Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 18, 2016

Malware detector for Android

Real time detection of malware on Android devices

Michele Chersich

Abstract

Malware is defined as any malicious software that gains access to a device for the purpose of stealing data, damaging the device, or annoying the user. The rise of mobile, network connected devices, also led to the rise of mobile malware. Current malware detection methods are divided in two groups: static and dynamic. Static methods are lightweight and suitable for constrained resources of mobile devices, but they are unable to detect malware at run-time, whereas dynamic methods are usually too complex to be run on mobile devices. The goal of this project is to implement lightweight dynamic detection on Android devices, assessing the behavior of programs at run-time, by monitoring a relatively small set of features (related to CPU and RAM) and evaluating these features by means of suitable detection algorithms. The originality of this approach lies in that the use of features related to CPU and RAM as a standalone feature set in mobile malware detection has never been taken into account so far. This lightweight approach is expected to work on any mobile operating system.

Advisor
Prof. Miroslaw Malek
Assistants
Dr. Alberto Ferrante, Jelena Milosevic

Advisor's approval (Prof. Miroslaw Malek): Date:

Contents

1 Introduction

The increasing number of mobile devices and their widespread usage reflects in the increase of mobile malware. The total number of mobile malware samples in 2015 has grown all over the year, reaching its maximum growth of 72% in the fourth quarter [4]. According to McAfee, Google's August 2015 notification that it would release monthly updates to its Android mobile operating system forced malware authors to develop new malware more frequently in response to the enhanced security in each monthly release of the operating system [4].

Therefore, the need for protection from malware is increasingly growing, resulting in higher demands for effective malware detection systems. Furthermore, because of the constrained computational resources of mobile devices, building such systems is a challenging task. Mobile devices are battery-operated and this significantly limits their ability to run complex malware detection systems. Existing solutions are either accurate, but too complex to be employed at run-time and within limited energy budget, or lightweight but unable to detect the large amount of malware that appears almost daily.

Two existing approaches to malware detection are static and dynamic analysis. Static detection is based on offline investigation of applications by means of static features (e.g. permissions, manifest files, API calls). It is efficient, but it performs poorly, considering the ever increasing number of malware samples. Dynamic malware detection is based on observing systems behavior at run-time. It requires complex algorithms, that can weigh too much on battery-operated mobile environments. Thus the need for solutions that are both lightweight and dynamic, being able to discriminate between malware and trusted behavior, without being too heavy a burden for mobile devices.

This project is about an Android application that implements MalAware [8], a lightweight dynamic malware detection method. Its main characteristics are the following:

- It is able to detect previously unseen malware samples
- It detects malicious applications at the beginning of their execution
- It uses only seven dynamic features related to memory and CPU usage, that can be easily collected on mobile devices
- It uses a linear classification algorithm that has linear complexity in the number of features and limited memory requirements
- It uses a simple sliding window method that is able to discriminate between malicious and trusted application at run-time

Also, MalAware is composed of two detection levels: the first one evaluates an application's behavior at a given time (record-level detection), whereas the second one evaluates the recent history of records (application-level detection). The final outcome is determined by application-level detection.

According to [8], in case of previously unseen malware samples, MalAware is able to detect 85.5% of malicious applications, with false positive rate of 17.2% and within first 2 minutes and 45 seconds of execution.

In this project, a working Android application that implements MalAware has been developed. Testing the implemented application has produced a malware detection rate of 88% and a false positive rate of 0%. These results are encouraging, as compared to the ones obtained experimentally. Therefore, the identity of MalAware as a malware detection method has been strengthened. Also, it has proven to be suitable for the constrained resources of mobile devices, thanks to the low computational overhead of its algorithms.

The rest of the report is organized as follows: in Section 2 an outline of the research on MalAware is given; in Section 3 the project requirements are specified; in Section 4 the system architecture and the main design choices are illustrated; in Section 5 the main implementation steps and issues are examined; Section 6 the test results of the implemented application on the Android emulator are presented; finally, in Section 7 the relevant conclusions are derived.

2 Related work

This project is mainly based on MalAware [8], a lightweight dynamic malware detection method. The main points, upon which the method is developed, are:

- malware detection can be performed at run-time
- there exist CPU and RAM related features that are relevant to malware detection
- training of algorithms can be performed with a dataset of known applications, and then the algorithms should be able to evaluate the behavior of any unknown application

The MalAware run-time detection system is composed of the following four blocks: *Selected features monitoring*, *Record-level classification*, *Application-level classification*, and *Alarm*. *Selected features monitoring* monitors suitable system parameters with a predefined frequency and extracts features from them. *Record-level classification* classifies every execution record as bad or good (i.e. potentially belonging to a malicious application or not). *Application-level classification* considers the recent history of classified execution records to classify an application as malware or not. The *Alarm* block is used to raise an alarm in case a malicious application is found.

