Received 22 June 2023, accepted 9 July 2023, date of publication 18 July 2023, date of current version 21 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3296606

RESEARCH ARTICLE

A Lightweight and Multi-Stage Approach for Android Malware Detection Using Non-Invasive Machine Learning Techniques

LEONARDO DA COSTA[1] **AND VITOR MOIA**[1,2]
[1]Samsung Research and Development Institute Brazil (SRBR), Campinas, São Paulo 13097-104, Brazil
[2]Eldorado Institute, Campinas, São Paulo 13083-898, Brazil

Corresponding author: Leonardo da Costa (leonardo.bc@samsung.com)

Part of the results presented in this work was obtained through the Defeat Exploits (DEFEX) project, funded by Samsung Eletrônica da Amazônia Ltda., under the Brazilian Informatics Law 8.248/91.

ABSTRACT Android has been a constant target of cybercriminals that try to attack one of the most used operating systems, commonly using malicious applications (denominated *malware*) that, once installed on a device, can harm users in several ways. Existing *malware* detection solutions are usually invasive as they obtain classification features by performing reverse engineering, decompilation, or disassembly of the analyzed application, which infringes licenses and terms of use of applications. In addition, these solutions often employ a single machine learning (ML) model to detect various types of *malware*, resulting in several false alarms. In this context, we propose an approach to detect Android *malware* consisting of a set of specific-type detectors in which each one performs a multi-stage analysis, based on rules and ML techniques, in different phases of the application cycle (before and after its installation). Our approach also differs from state-of-the-art solutions by being non-invasive, since it leverages a process to obtain application's features that does not infringe licenses and terms of use of applications. In addition, according to experiments performed on a real Android smartphone, our proposal presents the following additional advantages over state-of-the-art solutions: a more efficient process to classify applications that is three times faster and requires ten times less CPU usage in some cases (saving device energy); and a better detection performance, with higher balanced accuracy, nine times less false positive cases, and ten times less false negative cases.

INDEX TERMS Android, machine learning, malware detection, multi-stage analysis, non-invasive feature extraction.

I. INTRODUCTION

Android has been constant target of attacks through the use of malicious applications (*malware*) that can harm users, for instance, by leaking sensitive data (e.g., bank account details), blocking access to information and demanding monetary compensation for the ransom, or even leveraging social engineering scams. A large number of malicious applications is distributed in a daily basis through different distribution vectors [1], such as application markets, including official and third party stores, or untrusted sources like Web repositories.

The associate editor coordinating the review of this manuscript and approving it for publication was Li He.

The huge number of existing Android *malware* [2], and the high speed in which new malicious applications are created have motivated the security community to design and continuously improve solutions to detect this threat.

An Android application is distributed via an APK (Android Application Package) file, containing all the necessary data to install the application on a device. Although the processing power of mobile devices has been evolving over the years, these devices are still restricted to their battery life, and it is paramount to continue creating applications and solutions for such devices that do not significantly impact their resources. For this reason, a common approach employed to analyze Android applications for malicious behavior is still through static analysis, using data obtained from the APK file without the need for running the application. Some of the components that can be verified during this analysis are:

- ***Manifest* file:** contains metadata related to the application, such as *permissions* and *intents*. *Permissions* are elements of the Android framework that allow users to control which applications can have access to certain device resources. For example, the *CALL_PHONE permission* allows an application to place phone calls. All *permissions* that the application will request for the user at some point must be declared in the *Manifest* file. In turn, an *intent* is a messaging object used to request an action from another application component. For instance, the *DIAL intent* can be employed by an application to dial a phone number in the phone call application and show it to the user.
- **DEX files:** these are files in the Dalvik Executable format related to the application source code. Android applications are generally written in the Java or Kotlin programming languages, compiled to bytecodes understandable by the virtual machines corresponding to these languages, and then translated to Dalvik bytecodes stored in the DEX files. These files are used to execute the application on Android devices.
- ***Assets* folder:** contains files of different formats (e.g., audio, image) used by the application.

