# Extinguishing Ransomware - a Hybrid Approach to Android Ransomware Detection

Alberto Ferrante[1], Miroslaw Malek[1],
Fabio Martinelli[2], Francesco Mercaldo[2], and Jelena Milosevic[1]

[1] Advanced Learning and Research Institute, Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
[2] Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy

**Abstract.** Mobile ransomware is on the rise and effective defense from it is of utmost importance to guarantee security of mobile users' data. Current solutions provided by antimalware vendors are signature-based and thus ineffective in removing ransomware and restoring the infected devices and files. Also, current state-of-the art literature offers very few solutions to effectively detecting and blocking mobile ransomware. Starting from these considerations, we propose a hybrid method able to effectively counter ransomware, that first investigates applications to be used on a device prior to their installation (static approach) and then observes their behavior at runtime and identifies if the system is under attack (dynamic approach). To detect ransomware, the static detection method uses the frequency of opcodes while the dynamic detection method considers CPU usage, memory usage, network usage and system call statistics. We evaluate the performance of our hybrid detection method on a dataset that contains both ransomware and legitimate applications. Additionally, we evaluate the performance of the static and of the dynamic stand-alone methods for comparison. Our results show that although both static and dynamic detection methods perform well in detecting ransomware, their combination in a form of a hybrid method performs best, being able to detect ransomware with 100% precision and having a false positive rate of less than 4%.

**Keywords:** ransomware, malware, hybrid detection, machine learning, Android, security

## 1 Introduction

According to [8], 2016 will be remembered as "the year of ransomware", confirming the predictions of exponential growth of these kinds of attacks from previous years [7]. Recent events, such as the WannaCry ransomware attack of May 2017, in which over 200,000 computers in more than 150 countries were rendered unusable with ransom demands [3], demonstrated that these predictions might be

[3] `http://wapo.st/2pKyXum?tid=ss_tw&utm_term=.6887a06778fa`

beaten in 2017. Ransomware is such a relevant security problem that law enforcement agencies from all around Europe teamed up with antimalware and other IT security companies to form the *No More Ransom!* project [4]. Ransomware not only targets PCs and servers but also mobile devices. In particular, in the mobile device realm, as opposed to other threats that silently try to get into possession of data by copying them and leaking through network, ransomware denies access to data by encryption or simply by locking the device and scaring users (i.e., users are made believe that data are encrypted even if they are not). In some cases even paying the requested ransom does not guarantee that the access to data is restored. Having in mind that data are one of users' most valuable assets, a mechanism that can detect ransomware at installation time of applications or during their execution is highly desirable. However, although mobile ransomware threats are on the rise, we have found only a small number of related works in literature addressing the problem of mobile ransomware detection.

We propose to use a hybrid detection method that is composed of a static method, to be used when applications are installed and/or updated, and a dynamic method, to be used at runtime. Static methods detect ransomware by considering features that can be obtained without running the applications (e.g., frequency of op-codes). Dynamic methods, instead, are based on features that can only be obtained at runtime and that represent the behavior of applications (memory, CPU, network and statistics on system calls). Static approaches are less computationally intense than dynamic methods and that they do not need applications to be run for identifying malware [10], but they are typically ineffective with obfuscated code as well as with run-time infections. On the other hand, dynamic methods are effective in identifying new threats, outperforming static methods, but they need applications to be run to identify malicious behaviour, potentially infecting the device [9]. In addition dynamic methods are able to discriminate malware even when its code is obfuscated [15]. The main idea behind using a hybrid approach is to have the advantages of both static and dynamic methods while reducing their disadvantages.

Having this in mind, the main contributions of this paper are as follows:

1. Evaluation of the effectiveness of a static approach, based on the frequency of op-codes, in detection of mobile ransomware (Section 4.1).
2. Evaluation of the effectiveness of a dynamic approach, based on the monitoring of memory, CPU, network and statistics on system calls, in detection of mobile ransomware (Section 4.2).
3. Evaluation of the effectiveness of a hybrid, combined static and dynamic, approach in detection of mobile ransomware (Section 4.3).

## 2 Hybrid Detection Method

Hybrid detection employs both static and dynamic detection. Figure 1 depicts the high-level workflow of the proposed approach: we first use a static detection

[4] `https://www.nomoreransom.org`

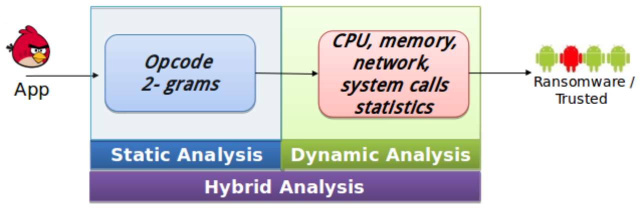

**Fig. 1.** High-level workflow of the proposed approach. The static analysis consists of a classification based on features obtained from the executable of the application under analysis, while the dynamic one is based on a feature set obtained when the application is running.

method when applications are installed, updated, or during periodic checks (e.g., every week). Applications identified as malware are denied running on the device. All the other applications are instead allowed to run, but subject to dynamic detection while they are executed. In this way we are able to complement the coverage of static detection with the one of dynamic detection.