The training of the blocks to be used at run-time is performed offline. The features to be monitored by the *Selected features monitoring* block were identified by using a set of feature selection methods. An effective approach to identification of these features is presented in [7]. The approach is validated by analysing the features related to memory and CPU usage during execution of 1080 mobile malware samples, belonging to 25 malware families coming from the Malware Genome Dataset. The results show that there are features that are more frequent and more significant than others.

The evaluation of these features as potentially belonging to malicious applications or not is performed by *Record-level classification*, for which Logistic Regression with ridge estimator has been chosen as linear classification algorithm, providing the best trade-off between performance and lowest model size and testing time. The selected features have been used to train the Logistic Regression algorithm, that afterwards has been employed to classify unknown execution records.

In MalAware, the final classification of an application as either malware of trusted is performed by *Application-level classification*. This method only needs to keep a limited history of past execution record classifications and is based on a sliding window mechanism: considering a sliding window of length n, the percentage of records classified as malware in the n most recent records is used: if it is greater than a threshold t, then the window is marked as malware. For the sake of robustness, multiple results, obtained in disjoint sliding windows, are considered: when w windows are marked as malware, the application is classified as malware. Figure 1 shows an example of this algorithm, where a malicious application is identified at sample 12.

Using a dataset of malware samples coming from the Malware Genome Project [11] and trusted samples coming from Google Play Store, a set of features has been observed and the most significant ones have been selected, based on statistical significance and information gain measure. Then, the obtained features have been used to train *Record-level classification*. Also, parameters for *Application-level classification* (window length, threshold and number of disjoint sliding windows) have been tuned, based on the dataset.

Here follow the results of the experiment. The features recognized as the most indicative are *.so mmap Shared Dirty*, *.jar mmap Pss*, *.ttf mmap Pss*, *.dex mmap Private Dirty*, *Other mmap Private Dirty*, *CPU Total*. The first six are related to memory usage, while the last one is related to CPU usage. More details can be found in Table 1. The best parameter set for *Application-level classification* has been found to be: $n = 5, t = 85\%, w = 15$, considering three different metrics: highest F-measure, best malware detection rate (with false positive rate below 30%) and lowest false positive rate (with malware detection rate greater than 70%).

MalAware is efficient and therefore suitable for mobile devices. The small number of features considered (down to seven) and the detection algorithm with linear complexity in the number of features and records guarantee efficiency.

The results show that the seven features related to CPU and memory that have been selected, contain enough information to discriminate between malicious and trusted applications.

3 Project requirements and analysis

The goal of this project is to detect malware at run-time, monitoring the behavior of applications and evaluating it. In order to do this, the application implements MalAware [8] (see section 1), a lightweight dynamic malware detection method. The main tasks involved are the following:

- *Collect features in records*: a selected set of system features, regarding running applications, need to be read and collected into records.
- *Store records*: records need to be stored in a database for future evaluations.
- *Evaluate records*: every record needs to be classified as either malware or trusted.
- *Evaluate history of records*: to determine whether an application actually behaves as either malware or trusted, the recent history of records is evaluated. If, e.g., a certain number of consecutive malware records is found, then malware behavior could be confirmed (for more details, refer to section 5).
- *Display the results*: visualize the output as soon as malware is found, or after a timeout occurs if no malware entry is found.

3.1 Selected features monitoring

Selected features monitoring consists in periodically reading (with a predefined frequency) the values of the corresponding system parameters, extracting and organizing them in execution records. The following set of system features has to be monitored:

Category	**Feature Name**	**Feature Description**
Memory mapped native code	.so mmap Shared Dirty	Shared memory, in the *Dirty* state, being used for Dalvik or ART code. The *Dirty* state is due to fixups to the native code when it is loaded into its final address
Memory-mapped Dalvik code	.dex mmap PSS	Memory usage for Dalvik or ART code, including pages shared among processes
	.dex mmap Private Dirty	Private memory, in the *Dirty* state, being used for Dalvik or ART code. The *Dirty* state is due to fixups to the native code when it is loaded into its final address
Memory-mapped fonts	.ttf mmap PSS	Memory usage for true type fonts, including pages shared among processes
CPU usage	CPU Total	Total (User + System) CPU usage by the considered application
Other memory-mapped files and devices	Other mmap Private Dirty	Private memory used by unclassified contents that is in the *dirty* state
	.jar mmap PSS	Memory usage for Java archives, including pages shared among processes

Table 1. CPU and RAM related features [8]

As we can see in the table, there is one CPU-related feature and six RAM-related features. Due to the inability to monitor *.so mmap shared dirty* feature in all platform versions [2], only the other six features were monitored. The values of all features need to be read at regular, user-defined time intervals and then evaluated, as explained in the following subsection.

