness in comparison to other approaches. In this paper, we explore such a possibility by progressively refining the System API-based information employed in our previous work [10]. In particular, *we inspected the capabilities of multiple types of System API-related information to discriminate ransomware from malware and goodware*. More specifically, we aimed to provide an answer to the following Research Questions:

- **RQ 1.** Does the use of finer-grained information related to System API (i.e., classes and methods) improve detection performances in comparison to more general System API packages?
- **RQ 2.** Is System API-based information suitable to detect novel attacks in the wild?
- **RQ 3.** Does using System API-based information provide comparable performances to other approaches that employ multiple feature types?
- **RQ 4.** Is System API-based information resilient against obfuscation attempts?

To answer such Research Questions, we explored three types of System API-based information: the first one only used information related to System API packages (as already shown in [10]), the second one analyzed System API classes, and the third one employed information related to System API methods. We evaluated the performances of the three systems on a wide range of ransomware, malware and goodware samples in the wild (including previously unseen data). Moreover, we tested all systems against a dataset of ransomware samples that have been obfuscated with multiple techniques (including class encryption).

The attained results showed that all System API-based techniques provided excellent accuracy at detecting ransomware and generic malware in the wild, by also showing capabilities of predicting novel attacks and resilience against obfuscation. More specifically, using finer-grained information even improved the accuracy at detecting previously unseen samples, and provided more reliability against obfuscation attempts. From a methodological perspective, such results demonstrate that it is possible to develop accurate systems by strongly reducing the complexity of the examined information and by selecting feature types that represent how ransomware attacks behave.

Finally, to demonstrate the practical suitability of System API-based approaches on Android devices, we ported to Android `R-PackDroid` (the package-based strategy originally proposed in [10] and further explored in this work). Our application, which can detect both ransomware and generic malware in the wild, shows that methodologies based on System API can be implemented with good computational performances even in old phones, and its a demonstration of a full working prototype being deployed on real analysis environments. `R-Packdroid` can be downloaded for free from the Google Play Store[1].

With this work, we claim that it is possible to create effective, deployable, and reasonably secure approaches for ransomware and malware detection by only using specific feature types. Hence, we believe that the attention of research should be shifted to finding effective and explainable feature types to make detection even more accurate and robust.

Paper structure. Section 2 provides the basic concepts of Android apps; Section 3 discusses the essential characteristics of Android ransomware and describes the key-intuitions behind using System API calls as critical information; Section 4 provides a description of the related work in the field; Section 5 describes the employed detection methodologies; Section 6 illustrates the experimental results attained with all the methodologies, as well as a comparison between our systems and other approaches in the wild; Section 7 describes the implementation details of `R-PackDroid` and its computational performances; Section 8 discusses the limitations of our work, which is finally concluded by Section 9.

[1] `http://pralab.diee.unica.it/en/RPackDroid`

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 bytecode format called `Dalvik` (which is further referred to as `DexCode`), whose instructions are contained in the `classes.dex` file. This file is 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

The key point presented in this work is that the static extraction of System API-based information can be effective at detecting ransomware. More specifically, System APIs encapsulate many of the key actions performed by such attacks. To better reinforce this concept, in the following, we describe the basic actions performed by Android 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 [11], 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 continuously shown; *(ii)* by setting up specific parameters of specific 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 pornographic material. There are, however, samples that also perform data encryption. According to [11], 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.

```
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.

As locking and encryption actions require the use 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*. It would be extremely time consuming and 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).

In an Android application, based on Java, multiple methods are associated with classes that belong to packages. Because of these characteristics, it is possible to encode and represent System API information by either using packages, classes, or methods. More specifically, methods and classes better detail the functionality performed by the single API, but their number is significantly higher in comparison to packages. Hence, a solution that would employ the analysis of API methods would be far more complex than one that analyzes packages.

[2]MD5: `0cdb7171bcd94ab5ef8b4d461afc446c`
[3]MD5: `59909615d2977e0be29b3ab8707c903a`

```
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 scope of this work is mostly oriented to ransomware detection, this Section will be mainly focused on describing systems that aim to detect such attacks (*ransomware-oriented* detectors) specifically. A brief description of the other detectors will be provided at the end of this Section.

The most popular and publicly available *ransomware-oriented* detector is `HelDroid`, proposed by Andronio et al. [7]. This tool includes a text classifier (based on NLP features) that works on suspicious strings used by the application, 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. [8] 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. However, 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 employing, e.g., string encryption [12]. Moreover, it strongly depends on the presence of a language dictionary for that specific ransomware campaign.

Yang et al. [16] 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.

Song et al. [15] proposed a method that aims to discriminate between ransomware and goodware using process monitoring. In particular, they considered system-related features representing the I/O rate, as well as the CPU and memory usage. The system has been evaluated with only one ransomware sample developed by the authors, and no implementation is publicly available.

Cimitile et al. [13] introduced an approach to detect ransomware that is based on formal methods [17, 18](by using a tool called `Talos`), which help the analyst identify malicious sections in the app code. In particular, starting from the definition of payload behavior, the authors manually formulated logic rules that were later applied to detect ransomware. Unfortunately, such a procedure can become extremely time-consuming, as an expert should manually express such rules.

Gharib et al. [14] 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. 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. This approach has the drawback that heavily obfuscated apps can escape the static filter, thus avoiding to be dynamically analyzed. Finally, Chen et al. [11] 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 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.

Concerning *generic malware-oriented* detectors, Arp et al. [3] 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, and so forth. Tam et al. [19] introduced a system to perform dynamic analysis and detection of Android malware by analyzing the system calls performed by the application. Avdieenko et al. [20] used taint analysis to detect anomalous flows of sensitive data, a technique that allowed to detect novel malware samples without previous knowledge. Yang et al. [21] analyzed malicious apps by defining and extracting the context related to security-sensitive events [22, 23]. In particular, the authors defined a model of context based on two elements: activation conditions (i.e., what makes specific events occur) and guarding conditions (i.e., the environmental attributes of a specific event).