Following, we discuss the two methods, static and dynamic, that compose our hybrid method. Additionally, we discuss the steps required to develop and use them – pre-processing, learning, and classification – as well as the system features, both static and dynamic, that are considered.

## 2.1 Static Analysis

Structural code analysis has been identified by the research community as effective and highly accurate in static detection methods. For instance, in [4, 3], the opcode occurrences are considered as main features for detecting malware, with an obtained precision of 0.9. Additionally, the method proposed in [2] demonstrated that the sequences of opcodes are very effective in detecting Android malware, providing an accuracy of 96.88%. In this approach, the authors considered a binary classification problem in which an input application $a$ has to be classified as malware or trusted using the occurrences of two opcodes as features (i.e., 2-grams). $n$-grams with $n = 2$ demonstrated to provide the best possible performance in identifying Android malware with respect to other values of $n$ [4, 3, 12].

Other approaches, such as the ones based on features derived from the permissions required by the applications and/or system call occurrences provide good detection performance, but not as good as the previously described methods.

Thus, we adopt an approach similar to [2] for detecting ransomware. In our approach each application is pre-processed in order to obtain the numeric values of frequencies of opcode sequences that are suitable to be processed by the classifier. After pre-processing, the classifier undergoes the learning phase in which it is trained by using a labelled dataset. After the learning phase, the classifier can be used for the actual classification of the applications as ransomware or trusted.

Let $a$ be the Android application under analysis. In the pre-processing phase, we first use the apktool [5] tool in order to extract the Smali classes of the application under analysis, thus being able to obtain a unique file containing the full set of opcodes (without the relative argument and parameters) of all the $a$ application classes.

We then compute the frequency of 2-grams opcodes as follows: let $O$ be the set of possible opcodes, and let $\mathcal{O} = \bigcup_{i=1}^{i=n} O^i$ the set of $n$-grams (i.e., sequences of opcodes whose length is up to $n$, we consider $n = 2$). We denote with $f(a, o)$ the frequency of the $n$-gram $o \in \mathcal{O}$ in the application $a$: $f(a, o)$ is hence the number of occurrences of $o$ divided by the total length of the opcode sequences in $a$. Finally, we set the *feature vector* $\boldsymbol{f}(a) \in [0, 1]^{|\mathcal{O}|}$ corresponding to $a$ to $\boldsymbol{f}(a) = (f(a, o_1), f(a, o_2), \dots)$ with $o_i \in \mathcal{O}$. Additionally, we split the application code into chunks corresponding to class methods.

In the next phase, called learning, we train a binary classifier $C$ from two sets $A_M$, $A_T$ of ransom and trusted applications (the *learning sets*), respectively. The learning phase is divided into a feature selection phase and the actual classifier training phase. The aim of the feature selection phase is twofold: on the one hand, we want to reduce the dimension of the input since with $n = 2$ the size $|\mathcal{O}|$ of each feature vector $\boldsymbol{f}$ can be up to $\approx 10^{12}$. On the other hand, we want to retain only the more informative $n$-grams, with respect to the output label.

At first, the average frequencies $\bar{f}_M(o)$ and $\bar{f}_T(o)$ are computed for each 2-gram $o \in \mathcal{O}$ on the ransomware and trusted samples.Then, the relative difference $d(o)$ between the two average values is computed. This relative difference is high if the 2-gram $o$ is frequent among ransomware applications and infrequent among trusted applications (and vice versa). Then, we build the set $\mathcal{O}' \subset \mathcal{O}$ of $n$-grams composed of the $h$ $n$-grams with the highest values of $d(o)$, where $h$ is a parameter of our method. We do not include in $\mathcal{O}'$ the $n$-grams for which $d(o) = 1$ (i.e., we purposely do not consider those 2-grams which occur in only one subset of applications in the learning set): this way, we strive to avoid building a classifier which works well on seen applications but fails to generalize. Finally, we retain in $\mathcal{O}'$ only the remaining 2-grams with the greatest value. Accordingly, we set the *reduced feature vector* $\boldsymbol{f}'(a)$ corresponding to $a$ using only the frequencies of the 2-grams in $\mathcal{O}'$, i.e., $\boldsymbol{f}'(a) = (f(a, o_1), f(a, o_2), \dots)$ with $o_i \in \mathcal{O}'$. We consider as *reduced feature vector* the 50 2-grams with the greatest value, i.e. the most frequent 2-grams. The second step of the learning phase consists of training the actual classifier $C$ using the reduced feature vectors obtained from the applications in the learning sets and the corresponding labels. In this work, we experimented with the J48, NaiveBayes, and Logistic Regression classification algorithms.

In the last phase, named classification, we determine if an application $a$ is labelled as ransomware or trusted, according to the learned classifier $C$. To this end, we pre-process $a$ as previously explained in order to obtain the

[5] `https://ibotpeaches.github.io/Apktool`

reduced feature vector $\boldsymbol{f}'(a)$. Then, we input $\boldsymbol{f}'(a)$ to $C$ and classify $a$ into $\{\text{ransomware}, \text{trusted}\}$. In a real system, this step is run every time a new application is installed or updated. Additionally, it can be run periodically to check all the applications installed on the mobile device.