3.2 Record-level Classification

In order to evaluate a single record to either malware or trusted, the Logistic Regression algorithm [6] with ridge estimator is used. As by the experiment reported in [8], the selected features have been used to train the algorithm. During the training process, it has been observed that logistic regression performs well on record-level classification. For this reason, weights obtained in the training process are used at run-time in order to evaluate each record. In the case of the selected features, the following weights have been obtained:

Feature	Weight
.jar mmap Pss	0.5386
.ttf mmap Pss	0.0046
.dex mmap Pss	-0.0069
.dex mmap Private Dirty	0.0137
Other mmap Private Dirty	-0.0646
CPU Total	-0.0456
Intercept	4.4309

Table 2. Weight of every feature in record-level classification

The intercept value is simply the y-intercept of the linear classifier.

Once we have the trained classifier, let X be a record, β the vector of weights and p the probability of the record being malware. Then we can compute p, as follows:

$$p(X) = exp(\sum_{i=1}^{6} \beta_i X_i)/\{1 + exp(\sum_{i=1}^{6} \beta_i X_i)\} \quad [6]$$

If the probability is greater than 0.5, then the record is classified as malware, otherwise it is classified as trusted.

3.3 Application-level Classification

In order to ultimately evaluate an application's behavior, the recent history of record-level classifications is considered. As by [8], application-level classification is based on a sliding window algorithm. Considering a window length n, a threshold t and a number of checks w, then an application is marked as malware if the following conditions hold:

- let m be the number of malware records in a window and let $\rho = m/n$ be the malware rate: then, a window is marked as malware if $\rho \geq t$
- if w disjoint windows are marked as malware, the application is identified as malware

By disjoint window, we mean that they have no overlapping elements with respect to each other. An example of the sliding window algorithm is shown in Figure 1.

According to [8], the best parameter choice is $n = 5$, $t = 85\%$ and $w = 15$, yielding a malware detection rate of 85.5%, a false positive rate of 17.2% and F-measure equal to 0.85.

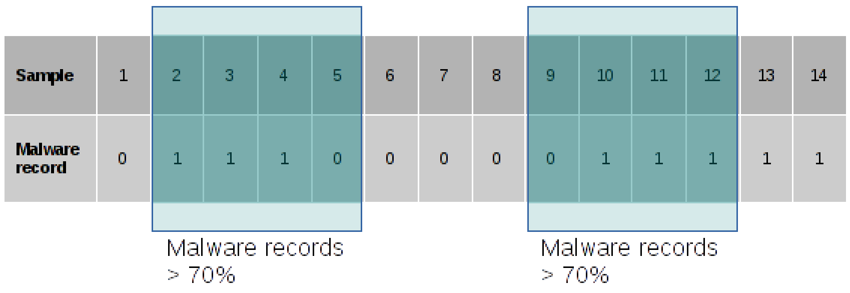


Figure 1. An example of the sliding window algorithm, with n=4, t=70%, w=2 [8]

3.4 Graphical User Interface

The Graphical User Interface is devoted to two tasks: informing the user that the device is being analyzed (Figure 2), and notifying the user with analysis results, either positive or negative (Figure 3). If malware is detected, only the red light should be lit; otherwise, only the green light. The output of the lights should be supported by the total count of malware and trusted applications found. Below the lights, the list of malware applications detected, if any, should be displayed.