Aresu et al. [24] clustered Android malware by using the network HTTP traffic generated by those applications. Such clusters can be used to generate signatures that allow discriminating between malware and legitimate applications. Canfora et al. [25] 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. The attained results showed that both techniques could be suitable for Android

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

Work	Tool	Year	Static	Dynamic	Machine-Learning	Available
Chen et al. [11]	RansomProber	2018		✓		
Cimitile et al. [13]	Talos	2017	✓			
Gharib et al. [14]	Dna-Droid	2017	✓	✓	✓	
Song et al. [15]	/	2016		✓		
Zheng et al. [8]	GreatEatlon	2016	✓		✓	✓
Yang et al. [16]	/	2015		✓		
Andronio et al. [7]	HelDroid	2015	✓		✓	✓

malware detection.

Chen et al. [4] proposed `StormDroid`, a static and dynamic machine-learning based system that extracts information from API-calls, permissions and behavioral features. Ahmadi et al. [5] proposed `IntelliAV`, a *generic malware-oriented* detector that is publicly available. Such a detector provides a level of dangerousness for each app but does not directly specify the family nor the type of attack.

Garcia et al. [9] proposed `RevealDroid`, a static system for detecting Android malware samples and classifying them in families. The system employs features extracted from reflective calls, native APIs, permissions, and many other characteristics of the file. The attained results showed that `RevealDroid` was able to attain very high accuracy, resilience to obfuscation. However, the number of extracted features can be extremely high and depends on the training data.

Zhang et al. [26, 27] leveraged machine learning-based solutions that employed dependency graphs extracted by observing network events. In particular, in [26], they proposed a traffic analysis method that employed scalable algorithms for the detection of malware activities on a host. Such a detection was performed by exploring request-level traffic and the semantic relationships among network events. The attained results showed high accuracy at detecting spyware, DNS bot, and data exfiltrating malware. In [27], they employed a learning-based solution that analyzed information extracted from the dynamic analysis of the network events generated by Android malware. In particular, they profiled the traffic generated by benign applications and modeled (through graphs called *triggering relation graphs*) the triggering relationship of the generated network events (i.e., how such events are related to each other) to identify anomalous, malicious ones. The authors demonstrated that these graphs could be particularly useful to detect suspicious network requests.

Finally, for the sake of completeness, we mention here other recent works that have analyzed the topic of ransomware detection on X86 (in particular, on Windows platforms) by employing dynamic analysis techniques to perform early detection of the attack. Using such techniques avoid possible damages to the operating system and its files. [28, 29, 30, 31].

5. Methodologies

We now describe the general structure of systems that employ System API information to identify ransomware, also known as *ransomware-oriented* detectors. While the majority of learning-based detection systems combine various types of information to detect as many attacks as possible, *ransomware-oriented* detectors tailor their detection on a smaller set of information (System API) that is typically employed in ransomware. However, as System APIs are also widely used in generic malware and legitimate files, this information type also allows detecting other attacks that differ to ransomware. In this way, it is possible to create a powerful, wide-spectrum detector that features a much lower complexity in comparison to other approaches. Typically, such systems take as input an Android application, analyze it and return 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`. The required information is extracted by only inspecting the executable code and does not perform any 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 (System API)**. In this phase, the code lines received from the previous phase are further analyzed to extract the related *System API* information (either packages, classes, or methods). The *occurrence* of such pieces of information is then counted, thus producing a vector of numbers (*feature vector*) that is sent to a classifier.

- **Classifier**. Classification is carried out through 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 excellent results [3, 2, 9]. In particular, our approaches employ Random Forest classifiers, which are especially useful to handle multi-class problems, and which are widely used for malware detection. The complexity of such classifiers depends

on the number of trees that compose them. Such a number must be optimized during the training phase.

The structure above is graphically represented in Figure 1. In the following, we provide more details about each phase of the analysis, by focusing in particular on the type of features that can be extracted from the application.

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 information* employed by the executable code of the application. The choice of System API information is related to two basic ideas:

- **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). Focusing on user-implemented APIs (as it happens, for example, with `Drebin` [3]) exposes the system to a risk of being evaded by simply employing different packages to perform actions.
- **Independence from Training**. System API calls are features independent of the training data that are used. As a consequence, it is less likely that applications are not correctly analyzed only because they employ never-seen-before APIs.
- **Resilience against obfuscation**. Using heavy obfuscation routines typically lead to injecting system API-based code in the executable, which can be extracted and analyzed, allowing to detect suspicious files.

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 an 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 invocation is inspected to extract the relevant API information for each methodology, according to a System API reference list that depends on the operating system version (in our case, Android Nougat - API level 25 - a widely-used API set). Only the API elements that belong to the reference list are analyzed. In this paper, we consider three different methodologies, based on, respectively, package, class, and method extraction. If a specific API element is found, its occurrence value is increased by one.

In the following, we provide a more detailed description of the methodologies employed in this paper, by referring to the example reported in Listing 3. The code is parsed in three ways, according to each feature extraction strategy. For each example, we used a very small subset of the employed reference API.

- **Packages Extraction.** In this methodology, we extract the occurrences of the System API packages (a total of 270 reference features), in the same way of our previous work [10]. In the example of Listing 3, we used a subset composed of three reference API packages: `java/io`, `java/crypto` and `java/lang`. The four `invoke` instructions are related to the `javax/crypto` and `java/io` packages, which are counted respectively twice. The `java/lang` package is never used in this snippet. Hence, its value is zero.
- **Classes Extraction.** In this methodology, we extract the occurrences of the System API classes (a total of 4609 reference features). Notably, such classes belong to the System API packages of the previous methodology (and, for this reason, their number is significantly higher than packages). In the example of Listing 3, we used a subset composed of two reference API classes: `java/io/FileInputStream` and `javax/crypto/CypherOutputStream`, each of them appearing twice.
- **Methods Extraction.** In this methodology, we extract the occurrences of the System API methods (a total of 36148 reference features). These methods belong to the System API classes of the previous methodology, leading to a very consistent number of features. This strategy is very similar to other ones employed by other systems (e.g.. [3, 9]), which have used these features together with user-implemented APIs and other features. In the example of Listing 3, we used a subset composed of four reference API methods: `java/io/FileInputStream/read`, `javax/crypto/CypherOutputStream/flush`, `javax/crypto/CypherOutputStream/close` and `java/io/FileInputStream/close`. Each API call appears only once. Note that, although there are two methods named *close*, they belong to two different classes, and they are therefore considered as different methods.