## 2.2 Dynamic Analysis

In order to perform dynamic detection of ransomware, an effective method based on the observation of system behavior has to be used; since this method needs to be used at runtime on mobile devices, it also needs to be lightweight on computational and energy resources. For this purpose, as the most suitable candidate we identified MalAware, the approach proposed in [13], that uses only seven memory and CPU related features in order to perform on-device detection at runtime, and that is based on a two-steps detection system of low complexity that first classifies execution records, and then complete applications by relying on the past classifications of execution record. Thus, we use a similar two-steps detection approach, but in addition to memory and CPU usage we also consider features representing network usage and statistics on system calls. Similarly to static detection, the development of the dynamic detection method undergoes the two phases of pre-processing and learning, with the classification phase used at runtime to actually detect malware.

The pre-processing phase is necessary to extract features from the execution logs of the considered applications. As explained in Section 3, execution logs are obtained by running the considered applications in an instrumented environment.

In the learning phase, we first train a classifier to recognize execution records associated with malicious behavior, similarly to the work presented in [14]. The training of classifiers is performed by using the same training set used for the development of static detection as well as the same classifiers of low complexity, suitable for on device, runtime detection: Naive Bayes, Decision Trees (J48), and Logistic Regression. The results of classification are then used as input for a sliding-window based mechanism that considers the history of past execution records to classify the applications. Namely, considering a sliding window of length $n$, the percentage of records classified as ransomware in the last $n$ instants of time is used to determine whether an application is ransomware or not. To make the mechanism more robust, multiple results, obtained in disjointed sliding windows, are considered: when $w$ windows are marked as ransomware, the application is classified as ransomware. In the training phase we also choose the most suitable parameters for the sliding window mechanism, namely window size, threshold, and number of checks. The choice is done by exploring different combinations of parameters and y studying the obtained results as discussed in Section 4.

In the classification phase, the application classifier obtained in the training phase is used at runtime to detect ransomware. The full detection system is composed of three main parts: a feature monitoring block, the record-level classifier, and the applications-level classifier. The first block monitors and extracts features periodically and sends them to the record-level classifier which, in turn, sends its classification results to the application-level classifier. This last classifier is the one that can mark an application as ransomware and raise an alarm.

## 3 Experimental Setup

In this section, we first describe our dataset, followed by the setup of the experimental environment and the description of collected features.

### 3.1 Dataset

We based our experiments on a dataset containing 3,058 mobile applications: 2,386 Android trusted applications downloaded from the Google Play Store [6] and 672 applications containing ransomware taken from the freely available HelDroid dataset [7]. These ransomware samples appeared from December 2014 to June 2015. Following, we list the malware families to which these samples belong: [1]:

- Ransomware applications in the *Locker* family block the screen of infected devices and request a ransom for unlocking it; no file is actually encrypted.
- The *Koler* payload is downloaded by exploiting site redirection; the screen is then occupied by the ransom browser page that cannot be dismissed if not for very short periods of time.
- Ransomware in the *Svpeng* family is based on an overlay attack: legitimate applications launched by the user are overlayed with fake windows imitating the legitimate applications and thus fooling the victim. Additionally, users receive a message, pretending to be sent by FBI and claiming that the device has been locked due to access to child pornography websites. To unlock the phone, a ransom need to be paid.
- Samples in the *ScarePakage* ransomware family masquerade as well-known applications, such as Adobe Flash or antimalware applications, and, when launched, they pretend to scan your phone. After completing the fake scan, the device is locked and after a reboot a fake FBI message is shown. A ransom is asked to bring back the device to normal.
- Ransomware applications in the *SimpleLocker* family scan the SD card for images, documents and videos and encrypt them by using the AES encryption algorithm; a message notifying the user and asking for a ransom is shown on the display. This is the only family in our dataset that actually encrypts data on the device and it was the first one discovered for Android.

In order to download trusted applications we crawled the Google Play Store by using the *Python Android Market Library* open-source crawler [8]. The crawler

[6] `https://play.google.com/store`
[7] `http://ransom.mobi`
[8] `https://github.com/liato/android-market-api-py`

is configured to download trusted applications equally distributed in all the different categories of the market.

All the applications were checked by means of VirusTotal [9], a service that runs 57 different antimalware on the submitted applications: the analysis confirmed that our trusted samples did not contain any known malicious payload, while the malicious samples contain ransomware specific payloads.

For developing and validating the proposed approach, we have separated the collected applications into a training and a test dataset. The training dataset contains two thirds of the available trusted applications and two thirds of the available application for each of the ransomware family; the testing dataset contains the remaining applications.