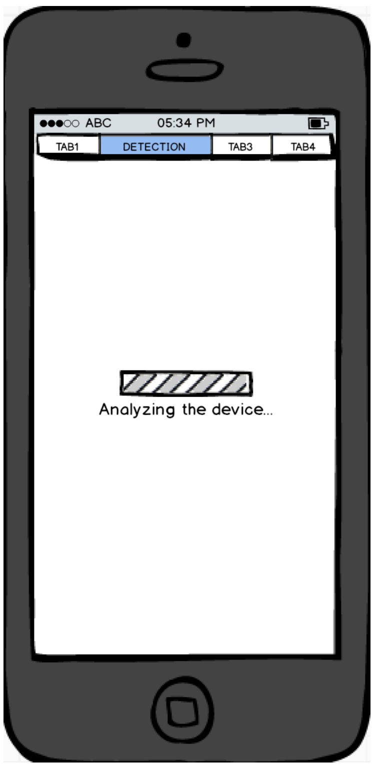


Figure 2. Analyzing the device

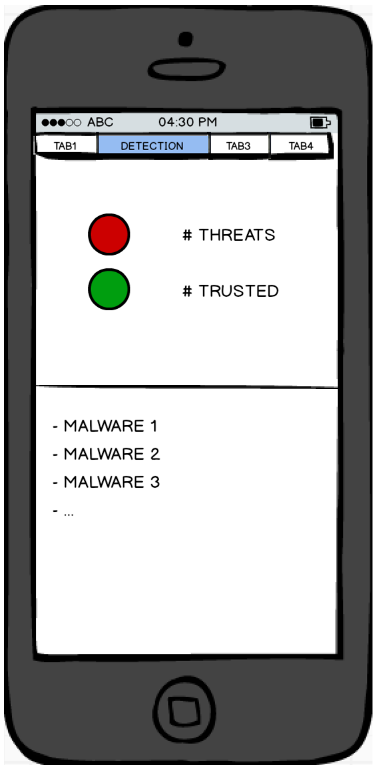


Figure 3. Displaying the results

4 Project design

The core of the application consist of three services: Monitor, Detector, and Database.

The Monitor module reads system information about running applications and extracts a selected set of features, collecting them together in records, and stores these records in the database. Then, according to a user-defined time frequency (e.g. every 2 seconds), it sends the records to the Detector module.

The purpose of the Detector module is twofold: detecting malware behavior both on the single record (Record-level detection) and on the history of records (Application-level detection). When Application-level detection finds malware behavior, the GUI is notified.

The database is used by Monitor to store all records for future reference, and later on by Detector to include in each entry also the evaluation of each record, as output by *Record-level classification*.

The GUI informs the user with the results, which can be either: “no malware found" or “malware detected". Also, the GUI provides the user with “Settings" for the application (e.g. choosing the time frequency the Monitor uses to send its records to the Detector).
A flow chart that summarizes the behavior thus far described is shown in Figure 4 and the corresponding class diagram is shown in Figure 5.

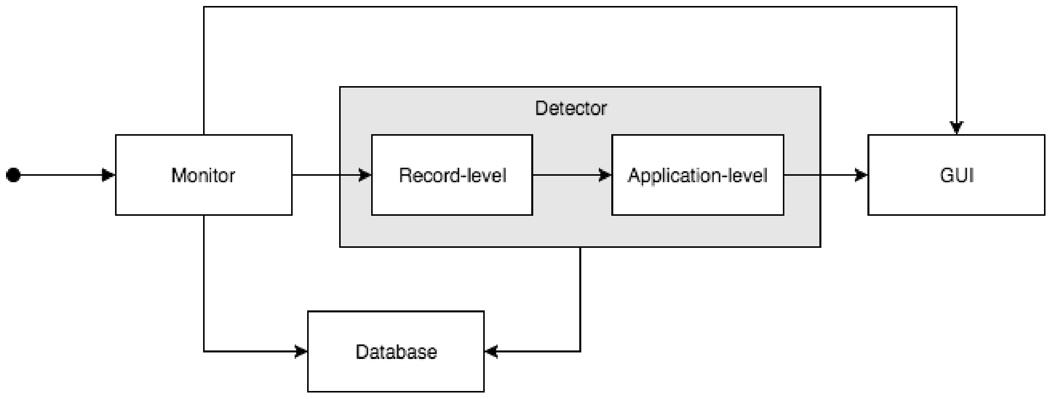


Figure 4. Flow chart. Monitor keeps sending records about CPU and RAM usage to Detector. Detector notifies the GUI when malware is found or after a timeout occurs, when it determines that there is no malware. Monitor stores the records in the database and Detector updates the stored entries with their record-level evaluations.