6. Experimental Evaluation

In this Section, we report the experimental results attained from the evaluation of the three API-based strategies. Note that, for the sake of simplicity and speed, we did not run the experiments on Android phones, but on an X86 machine. However, we built a full, working implementation of one of the three approaches, which can be downloaded from the Google Play Store (see next Section).

The rest of this Section is organized as follows: we start by providing an overview of the dataset employed in our

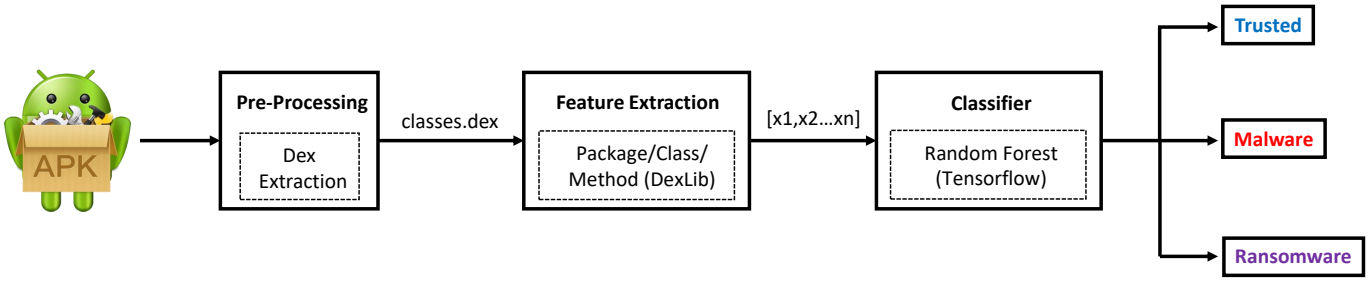

Figure 1: General Structure of a System API-based, ransomware-oriented system.

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

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

experiments. Then, we describe the results attained by four evaluations. The first one aimed to establish the general performances of API-based approaches by considering random distributions of training and test samples. The second one aimed to show how API-based approaches behaved when analyzing samples released after the training data. The third one aimed to show a comparison between our API-based approaches and other systems that employed mixed features. Finally, we evaluated the resilience of API-based approaches against obfuscation techniques and evasion attacks.

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 39 157 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[4] (which aggregates the detection of multiple anti-malware solutions) and from the `HelDroid` dataset[5] [7]. With respect to the samples obtained from `VirusTotal`, we used the following procedure to obtain the samples: *(i)* we searched 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 [32], which essentially combines the various labels provided by anti-malware solutions to create a unique label that identifies the sample itself; *(iii)* we considered only those samples whose family was coherent to ransomware behaviors, or was known to belong to ransomware.

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

In general, our goal was obtaining a representative corpus of ransomware to ascertain the prediction capabilities of API-based techniques. 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 families above, please see Section 3.

6.1.2. Malware and Trusted

We considered a dataset composed of 17 744 Android malware samples that do not belong to the ransomware

[4] `http://www.virustotal.com`

[5] `https://github.com/necst/heldroid`

category, taken from the following sources: *(i)* `Drebin` dataset, one of the most recent, publicly available datasets of malicious Android applications[6] (which also contains the samples from the Genome dataset [33]); *(ii)* `Contagio`, a popular free source of malware for X86 and mobile; *(iii)* `VirusTotal`. These samples were chosen to verify whether even non-ransomware attacks could be detected with features that are particularly effective at classifying ransomware samples.

In order to download trusted applications, we resorted to two data sources: *(i)* we crawled the Google Play market using an open-source crawler[7]; *(ii)* we extracted a number of applications from the `AndroZoo` dataset [34], which features a snapshot of the `Google Play` store, allowing to access applications without crawling the Google services easily. We obtained 18 396 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 System API-Based methods (described in Section 5) at detecting ransomware and generic malware. To do so, for each strategy, 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 each API-based strategy in two different cases:

- **Ransomware against benign samples**. The crucial goal of our work is detecting ransomware attacks and, more importantly, *to avoid them being considered as benign files*. A critical mistake would most likely compromise the whole device by locking it or encrypting its data. For this reason, it is essential to verify whether ransomware attacks can be confused with benign samples.
- **Generic malware against benign samples**. Even if System API-based strategies were employed to detect ransomware, they could also be used to classify generic malware (see Section 5). Hence, the goal here is to verify, from a practical perspective, if System API-based information can correctly detect other non-ransomware attacks and distinguish them from legitimate files.

[6] `https://www.sec.cs.tu-bs.de/~danarp/drebin/`
[7] `https://github.com/liato/android-market-API-py`

Results are reported in Figure 2. Parts (a) and (b) show the ROC curves that describe the performances attained on ransomware and generic malware detection by the three System API-based methods (packages, classes, methods). By observing these curves, we can deduce the following facts:

1. All System API-based techniques were able to precisely detect more than 97% of ransomware samples with only 1% of false positives. Because our dataset included a consistent variety of families, we claim that all strategies can detect the majority of ransomware families in the wild. Worth noting, there are no differences in results between using packages, classes or methods. This result means that, concerning general performances, using finer-grained features does not improve detection.
2. All System API methods featured good accuracy with relatively low false positives (around 90% at 1%, more than 95% at 2%) at detecting generic malware. While using class-related features did not bring significant improvements to detection, using methods allowed for a 10% improvement for false positive values inferior to 0.5%.

To better understand the results attained by our strategies, parts (c), (d) and (e) report a ranking of features used by the classifier for each strategy (respectively, packages, methods, and classes), according to their discriminant power. The feature ranking is calculated according to the features Information Gain IG, given by the following formulation:

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

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.

As a result, note how the information gain for each feature is not so high, meaning that the system does not particularly overfit on specific information and that the final decision is taken by considering a combination of multiple features. At the same time, each feature value is reduced, in comparison to packages, by one magnitude for classes and methods. In other words, using more features allows for distributing the importance of the analyzed information through more elements. This characteristic is two-faced: while it makes the overall behavior of the system less interpretable, it may increase the effort that an attacker has to make to evade the system.