## 3.2 Experimental Environment for Dynamic Analysis

Dynamic analysis relies on runtime observation of memory usage, CPU usage, network behavior, as well as statistics on system calls. Therefore, execution traces containing this information need to be collected by executing the applications in a controlled environment. These traces have been recorded by running the applications, one at a time, on the Android emulator. Monitoring scripts, with a monitoring interval of two seconds have been used. The procedure of executing the applications was automated by means of a Linux shell script, which has been run on a Linux PC and made use of Android Debug Bridge (adb) [10], a command line tool that allows the PC to communicate with an emulator instance or with an Android device. To collect system calls we used strace [11], a tool for tracing system calls. Network log files were collected by capturing the network traffic of the emulator. Network statistics have been obtained by logging all network traffic of the emulator and by successively running the *tcpstat* tool, set to consider 2s intervals. Log files for CPU, memory, and network are later unified by using timestamps recorded at execution time.

For applying stimuli to applications, the Monkey application exerciser [12] has been used in the script. It is a command-line tool that sends a pseudo-random stream of user events into the system, which acts as a stress test on the application software. One of the main problems of dynamic detection methods is in the execution of samples during the development phase. In fact, there is currently no method to verify automatically that malicious payloads are activated correctly. Due to the high number of samples that need to be used to obtain good quality classifiers, it is not even possible to perform this verification manually. This introduces some additional uncertainty in dynamic detection methods. In this work, we have tried to minimize the number of non-activated malicious payload by providing a high number of stimuli ($20,000$, with a limit on execution time of 15 minutes) to each application (both malware and benign). It is our belief that

[9] `https://www.virustotal.com/`

[10] http://developer.android.com/tools/help/adb.html

[11] `http://linux.die.net/man/1/strace`

[12] http://developer.android.com/tools/help/monkey.html

the duration that we have chosen is a good tradeoff between time when most of the ransomware samples expose their malicious intentions and duration of the overall experimentation. The Android emulator of choice is the one included in the Android Software Development Kit [13] release 20140702, running Android 4.0, that was one of the most popular versions of Android in period from when ransomware samples originate. The reason why an Android emulator has been chosen instead of real devices is that this solution provides the ability to run a large number of applications, making the obtained dataset more significant. The Android operating system has been re-initialized each time before running each application, to avoid possible interferences (e.g., changed settings, running processes, and modifications of the operating system files) from previously run samples.

In total, 87 features could be extracted from the execution traces, all referred to single applications resource usage. All the features considered are listed in Table 1. Out of the considered features, 59 are related to different aspects of memory usage; five are related to CPU: three to CPU usage, and two to virtual memory exceptions (major and minor faults); 15 are related to network traffic; five represent statistics on system calls.

## 4 Experimental Results

In this section, we report the results obtained when considering static detection alone, dynamic detection alone, and the hybrid method. In order to evaluate the detection performance of the developed systems, we report four metrics: precision, recall, F-Measure, and Receiver Operating Characteristics (ROC) curve. F-Measure represents a weighted average of precision and recall, while the ROC Area is defined as the probability that a randomly chosen positive instance is incorrectly classified as a negative instance; ROC Area is a suggested metric to represent detection performance when the considered datasets are unbalanced, as it is in our case, and in malware detection in general [5].

### 4.1 Static Detection

We first discuss the results obtained by using the static detection method described in Section 2.1. As mentioned in Section 2.1 we have experimented with three different classifiers, namely J48, Naive Bayes, and Logistic Regression as well as with $n$ equal to 2. In feature selection, we use $h = 50$; in fact, we have determined experimentally that when a greater value is used, performance decays, providing non-meaningful $n$-grams. After the learning phase, performed by using the training set described in Section 3, we applied the obtained classifier, $C$, to each application of the test set and we measured precision, recall, F-Measure and ROC Area.

The obtained results are shown in Table 2. With the three different classification algorithms considered in the study, we have obtained a precision ranging

[13] https://developer.android.com/sdk/index.htm

**Table 1.** List of all the considered features; totals are related only to single applications; unless differently specified, all numbers are related to the considered monitoring period.