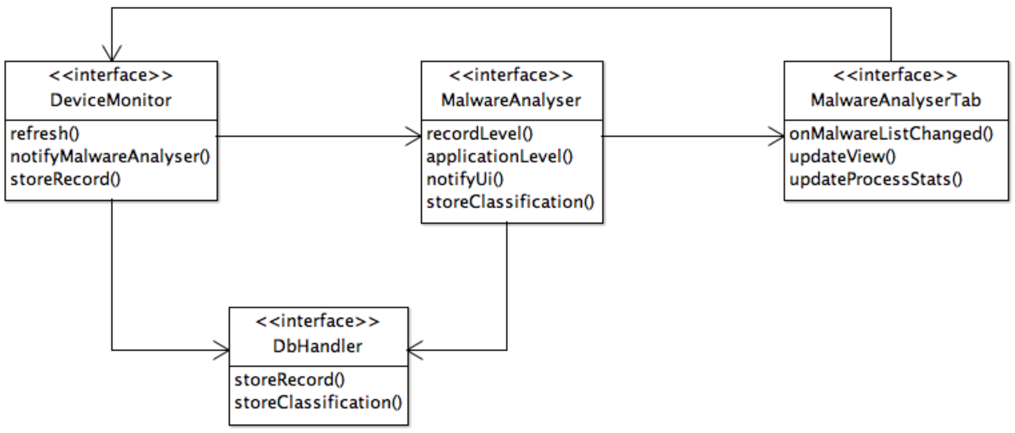


Figure 5. Class diagram. The behavior is the same as in Figure 4: method `refresh()` updates the usage information in the Monitor, `notifyMalwareAnalyser()` sends it to the Detector, that runs the two algorithms (`recordLevel()`, `applicationLevel()`) and, if needed, notifies the GUI.

The two diagrams above illustrate the same behavior: the DeviceMonitor interface implements the Monitor unit from the previous diagram, MalwareAnalyser implements the Detector unit, MalwareAnalyserTab the GUI and DbHandler the database. The three services mentioned at the beginning of this section are threads running in parallel, servicing requests from one another, communicating via message-passing and synchronizing via mutex-locks. For example, the MalwareAnalyser service, while running its detection algorithms, may need to perform synchronized reads and writes to the list of active processes, as the same list is read and written by DeviceMonitor in the meantime.

5 Implementation

As a basis for implementing malware detection, a monitoring application was provided, developed as part of a master thesis. The base application was built upon three main core components: monitor (*DeviceMonitor*), detector (*MalwareAnalyser*) and database (*DbHandler*), which are threads that run in parallel, servicing requests from one another. In the base program, only *DeviceMonitor* and *DbHandler* were implemented, to monitor the usage by applications and processes of CPU, RAM and a few other user-selected features.

The goal of this project is to implement malware detection, which is done in two phases: extending the monitoring module to create records of CPU and RAM-related features and then implementing the detection module, to collect and evaluate these records by means of suitable algorithms.

5.1 Monitoring

The base *DeviceMonitor* module collects information about the usage by applications and, optionally, by all system processes of CPU, RAM, battery and display. At regular time intervals (at the occurrence of a *timeout* event), selected by the user, the updated information is collected in records (*StatisticsData*) and sent to the GUI thread for visualization.

Since the monitoring module must serve the detection module, it had to be adapted, leading to two main issues: collecting the features and extend the records to contain them, and notifying the detection module that information has been updated and a new record is ready to be fetched.

The record (*StatisticsData*) has been extended to include 1 CPU-related feature and 5 RAM-related features (please refer to [7], [8], [9] and Table 1).

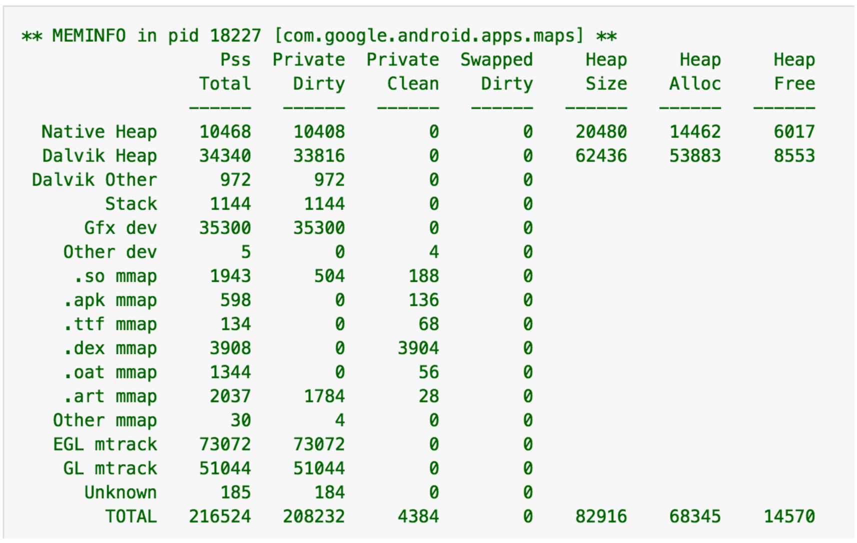
These features are collected by two classes: *Memory* and *Processor* (in the `refresh()` method, that implements the interface defined in the *ResourceUsage* abstract class). While the base *Processor* class collected the CPU usage already, *Memory* had to be extended to collect the remaining five features. The RAM-related features are collected by *Memory*, executing at run-time the following `adb [3] shell` command:

```
dumpsys meminfo <package_name|pid>
```