Analyzing the most discriminant methods can give a clearer idea of which information is used to classify applications. Features are related to string building (e.g., the `ToString` method), Array management (e.g., `ArrayList@size`, `ArrayList@remove`), creation of folders (e.g., `File@mkdirs`), SMS, URI and Layout management,

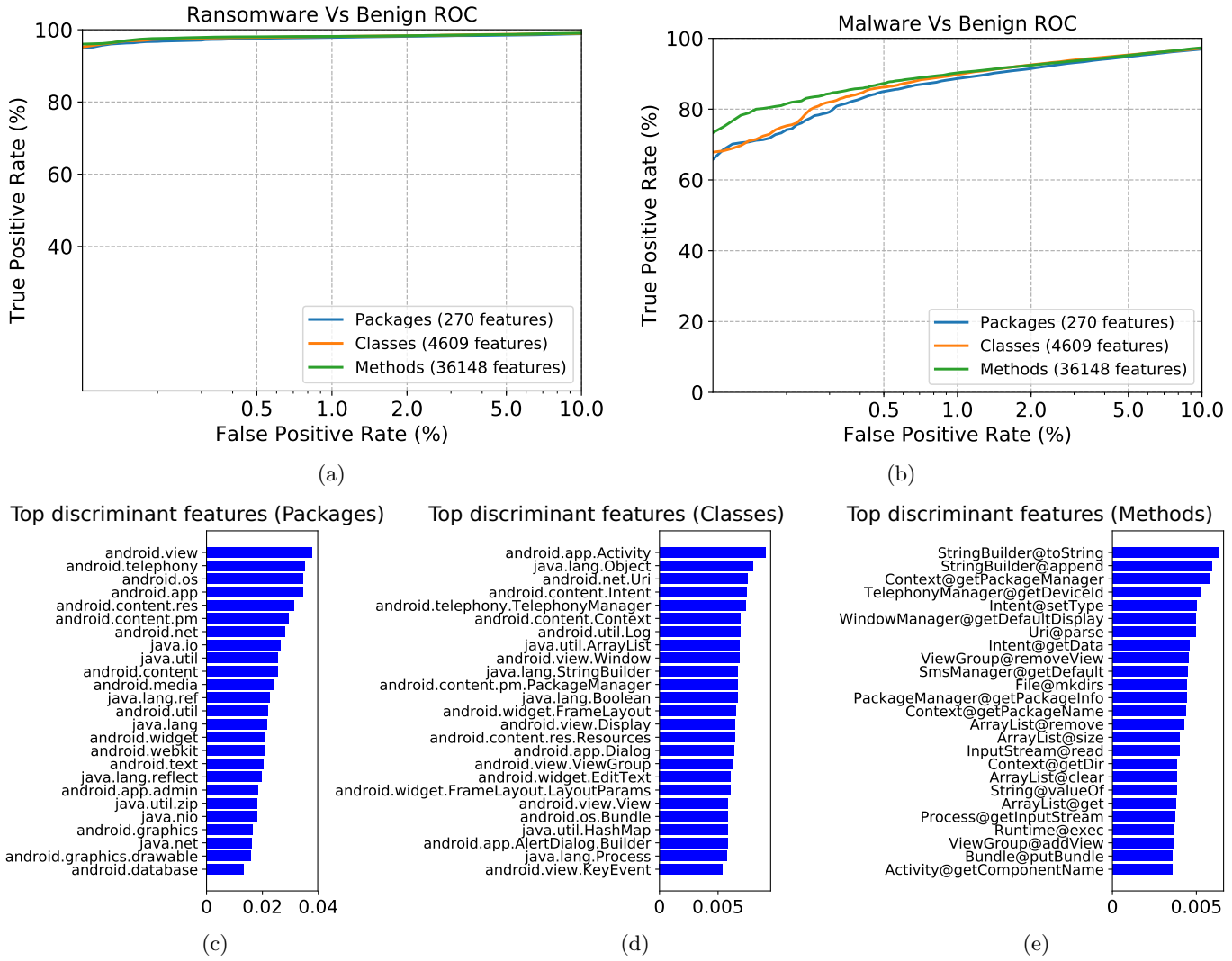

Figure 2: Results for the experiment 1. In the first row, we report the ROC curves (averaged on 5 splits) attained by the three System API methods for ransomware (a) and generic malware (b) against benign samples. In the second row, we report the top-25 features, ordered by the classifier information gain (calculated by averaging, for each feature, the gains that were obtained by training the 5 splits), for each methodology ((c) for packages, (d) for classes, (e) for methods).

and so forth. These features may be easily associated both to ransomware and malware behavior, and the same behavior is shown on classes and packages.

A careful examination of the feature ranking may also help to understand why specific samples are regarded as false positives or false negatives. In particular, detection is performed by weighting the information provided by a combination of the most discriminant features. For this reason, in some cases, specific ransomware (or generic malware) samples may contain discriminant features that are distributed differently to the malicious training distribution. For example, the `toString` call may appear, on average, 5 times on ransomware, but one specific sample may feature it only 1 time. This phenomenon may occur for various reasons, including the possibility that an attacker may be using customized variants of System API information to avoid detection (see Section 8). Likewise, the techniques used to create the sample (e.g., repackaging) may have an impact on the distribution of the features. Further refinement of the feature list (or a change of the weights of specific feature) may help to reduce the amount of misclassified samples.

6.3. Experiment 2: Temporal Performances

In this experiment, we assessed the capabilities of System API-based methods at detecting ransomware samples that were first seen (according to the creation date of the `classes.dex` executable belonging to each application) *after* the data that were used as training set. This assessment is useful to understand if, without constant upgrades to its training set, such methods would be able to detect novel, unseen ransomware samples.

Figure 3: Results of the temporal evaluation for the System API-based strategies. The accuracy values are reported for the three System API-based detection. The training data belong to 2016, while the test data is composed of ransomware released in different months of 2017.