| Category | | Feature Names |
|---|---|---|
| CPU | CPU Usage | Total CPU Usage, User CPU Usage, Kernel CPU Usage |
| | Virtual Memory | Page Minor Faults, Page Major Faults |
| Memory | Native memory | Native Pss, Native Shared Dirty, Native Private Dirty, Native Heap Size, Native Heap Alloc, Native Heap Free |
| | Dalvik memory | Dalvik Pss, Dalvik Shared Dirty, Dalvik Private Dirty, Dalvik Heap Size, Dalvik Heap Alloc, Dalvik Heap Free, Cursor Pss |
| | Cursor memory | Cursor Shared Dirty, Cursor Private Dirty |
| | Android shared memory | Ashmem Pss, Ashmem Shared Dirty, Ashmem Private Dirty |
| | Memory-mapped native code | .so mmap Pss, .so mmap Shared Dirty, .so mmap Private Dirty |
| | Memory mapped Dalvik code | .dex mmap Pss, .dex mmap Shared Dirty, .dex mmap Private Dirty |
| | Memory-mapped fonts | .ttf mmap Pss, .ttf mmap Shared Dirty, .ttf mmap Private Dirty |
| | Other memory-mapped files and devices | .jar mmap Pss, .jar mmap Shared Dirty, .jar mmap Private Dirty, .apk mmap Pss, .apk mmap Shared Dirty, .apk mmap Private Dirty, Other mmap Pss, Other mmap Shared Dirty, Other mmap Private Dirty |
| | Non-classified memory allocations | Unknown Pss, Unknown Shared Dirty, Unknown Private Dirty, Other dev Pss, Other dev Shared Dirty, Other dev Private Dirty |
| | Memory Totals | TOTAL Pss, TOTAL Shared Dirty, TOTAL Private Dirty, TOTAL Heap Size, TOTAL Heap Alloc, TOTAL Heap Free |
| | Objects | Views, ViewRootImpl, AppContexts, Activities, Assets, AssetManagers, Local Binders, Proxy Binders, Death Recipients, OpenSSL Sockets |
| | SQL | heap, MEMORY_USED, PAGECACHE_OVERFLOW, MALLOC_SIZE |
| Network | Link layer networking | Number of ARP packets, AVG. PKT Size bytes, bps, Number of ICMP packets, Size in byte standard deviation |
| | Internet layer networking | Number of IPv4 packets, Network load over last minute, Maximum packet size in bytes, Minimum packet size in bytes, Number of bytes, Number of packets, Number of packets per second, Number of IPv6 packets |
| | Transport layer networking | Number of TCP packets, Number of UDP packets |
| System calls | | Number of Syscalls, No. of different syscalls, Average no. of calls per syscall, No. of calls occurring once, No. of calls occurring multiple times |

from 0.968 to 0.998 and a recall ranging between 0.988 and 0.997. The obtained F-measure is between 0.980 and 0.998 and the ROC area is ranging from 0.998 to 1.000. The classifier with best performance is J48.

The static method classifies correctly all the benign applications (i.e., no benign applications are marked as ransomware), but nine ransomware applications are misclassified as benign. In other words, the static method has no false positive, but it has nine false negatives. False negatives are represented by three malware samples from the *Simple Locker* family and six samples from *Koler* family; all the samples from the other familes are classified correctly as malware.

**Table 2.** Classification results for ransomware and benign applications when features extracted by static analysis are considered along with the J48, NB (Naive Bayes) and LP (Logistic Regression) classifiers.

| Algorithm | Precision | | Recall | | F-Measure | | RocArea | |
|---|---|---|---|---|---|---|---|---|
| | Ransom | Trusted | Ransom | Trusted | Ransom | Trusted | Ransom | Trusted |
| J48 | 0.998 | 1.000 | 0.997 | 1.000 | 0.998 | 1.000 | 0.998 | 0.998 |
| NB | 0.968 | 1.000 | 0.992 | 0.998 | 0.980 | 0.999 | 0.999 | 0.999 |
| LR | 0.994 | 0.999 | 0.988 | 1.000 | 0.991 | 0.999 | 1.000 | 1.000 |

**Table 3.** Classification results for ransomware and benign applications when features extracted by dynamic analysis are considered along with the J48, NB (Naive Bayes) and LP (Logistic Regression) classifiers.

| Algorithm | Precision | | Recall | | F-Measure | | RocArea | |
|---|---|---|---|---|---|---|---|---|
| | Ransom | Trusted | Ransom | Trusted | Ransom | Trusted | Ransom | Trusted |
| J48 | 0.988 | 0.994 | 0.976 | 0.997 | 0.982 | 0.995 | 0.998 | 0.998 |
| NB | 0.382 | 0.975 | 0.940 | 0.605 | 0.543 | 0.747 | 0.922 | 0.943 |
| LR | 0.933 | 0.973 | 0.894 | 0.983 | 0.913 | 0.978 | 0.986 | 0.986 |

## 4.2 Dynamic Detection

As described in Section 2.2, the applied dynamic detection method identifies ransomware applications as such by first detecting potentially malicious execution records, and then, based on the classification of records, by classifying complete applications. In Table 3 we enclose the classification results obtained by using the classification algorithms for the detection of malicious records. While Naive Bayes provides lower detection accuracy, both Logistic Regression and J48 perform well as shown by all considered metrics. Both of these methods are of low complexity, and, thus, suitable for on-device detection. Due to its ability to assign a probability of being malicious to a record, instead of just providing a binary decision, we opted for the Logistic Regression classifier.

The output of record-level classification is, in all effects, a labelling of the execution records in the execution traces as malicious or not. This piece of information is used by the sliding windows mechanism to classify the applications as ransomware or not. In order to find the most suitable parameters of the sliding window mechanism for our scenario, we have explored different combinations of parameters, depicted in Table 4, by running a batch of experiments on the training data set. These sets of parameters were determined by running some preliminary experiments. Based on these obtained results, we have selected the configurations that provide highest F-Measure, highest detection accuracy with false positive below 20%, lowest false positive with accuracy higher than 80%, and lowest detection time. These configurations are considered having in mind different possible application scenarios (those in which detection accuracy would have the highest priority, those where the lowest false positives would have the highest priority, and those in which balance of both of them would be preferred). The best configurations according to the aforementioned metrics are shown, along with the corresponding detection performance, in Table 5. In these results, detection time is measured from the first execution record marked as