Executing this command requires a special Android permission: `android.permission.DUMP`, which is reserved to system processes. This protection measure can be bypassed executing the following command on the terminal, once the application is installed on the device:

```
adb shell pm grant alarilab.malwaremonitor android.permission.DUMP
```

The effect lasts until the application is uninstalled or a factory reset is performed on the device, and grants the application the DUMP permission, required to execute `dumpsys`. The output of `dumpsys meminfo` looks like the one shown in Figure 6.



```
** MEMINFO in pid 18227 [com.google.android.apps.maps] **
                 Pss  Private  Private  Swapped     Heap     Heap     Heap
               Total    Dirty    Clean    Dirty     Size    Alloc     Free
              ------   ------   ------   ------   ------   ------   ------
  Native Heap  10468    10408        0        0    20480    14462     6017
  Dalvik Heap  34340    33816        0        0    62436    53883     8553
 Dalvik Other    972      972        0        0
        Stack   1144     1144        0        0
      Gfx dev  35300    35300        0        0
    Other dev      5        0        4        0
     .so mmap   1943      504      188        0
    .apk mmap    598        0      136        0
    .ttf mmap    134        0       68        0
    .dex mmap   3908        0     3904        0
    .oat mmap   1344        0       56        0
    .art mmap   2037     1784       28        0
   Other mmap     30        4        0        0
   EGL mtrack  73072    73072        0        0
    GL mtrack  51044    51044        0        0
      Unknown    185      184        0        0
        TOTAL 216524   208232     4384        0    82916    68345    14570
```

Figure 6. dumpsys sample output [2]

The information collected in the output may vary slightly across platform versions [2]. The *Memory* class reads the values for the five required features and puts them in the record, which is now ready to be fetched by the detection unit. It is worth noting that a record gathers information about a single system resource (e.g. CPU, RAM), therefore, in order to get all six features, two records are created, one by the *Processor* class, containing the CPU usage value, and the other by the *Memory* class, containing the RAM-related features. The two records will be merged into one at a later stage, by the detection unit.

The other modification required is that the monitoring module notify the detection module at regular time intervals that a new record is ready to be fetched: in order to achieve this, a notification routine has been added to be called every time a timeout event occurs in the monitoring module (see *DeviceInfos* class, `onTimeOutEvent(Message)` and `notifyMalwareAnalyser() methods`). A message is sent to the detection unit, which in turn fetches the record from the monitoring unit. As said before, the record are actually two, one for the CPU-related feature and one for the RAM-related features. At this stage, the two records are collected and merged into one.

The one record now contains all feature values that can be evaluated in the next step: record-level detection.

5.2 Detection

Detection is implemented in class *MalwareAnalyser*: after receiving a notification message, containing as parameter the list of active processes, it acquires a lock on the list of processes (see method `HandleMessage(Message)`) and then runs the detection algorithm (method `runDetection()`). The lock will be released when the execution of `runDetection()` reaches the end. This ensures that the list of active processes is consistent along a single iteration of the algorithm.