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 $D_{te} > D_{tr}$ (we chose the ROC operating point of the system corresponding to a false positives value of 1%). We performed our tests by choosing different values of D_{te}, where D_{tr} is December 31st, 2016. Concerning test data, we point out that the samples (which were extracted by the `VirusTotal` service) were unevenly distributed through the months. More specifically, the number of ransomware samples that were submitted to the `VirusTotal` service was significantly different for each month of 2017. In particular, we could retrieve only a little amount of samples whose first release date was between January and September 2017. Conversely, we could obtain a consistent amount of samples whose D_{te} was October and November 2017. Therefore, we grouped the samples gathered in subsequent months to obtain temporal ranges with similar amounts of testing data. We considered three main ranges for D_{te}: *(i)* January to September 2017; *(ii)* October 2017; *(iii)* November 2017.

Results are provided in Figure 3, which shows that by training the system with data retrieved in 2016, class- and method-based strategies could accurately detect ransomware test samples released in 2017. However, the package-based strategy struggled at detecting the test-set from November 2017. Notably, in comparison with class- and method-based strategies, the package-based approach showed almost a 10% accuracy loss when analyzing samples released till October, and more than 20% accuracy loss for samples released in November. Conversely, the other two methods exhibited stable accuracy on each temporal range. This result is particularly interesting, as it shows that the prediction of novel samples can be significantly improved by employing finer-grained features. However, the results attained by package-based features are nevertheless encouraging, as they showed that even a reduced number of features could attain good performances at detecting novel attacks. Overall, this experiment further confirmed that System API-based strategies could predict new ransomware attacks with good accuracy, even on test data released. In this case, using finer-grained features brings a consistent advantage to detection.

6.4. Experiment 3: Comparison with Other Approaches

This section proposes a comparison between System API-based strategies and other state-of-the-art approaches. We were particularly interested in comparing our approach to other publicly available ones, with a special focus on those who were specifically designed to detect ransomware. Additionally, we considered those approaches that, albeit not explicitly designed to detect ransomware, could tell if the analyzed sample is malicious or not. In particular, almost all of the analyzed tools (except for `Talos` [13] - which does not employ Machine Learning) discriminate between two classes (malware and benign or ransomware and benign), while our approaches discriminate between three classes (ransomware, malware, and benign). Hence, it is interesting to observe how increasing the number of classes may impact the precision of the analysis.

We performed a temporal comparison of all systems on the ransomware samples released in 2017 (for a total of 512 samples) by using as training (when possible) all data released until 2016.

The state-of-the-art approach that is closest to what we proposed in this paper (while being publicly available[8]) is `GreatEatlon` [8]. 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. Nevertheless, the system was released in 2016, meaning that data that was first seen in 2017 was for sure not included in its training set. Although not specifically tailored to ransomware detection, we also tested the performances of `RevealDroid` (which is publicly available[9] [9]) on the same test data. In this case, we could train the system with the same data used in our systems, which allowed us to provide a fairer comparison. Finally, we also tested the performances of the Android version of `IntelliAV` (available on the Google Play Store) [5]. As in `GreatEatlon`, we could not control the training data of the system. Moreover, as `IntelliAV` reports three levels of risk for each app (safe, suspicious, risky), we considered as malicious also the files that were labeled as suspicious by the system.

[8] `https://github.com/necst/heldroid`
[9] `https://seal.ics.uci.edu/projects/revealdroid/`

As classifier for GreatEatlon we chose Stochastic Gradient Descent (SGD), since this was the classifier that best performed on our test samples. Concerning RevealDroid, we chose the linear SVM classifier, as this was the one that provided the best results in the original work [9]. IntelliAV only employed Random Forests. Results are reported in Table 3.

The attained results show that System API-based techniques obtained very similar performances to RevealDroid (which could only, however, classify samples either as malware or benign). Such results are particularly interesting if we consider that RevealDroid extracted a huge number of features (more than 700,000) from multiple characteristics of the file, including native calls, permissions, executable code analysis, which also depended on the training data. With a much simpler set of information, we were able to obtain very similar performance concerning accuracy. This result is especially interesting from the perspective of adversarial attacks, as using fewer features for classification can make the system more robust against them (the attacker can manipulate less information to evade the system) [35]. The performances attained by System API-based approaches were also better than IntelliAV, which employed a combination of different features (including permissions, user-defined API, and more). System API-based strategies also performed significantly better than GreatEatlon, which based its detection also on information extracted from strings and language properties. Notably, using methods significantly improved the accuracy performances in comparison to packages and classes, in line to what obtained from Experiment 2. Finally, we analyzed the performances attained by Talos [13][10], a static analysis tool that employs logic rules to perform detection (hence, without machine learning). The attained results were very encouraging, but they strongly depended on the set of rules that have been (manually) established to perform ransomware detection. Conversely, System API-based approaches did not require any manual definition of the detection criteria. Additionally, the analysis times of Talos are significantly slower than the ones attained by methods proposed in this work (an average of 100 seconds per application, 400 times slower than ours - see Section 7.1).

6.5. Experiment 4: Resilience against Obfuscation

The goal of this experiment was assessing the robustness of System API-based strategies against obfuscated samples, i.e., understanding whether the application of commercial tools to samples could influence the detection capability of the systems. This evaluation 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 [12].

In this experiment, we primarily focused on obfuscated samples whose original (i.e., non-obfuscated) variant was already included in the training set. Such a choice was made because we wanted to assess if obfuscation was enough to influence the key-features of System API-based methods, thus *changing the classifiers' decision for a sample whose original label was malicious.*

To this end, we employed a test-bench of ransomware obfuscated with the tool DexProtector[11], a popular, commercial obfuscation suite that allows for protecting Android applications through heavy code obfuscation. Although such a tool is mostly used for legitimate purposes (e.g., protection of intellectual properties), it can also be used by attackers to make malicious applications harder to be detected. Out of the 3017 ransomware samples, we could obfuscate 2668 samples (the remaining could not be obfuscated due to errors of the obfuscation software) with three different strategies (for a total of 8004 obfuscated samples). The strategies employed to obfuscate samples were the following:

- **String Encryption**. This strategy encrypts strings that are related to `const-string` instructions, and injects a user-implemented method that performs decryption at runtime.
- **Resource Encryption**. It encrypts the external resources contained in the `res` and `assets` folders. To do so, it adds System API information to the `classes.dex` file, in order to properly manage the encryption routines.
- **Class Encryption**. This strategy encrypts user-implemented classes, and injects routines that allow to perform dynamic loading of such classes.