**Table 4.** Algorithm parameters used in the exploration phase.

| Parameter | Values |
|---|---|
| Sliding window length | 1, 3, 5, 7, 9, 10, 11, 12, 13, 15 |
| Threshold (%) | 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80 |
| Checks (no.) | 1, 2, 3, 4, 5 |

**Table 5.** Best results obtained with respect to the observed metrics. Detection time is measured from the first record identified as malicious.

| Configuration | Window length | Threshold (%) | Checks | Detection rate (%) | False positive rate (%) | F-Measure | Detection time (s) |
|---|---|---|---|---|---|---|---|
| Highest F-Measure | 9 | 78 | 1 | 90.82 | 3.46 | 0.85 | 20.46 |
| Highest detection rate / lowest detection time | 1 | 60 | 1 | 96.33 | 19.84 | 0.58 | 0 |
| Lowest false positive | 10 | 70 | 2 | 80.27 | 2.92 | 0.80 | 60.84 |

malicious. This is done to represent the fact that the malicious behavior does not always start at the beginning of the execution of the applications containing ransomware, even though, according to our results, for most applications the first malicious record is identified within the first few seconds of execution. The best trade-off between these requirements is represented by highest F-Measure that in our case is 0.85; this configuration provides high accuracy, with a low number of false positive and a short detection time.

After selecting these optimal parameters on the training set, we tested them on the test set; the results obtained are shown in Table 6. As expected, detection performance decreases with respect to the training set, but it remains high. For example, when the parameters for highest F-measure are selected, the detection rate decreases from 90% to 85%. The case of the parameters chosen for lowest false positive rate provides best performance on the test set, with an increase of the detection rate and a slight increase of the false positive rate.

Considering the fact that execution traces in the test set correspond to ransomware samples that were not used during training (even though other samples belonging to the same families were) and, therefore, they are unknown to the classifiers, we find the obtained detection performance very promising. In fact, in real life, a mix of both known and unknown malware samples would be analyzed by the ransomware detection mechanism and this should, in our opinion, further increase the obtained detection performance.

When best F-measure parameters are considered, 62 ransomware samples go undetected. By relying on Virustotal, we have classified these samples in categories. We have taken as a reference F-secure that provides descriptive ransomware definitions. Most of the ransomware samples that are not identified by our dynamic malware detection method belong to the Simplelocker (61%) and to the Koler (19%) families. The remaining samples were not identified as ransomware by F-secure, even though other antiviruses identified them as such. Similarly to static detection, and although providing promising results, dynamic detection alone cannot detect all the observed ransomware samples.

**Table 6.** Results obtained in the testing phase with the best parameters obtained in the exploration phase. Detection time is measured from the first record identified as malicious.

| Configuration | Window length | Threshold (%) | Checks | Detection rate (%) | False positive rate (%) | F-Measure | Detection time (s) |
|---|---|---|---|---|---|---|---|
| Highest F-Measure | 9 | 78 | 1 | 85.61 | 5.31 | 0.88 | 24.24 |
| Highest detection rate / lowest detection time | 1 | 60 | 1 | 93.03 | 25.06 | 0.79 | 0 |
| Lowest false positive | 10 | 70 | 2 | 84.68 | 3.81 | 0.89 | 44.72 |

## 4.3 Hybrid Detection

To evaluate the effectiveness of the hybrid approach, we have considered the list of ransomware applications in the test set that are not detected by our static method, and we have checked whether dynamic detection could correctly detect them as ransomware. All the nine applications that are not detected by our static method are correctly identified as ransomware by our dynamic method, with all the three sets of optimal parameters. Therefore, we have verified our initial assumption that by using a hybrid method we could increase the coverage of ransomware detection. In fact, starting from a detection rate of 99.8% for static detection and of 85.61% (with best f-measure parameters) for dynamic detection, we obtain a 100% detection rate for hybrid detection. While the static method has no false positives, the dynamic method has some, as shown in Table 6. Considering the extremely good detection performance, we can choose to optimize the dynamic method for detection speed or for the lowest number of false positives, choosing the corresponding parameters of Table 5. In summary, the results show that using the hybrid method is the way to go in order to provide effective protection against ransomware not only for its increased coverage, but also to unite the accuracy of static detection with the possibility of detecting ransomware at runtime of dynamic detection.

Finally, while our method provides 100% detection of ransomware, we need to point out that static and dynamic detection are different in what they offer. In fact, static detection is able to detect ransomware before the application containing it is executed, thereby preventing the malicious payload from executing and, thus, preventing any harm to the system. Dynamic detection, instead, detects malware while applications are being executed and, thus, when some harm to the system might have been already caused. As previously discussed, our detection method is relatively fast in detecting malicious behavior, but still it leaves open the possibility of harming the system. On the other hand, dynamic detection is able to capture ransomware at runtime, thus offering protection against dynamically installed code, which cannot be detected by static methods since they are run with much lower periodicity, and against malicious actions hidden in obfuscated code.