After fetching the list of records (*StatisticsData*) from the monitoring unit, every record is first classified as either malware (0) or trusted (1) by the Record-level Classification algorithm (for more details, see section 3.2 and class *RecordEvaluator*). Then, the result of Record-level evaluation is passed to the Application-level classification algorithm (see section 3.3 and class *ApplicationEvaluator)*, that maintains the recent history of records up to window length n, updating the history using a sliding window method, as follows (a reminder on the parameters: window length $n = 5$, threshold $t = 85\%$, number of checks $w = 15$):

1. the record evaluation is added to the window, removing the least recent one if the window is full at insertion
2. if the window is full, it is evaluated to either malware or trusted
3. if malware, the window slides by its whole length n, i.e. all records contained in the window are deleted; if trusted, it just slides by 1 record, which is done at step 1. at the next iteration
4. if the number of malware window evaluations reaches w, then the application in the list of active processes is marked as malware and the list is passed to the GUI, in order to display the positive results
5. every 5 minutes, if no malware was found, the GUI is notified, in order to display the negative results

5.3 GUI

The existing MalwareMonitor application provides a graphical user interface, organized in three tabs: "PROCESSES", visualizing the active applications and their usage information about CPU, RAM and battery; "SYSTEMINFOS", displaying hardware information about CPU, RAM and battery; "SETTINGS", allowing the user to enable and disable services and set several timings for refreshing system information (e.g. UI refresh time, CPU Infos refresh time, etc.).

As the detection service is added to the application, a new tab is required: "DETECTION" class `MalwareAnalyserTab`, whose purpose is to display the output of the device analysis. The detection service can be in exactly two states: analyzing the device or displaying the results. In the first state, a simple progress bar and a status message are shown (see Figure 2); in the second, the output depends on whether *MalwareAnalyser* notifies malware has been found or not. In either case, a red or green light is lit and the count of malware and trusted apps is displayed. If malware has been found, a `ListAdapter` is used to update a `ListView` with the contents of the list of processes received by *MalwareAnalyser*, including information on CPU and RAM usage. Otherwise, a simple text message is displayed to notify the user that no malware has been detected.

The developed application provides an intuitive graphical user interface, that could be further improved in case of more extensive applications usage.

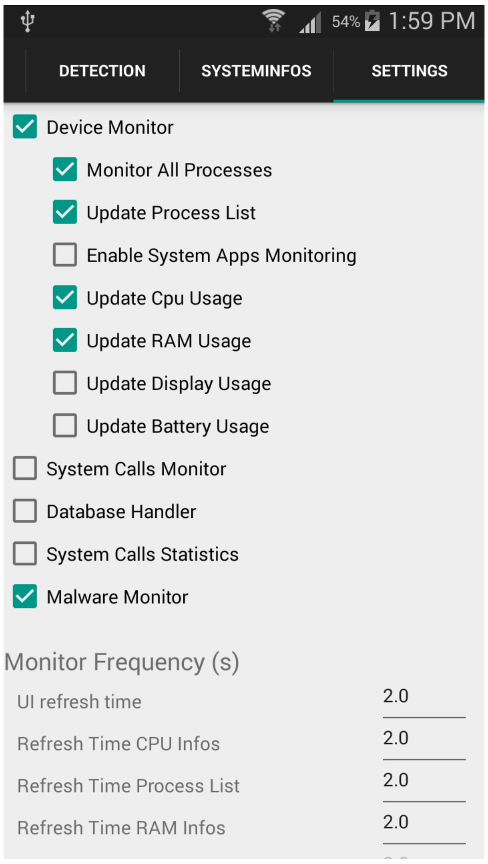
6 Results

In order to see whether the application's results are compliant with results reported in [8], tests have been carried out on the Android emulator (Nexus 5, API 21, Android 5.0, Intel x86_64), using a dataset of 50 malware samples and 50 trusted samples. The trusted samples are taken from the Play Store, whereas the malware samples belong to three different malware families (illustrated and described in Table 3) and are taken from the Malware Genome project [11].

Malware Family	**Description**
FakePlayer	Trojans belonging to this family pretend to be a movie player, but instead send SMS messages.
Geinimi	Trojans belonging to this family send personal data (location coordinates, device identifiers, the list of installed apps) to remote servers.
GingerMaster	Trojans belonging to this family are often bundled with trusted applications and try to gain root access, steal sensitive data and send it to remote servers.

Table 3. Malware families the used malware testing samples belong to [10] [5]

In order to test the application, the Settings tab has been configured as in Figure 7, enabling the monitoring service:



- [✓] Device Monitor
  - [✓] Monitor All Processes
  - [✓] Update Process List
  - [ ] Enable System Apps Monitoring
  - [✓] Update Cpu Usage
  - [✓] Update RAM Usage
  - [ ] Update Display Usage
  - [ ] Update Battery Usage
- [ ] System Calls Monitor
- [ ] Database Handler
- [ ] System Calls Statistics
- [✓] Malware Monitor

Monitor Frequency (s)

UI refresh time	2.0
Refresh Time CPU Infos	2.0
Refresh Time Process List	2.0
Refresh Time RAM Infos	2.0

Figure 7. Settings used for testing the application

In order to communicate with the emulated device, Android Debug Bridge (adb) [3] has been used. For every testing sample, a clean installation of Android has been performed on the emulator, the sample application has been installed by means of adb and launched by using Monkey (an adb shell tool, that sends a stream of user events into the system), the MalwareDetector application has been installed and launched from Android Studio [1].