Figure 4 reports the accuracy attained by the three System API-based strategies against the obfuscated samples. Such results show that all the detection strategies (without significant differences between each other) are resilient against obfuscation attempts. However, Class Encryption deserves separate consideration. This strategy employs heavy obfuscation, and it was explicitly performed to defeat static analysis. Typically, none of the static-based techniques that analyze the executable file should be able to detect such attacks correctly. However, this obfuscation strategy introduces a very regular sequence of System API-based routines that manage runtime decryption of the executable contents.

For this reason, it is sufficient to inject only one sample inside the training set to make all obfuscated samples

[10] We directly obtained Talos from the authors, as it is not currently publicly available.

[11] `https://dexprotector.com/`

Table 3: Detection performances for System API-based strategies, `GreatEatlon`, `IntelliAV`, `RevealDroid` and `Talos` on 512 ransomware test files released in 2017, by using training data from 2016. We use the ND (Not Defined) to indicate that a certain tool cannot provide the corresponding label for the analyzed samples.

System	Benign	Generic Malware	Ransomware
Talos	3	0	509
System API (Methods)	7	12	493
System API (Classes)	10	15	487
System API (Packages)	11	32	469
GreatEatlon	118	*ND*	394
RevealDroid	0	512	*ND*
IntelliAV	18	494	*ND*

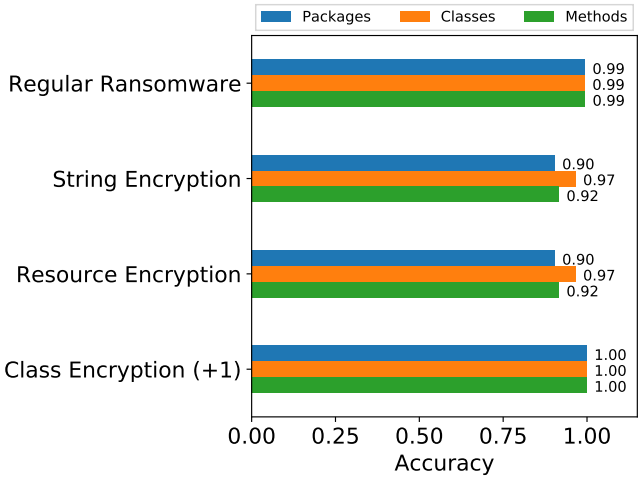

Figure 4: Accuracy performances attained on ransomware samples that have been obfuscated with three different techniques. The accuracy values are reported for the three System API-based detection.

to be detectable. Hence, we added the +1 mark to Class Encryption. Notably, this may create false positives when legitimate samples are obfuscated with the same strategy. Nevertheless, it is sporadic to find such applications, as Class Encryption strongly decreases the application performances [12], and much simpler obfuscation techniques are generally used.

7. Implementation and Performances

Although many solutions have been proposed in the wild to detect ransomware and generic malware, almost none (with the exception, for example, of [5]) was ported to Android devices, often due to the complexity of the proposed approaches. However, an offline, on-device solution is very useful to perform early detection of applications downloaded, for example, from third-party markets (which are more subjected to malware attacks). For this reason, and also to demonstrate the suitability of System API-based approaches, we ported the simplest of the three strategies (Package-based) with the name of `R-PackDroid` (as it implements the same approach introduced in our previous work [10]). This implementation scans for any downloaded, installed and updated applications, and it classifies them as ransomware, malware or legitimate. If an application is found as malicious, the user can immediately remove it.

`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 multiple `.smali` files with `ApkTool`). Therefore, we resorted to `DexLib`, a powerful parsing library part of the `baksmali`[12] disassembler (and used by `ApkTool` itself), to directly extract method calls and their related packages. This library allowed 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`[13], 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.

Figure 5 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 with 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

[12] `https://github.com/JesusFreke/smali`
[13] `https://www.tensorflow.org/`

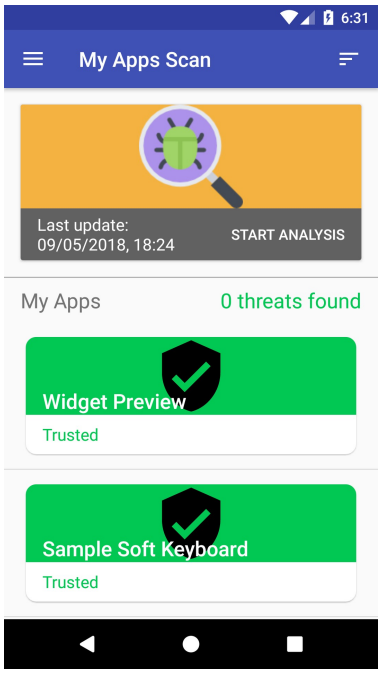

Figure 5: An example of the Android `R-PackDroid` screen.

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 also included). To this scope, we resort to the popular service `FireBase`[14]. `R-PackDroid` is available for free on the Google Play Store (for the moment, Android versions until 7.1 are supported).

7.1. 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)[15]. The choice of benign samples was made because 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, shown in Figure 6, proved that our system could analyze even huge applications in less than 0.2 seconds.

To evaluate the performances of `R-PackDroid` on a real Android phone, we ran the same analysis on a Nexus 5, a 5-years-old, 4-core device with 2 GB of RAM, equipped with the 6.0.1 version of Android. Results are reported

[14] `https://firebase.google.com/`

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

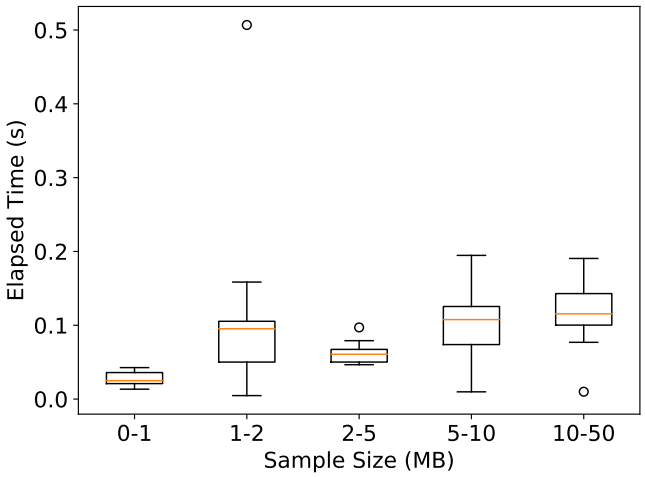

Figure 6: Analysis performances on a X86 workstation, with the elapsed time in seconds, for different `.apk` sizes.

in Figure 7. Even if the analysis times were 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 result was very encouraging, and it 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, was 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.