## 5 Related Work

The main difference between ransomware and other widespread mobile malware types is in their behaviour: as demonstrated in [18], Android malware generally focuses on remaining hidden while gathering and sending to the attackers user sensitive and private information. Ransomware, instead, leverages the knowledge of its presence by users to obtain the payment of a ransom. Due to this, current methods proposed by the research community for the identification of malware are not necessarily effective in detecting ransomware, as the recent rise in ransomware attacks demonstrates. As also stated in [1], this ineffectiveness of traditional methods, when used alone, exposes more than a billion users to this threat. Effective solutions for detection of ransomware on mobile devices are needed, but up to date, there are only a few works addressing this issue in the literature. Here, we first discuss them in detail and then, we compare current literature with the approach that we propose.

The first method that introduced ransomware detection for Android is HelDroid [1]. This tool includes a text classifier based on NLP features, a lightweight smali emulation technique to detect locking strategies, and the application for detecting file-encrypting flows. The main weakness of HelDroid is that it strongly depends on a text classifier: as a matter of fact, 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 techniques such as string encryption and data ciphering [15]. Furthermore, the proposed method strongly depends on language dictionaries; this is the reason why, as stated by the authors, when the analyzed ransomware is targeting non-english speakers, the dictionary must be switched to a different language. In addition, this method can be evaded by altering the occurrences of such words. From the performance point of view, they identify rightly 375 Android ransomware on a dataset composed of 443 samples: 11 ransomware were not detected due to unsupported language (e.g., Spanish, Russian) with 9 out of 12,842 false positives.

In [17], the authors propose a performance tool in order to help to understand what can be done to cope with Android ransomware detection. This tool provides the ability to dump the log of system messages, including stack traces. However, this method remained at the level of a proposal, with no implementation. Therefore, there are no results that can prove its effectiveness.

Song et al. [16] designed an approach with the aim of identifying mobile ransomware by using process monitoring. They consider features representing the I/O rate as well as the CPU and memory usage. They evaluate the proposed method with only one ransomware sample developed by the authors. This sample has the ability to encrypt the file by using AES.

An approach based on formal methods that is able to detect Android ransomware and to identify the malicious sections in the application code is described in [11]. The authors evaluate a dataset composed of 2,477 samples with real-world ransomware and trusted applications. Starting from the payload behaviour definition, the authors formulate logic rules that are later applied to detect ransomware. The main weakness of the proposed method is represented

**Table 7.** Current literature comparison in mobile ransomware detection.

| Authors | Approach | Weaknesses |
|---|---|---|
| Andronio et al. [1] | Text-based classification | Natural language dependent |
| Yang et al. [17] | High-level design | No implementation |
| Song et al. [16] | Process Monitoring | Evaluation on one self-developed sample |
| Mercaldo et al. [11] | Formal methods | Requires human intervention for rules building |
| Gharib et al. [6] | Hybrid detection | Dynamic detection is applied only on *Suspicious* applications |

by the human analyst effort required to build the logic rules. As a matter of fact the proposed method foresees the payload identification but the process rule building has to be done by hand, and as such, is a time consuming task.

A hybrid static-dynamic approach is proposed in [6]. In that work, applications are first scanned by using a static approach and marked as *Benign*, *Suspicious*, or *Malware*; only the applications marked as suspicious are then examined by using a dynamic approach. The static approach is based on text and image classification as well as on API calls and application permissions. The dynamic approach is based on sequences of API calls that are compared, with a periodicity of five minutes, against malicious sequences identified for each malware family. While global performance of the method is not clearly reported, it can be derived that the authors can obtain a recall of 98% with a number of false positives below 1.5%. While applying dynamic detection only on applications marked as *Suspicious* provides the ability to reduce the system overhead, it leaves the possibility of having malware not recognized as such (e.g., due to obfuscation) by static detection running on the system. Dynamic detection takes a minimum of five minutes to identify a ransomware sample.

Table 7 shows a comparison between the state-of-the-art methods for mobile ransomware detection. The only methods, as depicted in Table 7, that are able to discriminate automatically between ransomware and trusted applications are the ones proposed in [1] and in [6]. The former has the limitation that if the ransomware applications are translated into different languages than the original ones, their detector is not able to identify the threat unless re-trained. The latter, is a hybrid approach that has very good detection performance, but it applies dynamic detection only on a subset of applications that are not identified as ransomware and it takes up to five minutes to detect malware by using the dynamic method. On the other hand, the approach proposed by Mercaldo et al. in [11] is able also to localize the malicious behaviour and to obtain a precision equal to 1 with a recall equal to 0.99 but it requires the formulation of logical rules that require human intervention.