Malware Family	**Detection Accuracy**	**Detection Accuracy (%)**
FakePlayer	6/6	100%
Geinimi	38/40	95%
GingerMaster	0/4	0%
MALWARE	**44/50**	88%
TRUSTED	**50/50**	100%

Table 4. Test results on both malware and trusted samples. Malware detection rate is 88% and false positive results are 0%.

The results are shown in Table 4 and are compliant with the results obtained in [8] (see also section 1). The average accuracy of obtained results is 88% for malware and 100% for trusted applications. As listed in Table 4, the detection accuracy for different malware families varies. One reason for such variation (from 0% to 100%) might be in the small number of considered malware samples within each family. However, it can also happen that due to the similar behavior of GingerMaster malware samples with the trusted ones (as described above), their detection require additional information about the system, while for the other considered malware families memory and CPU information is enough to discriminate.

To sum up, the main contributions that arise from this project are:

- The implementation of the MalAware, lightweight malware detection method, in an Android environment
- The adoption and extension of the existing monitoring infrastructure in order to support collection of features related to CPU and memory behavior
- The development of Graphical User Interface for visualization
- The development of modular infrastructure, suitable for extension and addition of new detection components
- Testing and validation of the implemented approach both on the emulator and on the mobile device

7 Conclusions

A modular application for run-time malware detection on Android has been developed. The coherence between the background research [8] and the implementation has been verified by testing on the emulator. Even though the project is still in its experimental phase, the results are encouraging: 88% malware detection rate and 0% false positive rate. The results gathered by testing the application are even better than the ones obtained experimentally (85.5% and 17.2%). However, as the dataset tested so far is rather small, there is room for further testing on larger datasets, including samples from other malware families.

At a later stage, also unit testing will be performed, in order to ensure the correct behavior of every functional unit in the system (e.g. in monitoring, test that the features are read correctly; in record-level classification, test that a record is evaluated as expected; in application-level classification, test that a history of records contains the expected number of malicious windows, etc.).

In general, the test results reinforce the identity of MalAware as a method for malware detection on mobile devices. The selection of CPU and RAM related features, along with the *Record-level classification* and *Application-level classification* algorithms, are reliable tools in evaluating the behavior of both malicious and trusted applications.

The capability of evaluating the behavior of applications with unknown origin, after training the algorithms with a dataset of known applications, is very interesting, as it makes room for the assumption that evaluating the behavior of any unknown application by analysing the behavior of a limited set of known applications is possible. Yet, this assumption has to be verified.

References

[1] Android Studio. `https://developer.android.com/studio/index.html`.

[2] Android Studio User Guide. `https://developer.android.com/studio/profile/investigate-ram.html`.

[3] Android Debug Bridge. `http://developer.android.com/tools/help/adb.html`, 2015.

[4] Mcafee lab threats report. Tech. rep., McAfee Labs (March 2016), `http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf`, 2016.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[6] S. Le Cessie and J. Van Houwelingen. Ridge estimators in logistic regression. University of Leiden, The Netherlands, 1990.

[7] J. Milosevic, A. Ferrante, and M. Malek. What Does the Memory Say? Towards the most indicative features for efficient malware detection. Advanced Learning and Research Institute, Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland.

[8] J. Milosevic, A. Ferrante, and M. Malek. MalAware: Effective and Efficient Run-time Mobile Malware Detector. In *The 14th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC 2016)*, Auckland, New Zealand, 08/2016 In Press. IEEE Computer Society Press.

[9] J. Milosevic, M. Malek, and A. Ferrante. A Friend or a Foe? Detecting Malware Using Memory and CPU Features. In *13th International Conference on Security and Cryptography (SECRYPT 2016)*, Lisbon, Portugal, 07/2016 In Press. SciTePress Digital Library.

[10] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into Android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1808-1815, ACM*. SAC '13, New York, NY, USA, 2013.

[11] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy, pp. 95-109*. SP '12, IEEE Computer Society, Washington DC, USA, 2012.