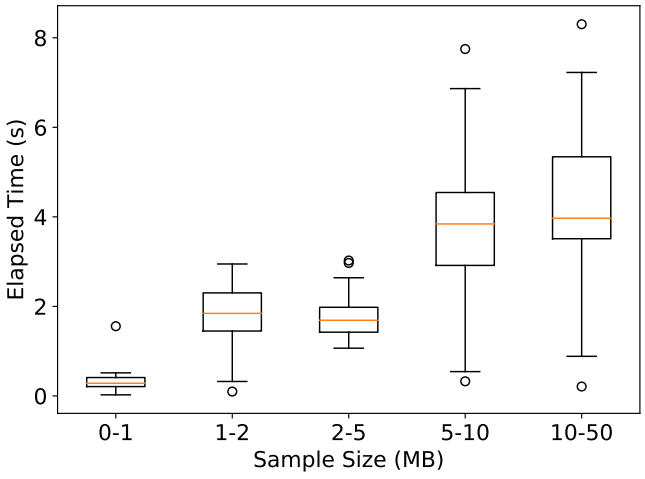

Figure 7: Analysis performances on a real device, with the elapsed time in seconds, for different `.apk` sizes.

8. Discussion and Limitations

The results attained in Sections 6 and 7 can be summarized with the following findings:

- **Finding 1**. System API-based information could be effectively used, *alone*, to properly distinguish ransomware from generic malware and legitimate applications.
- **Finding 2**. Using finer-grained information (classes and methods), albeit involving more features in the analysis, brought significant improvements to accuracy when detecting previously, unseen samples. Moreover, using API-methods allowed for more accuracy under low false positives values.
- **Finding 3**. System API-based approaches could obtain comparable performances to other approaches that involved more features of different types.
- **Finding 4**. System API-based approaches guaranteed robustness against typical obfuscation strategies such as string encryption. However, by including a few obfuscated samples in the training set, it was also possible to detect heavy, anti-static obfuscation techniques such as class encryption.
- **Finding 5**. System API-based approaches were well suitable to be ported and implemented on mobile devices, with excellent computational performances even on very large applications.

We point out that it would be possible to evade the proposed System API-based approaches by replacing System-related packages/classes/methods with semantically equivalent, user-implemented ones. For example, attackers may have two possibilities to replace System API-based methods: *(i)* creating copies of the original instructions of the methods and injecting them into fake methods; *(ii)* re-implementing the methods by using customized instructions/logic. However, these two approaches may feature some critical limitations. In the first approach, the attacker is forced to import the copies of the instructions to the `dex` code (as the methods become user-implemented). However, the imported codes may contain further references to other System API-based methods, which would need to be replaced. Therefore, this procedure may become unfeasible, considering the high variety of calls that can be invoked. The second approach may be hard to implement if the methods to be replaced are very sophisticated (e.g., methods related to cryptography or the execution of activities). We observe that the first approach can be more effective when replacing System API-based packages, as their variety is significantly lower in comparison to methods.

There would also be the possibility that a skilled attacker attempts to evade System API-based detection algorithms by performing Adversarial Machine Learning attacks, such as test-time evasion [36, 2]. In this scenario, the goal would be evading the classifier detection with a minimal number of changes by performing fine-grained modifications to the features of the analyzed test samples. However, this strategy may be challenging to be performed in practice. The problem, also known in the literature as *Inverse Feature Mapping* [36, 37, 38], is constructing the real sample that implements the modifications made to the feature vectors. As the changes to the feature vector would involve the injection or removal of specific System API-based information, they may not be feasible for the reasons we mentioned in the previous paragraph. We plan to inspect the adversarial-related aspects of System API-based methods, as well as the practical creation of evasive samples, in future work.

It is also worth noting that since Android Oreo (8.0), Google introduced new defenses against background processes that are typical of ransomware (e.g., the ones that directly lock the device). However, this does not exclude other malicious actions on the application level. For this reason, it is always better to have an additional system that can detect attempts at performing malicious actions.

Finally, we also point out 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 (regarding the whole corpus that we analyzed) is negligible (less than 1% of the whole file corpus).

9. Conclusions

In this work, we provided a detailed insight into how System API-based information could be effectively used (also on a real device) to detect ransomware and to distinguish it from legitimate samples and generic malware. The attained experimental results demonstrated that, by using a compact set of information tailored to the detection of a specific malware family (ransomware), it was possible to achieve detection performances (also on other malware families) that were comparable to systems that employed a much more complex variety of information. Moreover, System API-based information also proved to be valuable to detect obfuscated samples that focused on hiding user-implemented information. Notably, although it is tempting to combine as many information types as possible to detect attacks (or to develop computationally heavy approaches), it may not be the only, feasible way to construct accurate, reliable malware detectors. For this reason, we claim that future work should focus on developing reliable, small sets of highly discriminant features that cannot easily be manipulated by attackers (with a particular reference to machine learning attacks). Moreover, a clear understanding of the impact of each feature on the classifier detection (also known as *explainability*) can help analysts understand the classifiers errors and to improve their detection capabilities.

Acknowledgements

This work was partially supported by the following projects: *SPARTA* (H2020 EU-funded - GA #830892); *C3ISP* (H2020 EU funded - GA #700294); *INCLOSEC* (funded by Sardegna Ricerche - CUPs G88C17000080006); *PISDAS* (funded by Regione Autonoma della Sardegna - CUP E27H14003150007). The authors also thank Marco Lecis for his valuable contribution to the paper experiments.