The approach that we propose is different than all the others proposed in the current literature, since it is first, independent from the language application (for instance, we consider for the static analysis features derived from the structural characteristics of the code) and second, the considered extracted features (for both dynamic and static analysis) are fully automated and do not require the security analyst intervention. Additionally, our approach considers as target for dynamic detection all the running applications. In this way, also ransomware that is able to evade static detection, (e.g., by means of code obfuscation) can be identified. While our method requires continuous monitoring of the applications, it uses features and methods that are not as demanding as sequences of API calls and it can detect ransomware, on average in 20-60s, depending on the configuration chosen. The main weakness of the method proposed by Song et al. [16] is that the authors use only one ransomware sample, developed by them, to evaluate their solution. This makes any comparison with other methods, in terms of detection performance, impossible. Considering that approach proposed in [17] is only a high-level design with no implementation and validation about its effectiveness, it is not possible to perform a comparison between our proposal and the considered approach.

## 6 Conclusions

Mobile ransomware attacks are on the rise and pose threat not only to companies and governmental institutions but to the entire society. Both static and dynamic methods commonly used for ransomware detection offer good performance alone, but, according to our results, none of them can detect all different ransomware samples. Also other state-of-the-art methods do not provide effective solution.

In this paper, we propose a hybrid approach to ransomware detection that has a 100% detection rate coupled with a false positive rate below 4%, even when analyzing previously unseen applications. This performance was achieved using dynamic method to complement the static one, thus increasing coverage and allowing us to put together the advantages of both methods. Finally, given high achieved detection accuracy on one hand and the detection methods of low complexity used for on-device dynamic detection on the other, we foresee that such hybrid method can be used to detect ransomware not only in mobile phones but also in other IoT devices.

## Acknowledgements

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.

## References

1. Andronio, N., Zanero, S., Maggi, F.: Heldroid: Dissecting and detecting mobile ransomware. In: International Workshop on Recent Advances in Intrusion Detection. pp. 382–404. Springer (2015)
2. Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., Visaggio, C.A.: Effectiveness of opcode ngrams for detection of multi family android malware. In: Availability, Reliability and Security (ARES), 2015 10th International Conference on. pp. 333–340. IEEE (2015)
3. Canfora, G., Mercaldo, F., Visaggio, C.A.: Evaluating op-code frequency histograms in malware and third-party mobile applications. In: E-Business and Telecommunications, pp. 201–222. Springer (2015)

4. Canfora, G., Mercaldo, F., Visaggio, C.A.: Mobile malware detection using op-code frequency histograms. In: Proceedings of International Conference on Security and Cryptography (SECRYPT) (2015)
5. Carbonell, J.G., Michalski, R.S., Mitchell, T.M.: An overview of machine learning. In: Machine learning, pp. 3–23. Springer (1983)
6. Gharib, A., Ghorbani, A.: Dna-droid: A real-time android ransomware detection framework. In: Yan, Z., Molva, R., Mazurczyk, W., Kantola, R. (eds.) Network and System Security: 11th International Conference, NSS 2017, Helsinki, Finland, August 21–23, 2017, Proceedings. pp. 184–198. Springer International Publishing, Cham (2017), `https://doi.org/10.1007/978-3-319-64701-2_14`
7. Institute, I.: Evolution in the World of Cyber Crime. Tech. rep., Infosec Institute (June 2016), `http://resources.infosecinstitute.com/evolution-in-the-world-of-cyber-crime/#gref`
8. Labs, M.: McAfee Labs Threats Report – December 2016. Tech. rep., McAfee Labs (August 2016), `https://www.mcafee.com/au/resources/reports/rp-quarterly-threats-dec-2016.pdf`
9. Martinelli, F., Mercaldo, F., Saracino, A.: Bridemaid: An hybrid tool for accurate detection of android malware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 899–901. ACM (2017)
10. Martinelli, F., Mercaldo, F., Saracino, A., Visaggio, C.A.: I find your behavior disturbing: Static and dynamic app behavioral analysis for detection of android malware. In: Privacy, Security and Trust (PST), 2016 14th Annual Conference on. pp. 129–136. IEEE (2016)
11. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Ransomware steals your phone. formal methods rescue it. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems. pp. 212–221. Springer (2016)
12. Mercaldo, F., Visaggio, C.A., Canfora, G., Cimitile, A.: Mobile malware detection in the real world. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 744–746. ACM (2016)
13. Milosevic, J., Ferrante, A., Malek, M.: Malaware: Effective and efficient run-time mobile malware detector. In: The 14th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC 2016). IEEE Computer Society Press, IEEE Computer Society Press, Auckland, New Zealand (08/2016 2016)
14. Milosevic, J., Malek, M., Ferrante, A.: A Friend or a Foe? Detecting Malware Using Memory and CPU Features. In: SECRYPT 2016, 13th International Conference on Security and Cryptography (2016)
15. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. pp. 329–334. ACM (2013)
16. Song, S., Kim, B., Lee, S.: The effective ransomware prevention technique using process monitoring on android platform. Mobile Information Systems 2016 (2016)
17. Yang, T., Yang, Y., Qian, K., Lo, D.C.T., Qian, Y., Tao, L.: Automated detection and analysis for android ransomware. In: IEEE 17th International Conference on High Performance Computing and Communications, IEEE 7th International Symposium on Cyberspace Safety and Security, IEEE 12th International Conference on Embedded Software and Systems. pp. 1338–1343. IEEE (2015)
18. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 95–109. IEEE (2012)