References

[1] Symantec, Internet security threat report vol. 23 (2018).

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

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, 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, pp. 23–26.

[4] S. Chen, M. Xue, Z. Tang, L. Xu, 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, ACM, New York, NY, USA, 2016, pp. 377–388. doi:10.1145/2897845.2897860.

[5] M. Ahmadi, A. Sotgiu, G. Giacinto, Intelliav: Toward the feasibility of building intelligent anti-malware on android devices, in: A. Holzinger, P. Kieseberg, A. M. Tjoa, E. Weippl (Eds.), Machine Learning and Knowledge Extraction, Springer International Publishing, Cham, 2017, pp. 137–154.

[6] G. Canfora, F. Mercaldo, G. Moriano, C. A. Visaggio, Composition-malware: building android malware at run time, in: 2015 10th International Conference on Availability, Reliability and Security, IEEE, 2015, pp. 318–326.

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

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

[9] J. Garcia, M. Hammad, S. Malek, Lightweight, obfuscation-resilient detection and family identification of android malware, ACM Trans. Softw. Eng. Methodol. 26 (3) (2018) 11:1–11:29. doi:10.1145/3162625.

[10] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, F. Martinelli, R-packdroid: Api package-based characterization and detection of mobile ransomware, in: Proceedings of the Symposium on Applied Computing, SAC '17, ACM, New York, NY, USA, 2017, pp. 1718–1723. doi:10.1145/3019612.3019793.

[11] J. Chen, C. Wang, Z. Zhao, K. Chen, R. Du, G.-J. Ahn, Uncovering the face of android ransomware: Characterization and real-time detection, IEEE Trans. on Information Forensics and Security (TIFS) 13 (5) (2018) 1286–1300. doi:10.1109/TIFS.2017.2787905.

[12] D. Maiorca, D. Ariu, I. Corona, M. Aresu, G. Giacinto, Stealth attacks: An extended insight into the obfuscation effects on android malware, Computers & Security 51 (C) (2015) 16–31. doi:10.1016/j.cose.2015.02.007.

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

[14] A. Gharib, A. Ghorbani, Dna-droid: A real-time android ransomware detection framework, in: Z. Yan, R. Molva, W. Mazurczyk, R. Kantola (Eds.), Network and System Security: 11th International Conference, NSS 2017, Helsinki, Finland, August 21–23, 2017, Proceedings, Springer International Publishing, Cham, 2017, pp. 184–198. doi:10.1007/978-3-319-64701-2_14.

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

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

[17] A. Santone, G. Vaglini, Abstract reduction in directed model checking ccs processes, Acta Informatica 49 (5) (2012) 313–341. doi:10.1007/s00236-012-0161-3.

[18] M. Ceccarelli, L. Cerulo, A. Santone, De novo reconstruction of gene regulatory networks from time series data, an approach based on formal methods, Methods 69 (3) (2014) 298–305. doi:10.1016/j.ymeth.2014.06.005.

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

[20] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, E. Bodden, Mining apps for abnormal usage of sensitive data, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 426–436.

[21] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, W. Enck, Appcontext: Differentiating malicious and benign mobile app behaviors using context, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 303–313.

[22] F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, Car hacking identification through fuzzy logic algorithms, 2017. doi:10.1109/FUZZ-IEEE.2017.8015464.

[23] F. Martinelli, F. Mercaldo, A. Orlando, V. Nardone, A. Santone, A. K. Sangaiah, Human behavior characterization for driving style recognition in vehicle system, Computers & Electrical Engineering.

[24] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, G. Giacinto, Clustering android malware families by http traffic, in: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), 2015, pp. 128–135.

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

[26] H. Zhang, D. D. Yao, N. Ramakrishnan, Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery, in: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, ACM, New York, NY, USA, 2014, pp. 39–50. doi:10.1145/2590296.2590309.

[27] H. Zhang, D. D. Yao, N. Ramakrishnan, Causality-based sensemaking of network traffic for android application security, in: Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security, AISec '16, ACM, New York, NY, USA, 2016, pp. 47–58. doi:10.1145/2996758.2996760.

[28] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, F. Maggi, Shieldfs: A self-healing, ransomware-aware filesystem, in: Proceedings of the 32Nd Annual Conference on Computer Security Applications, ACSAC '16, ACM, New York, NY, USA, 2016, pp. 336–347. doi:10.1145/2991079.2991110.

[29] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, E. Kirda, UNVEIL: A large-scale, automated approach to detecting ransomware, in: 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX, 2016, pp. 757–772.

[30] E. Kolodenker, W. Koch, G. Stringhini, M. Egele, Paybreak:

Defense against cryptographic ransomware, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '17, ACM, New York, NY, USA, 2017, pp. 599–611. doi:10.1145/3052973.3053035.

[31] J. Huang, J. Xu, X. Xing, P. Liu, M. K. Qureshi, Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, 2017, pp. 2231–2244. doi:10.1145/3133956.3134035.

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

[33] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: IEEE Symposium on Security and Privacy, IEEE, 2012, pp. 95–109.

[34] K. Allix, T. F. Bissyandé, J. Klein, 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, ACM, New York, NY, USA, 2016, pp. 468–471. doi:10.1145/2901739.2903508.

[35] M. Melis, D. Maiorca, B. Biggio, G. Giacinto, F. Roli, Explaining black-box android malware detection, in: 26th European Signal Processing Conference, EUSIPCO 2018, Roma, Italy, September 3-7, 2018, 2018, pp. 524–528. doi:10.23919/EUSIPCO.2018.8553598.

[36] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, F. Roli, Evasion attacks against machine learning at test time, in: H. Blockeel, K. Kersting, S. Nijssen, F. Železný (Eds.), European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part III, Vol. 8190 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 387–402.

[37] B. Biggio, G. Fumera, F. Roli, Pattern recognition systems under attack: Design issues and research challenges, IJPRAI 28 (7).

[38] B. Biggio, F. Roli, Wild patterns: Ten years after the rise of adversarial machine learning, Pattern Recognition 84 (2018) 317 – 331. doi:10.1016/j.patcog.2018.07.023.