State-of-the-art solutions based on static analysis employ signature-based detection of *malware* APK files which is known for not being effective to identify modified and new *malware* patterns [3], [4]. To address this limitation, detectors based on machine learning (ML) have been proposed. They are capable of learning patterns of *malware* compositions and behaviors, and effectively detecting samples never seen before. Android *malware* detectors based on static analysis and ML have used different APK features. For example, metadata (e.g., *permissions*) declared in the *Manifest* file can form part of data (i.e., the feature vector) given as input to an ML model. Another example are API (Application Programming Interface) calls that can provide information on which Android API methods and classes the application uses in its source code, indicating its basic functionalities.

Previous ML-based solutions present drawbacks. First, they usually execute after the application installation on the device. Second, most of the solutions perform reverse engineering, decompilation, or disassembly of the APK file to obtain the data (i.e., the classification features) given to the ML classifier [5], [6], mostly related to API calls. These practices infringe many licenses and terms of use that developers create to protect their applications. Besides, the process to obtain the features and the large set of features used by previous solutions to perform classification can be inefficient and consume a considerable amount of the device's resources. Finally, previous solutions [7], [8], [9], [10], [11] usually employ a single and general detector to perform the classification of different types of *malware*, which may misclassify applications (i.e., *benign* applications are classified as *malware* - false positive case - or *malware* applications are classified as *benign* - false negative case) and make users eventually lose their trust on the security system given the many false alarms.

Solutions based on dynamic analysis and ML have been proposed with the aim of reducing the number of false alarms [12], [13]. They employ features obtained from the application execution, such as system calls, network data, performance metrics, etc. However, dynamic analysis is computationally expensive as it requires constantly monitoring the execution of an application, which can be even more inefficient when monitoring several applications at the same time. This is a paramount limitation due to the energy constraints of most Android devices. Moreover, dynamic analysis cannot be freely executed given the nature of the Android system that isolates applications using the sandbox model [14].

Solutions based on ML and dynamic analysis are more suitable when performed via an external (cloud) server (off-device analysis), to where the application is sent to be executed on an isolated environment for the detection of any malicious behavior. However, off-device and dynamic analysis-based solutions also contain major practical constraints. For example, the solution may either keep the user waiting for an analysis on the cloud and block the application installation on the device, or analyze the application in parallel, while the user runs it, blocking the application only in case malicious behavior is identified. Both approaches have drawbacks, since the first one will annoy users, while the second may be insecure (e.g., in case of a *ransomware* application). Other limitations are related to sending the (large-size) applications to the cloud for analysis or deciding for how long to perform the analysis (given that both it is difficult to find and trigger the malicious behavior, and emulated devices have restrictions [15]).

This work presents *Malware APK Detection Solution* (MADS), a novel approach to detect Android *malware* based on rules and machine learning techniques that overcomes the aforementioned limitations. Our approach consists of one or more detectors, wherein each detector is carefully designed to identify a single *malware* type (e.g., *ransomware*) using APK features obtained without the need to decompile or disassemble applications (i.e., without infringing licenses and terms of use of applications). Instead, features are obtained from the operating system in a non-invasive fashion by employing native functions of the Android operating system [16]. MADS detectors classify an APK as *malware* or *undetected* and comprises three analysis steps. The first one, based on a set of lightweight rules, checks for specific APK file characteristics (e.g., *permissions* requested) to indicate whether the application is a potential *malware* of interest. The second step analyzes the APK file based on a first ML model execution that receives as input a set of features obtained before the application installation. Last, the third analysis step employs a second ML model execution, which takes as input features that can be obtained after the APK file installation. Each analysis step of MADS may output a final classification label for an APK file, meaning that running the three steps is not always necessary. We stress that this paper is based on the patent files submitted in Brazil (INPI: BR 10 2022 000128 6) and USA (USPTO: 17/689,365).

In summary, our work brings the following contributions:

- By employing native functions of the Android operating system to obtain features from an APK file, our solution (MADS) executes in a lightweight fashion and does not infringe licenses and terms of use of applications. In contrast, state-of-the-art solutions extract features directly from the APK file, which is not efficient and infringes licenses and terms of use of applications, since this requires the reverse engineering, disassembly, or decompilation of the APK.
- Previous proposals generally consist of a single analysis step, based on a single or ensemble of classifiers, which aims at detecting several *malware* types. In contrast, MADS is based on specific-type detectors with stronger capacity to learn the specificities of each *malware* type or *malware* with particular behavior (e.g., those abusing accessibility features to harm users or those that encrypt data). Besides a MADS detector comprises a multi-stage analysis, wherein the first analysis step uses a set of rules to filter only applications of interest. These design choices of MADS contribute to reduce the number of misclassified applications and avoid unnecessarily analyzing all applications in a deeper fashion with ML-based classifiers.
- We collected a dataset of *benign* and malicious Android applications. After careful processing and analysis of all samples, we created sets with particular *malware* types, i.e., *banking*, *ransomware*, and *phone scam*. These specially created sets were used to train and test MADS, since public datasets were found to be outdated, unbalanced, having duplicated samples (at a feature vector level), and in some cases, with wrong labels.
- We implemented a prototype of both MADS and the state-of-the-art solutions' most common detection approach, and tested them, in a *Samsung Galaxy S21+* device, using the aforementioned dataset. The results show that MADS can reduce nine times the number of false positive cases, and ten times the number of false negative cases, compared to the state-of-the-art approach. In addition, we observed that MADS is three times faster to analyze applications and consumes less CPU while obtaining features. There were scenarios in which MADS was even ten times more efficient than the state-of-the-art approach in terms of CPU consumption.

The remainder of this paper is organized as follows. Section II presents related work on Android *malware* detection. Next, section III describes MADS, while section IV presents the evaluation performed to compare MADS with the state-of-the-art's most common approach to detect Android *malware*. Section V describes limitations of this work. Finally, section VI concludes this paper and points out future work.

II. RELATED WORK

In this section, we present related work developed to detect Android *malware* using ML-based techniques. In general, existing proposals rely on features obtained through static and/or dynamic analysis of APK files. The analysis is performed on-device or off-device (i.e., using a cloud service to make the processing) to classify applications as *benign* or *malware*. Next, we briefly describe the proposals and point out their limitations, which are addressed by MADS. Table 1 summarizes and compares the main characteristics of related work and MADS.

Salah et al. [10] present an approach based on static analysis to detect Android *malware* using ML and a technique to reduce the number of features used by the ML model. The approach obtains features from the *Manifest* file and from the disassembled DEX code of applications, including features related to *permissions*, application components, *intents*, API calls, and URL strings. A similar approach is presented by Fereidooni et al. [8]. The authors employ an ML-based approach to classify Android *malware* according to their families, using static features such as *intents*, *permissions*, system commands, suspicious API calls, and others related to malicious activities through a Dalvik bytecode analysis (e.g., reading IMEI, loading native, dynamic, and reflection code, etc.). The problem about these two proposals is related to the invasive feature obtainment process adopted, which requires disassembling the application to obtain the features, violating licenses and terms of use employed by Android applications.

Pektas and Acarman [17] leverage another static approach using API call graphs and deep learning to detect Android *malware*. Their idea is to capture all execution paths in terms of the invoked APIs from an analyzed application and construct an API call graph of each execution path. The graphs are processed and transformed into features given as input to a deep neural network for classification. Gao et al. [18] propose a similar idea by reconstructing the connections between applications and APIs, as well as between APIs and APIs, and giving them as input to a graph convolutional network model. Although adopting API calls for detection of Android *malware* is an interesting alternative, the detection solutions of these papers infringe terms of use and licenses of several applications by collecting the invoked APIs through source code analysis. In addition, these methods also suffer from the inefficient process of obtaining the features.

By adopting an approach based on dynamic analysis, Sanz et al. [19] rely on features extracted from the header of network packets sent and received by the analyzed application. Using these features, the solution employs two ML models (*AdaBoost* and *Random Forest*) to analyze network behaviors of suspicious applications. Sihag et al. [12] use network traffic for detection as well. To classify an application, they log its corresponding traffic and represent it as gray-scale images given as input to a neural network model. Relying on dynamic analysis based on network data is not efficient and may represent a risk, since the classification of

TABLE 1. Comparison between the characteristics of related work based on ML and MADS.

Proposal	Invasive feature obtainment	Analysis type	Main features used for classification	Analysis stage (before/after install)	Lightweight solution	Detectors type
Salah et al. [10]	Yes	Static	Permissions, intents, API calls, URL strings, app components	Before	No	General
Fereidooni et al. [8]	Yes	Static	Permissions, intents, systems commands, API calls, bytecode analysis	Before	No	General
Pektas and Acarman [17]	Yes	Static	API calls	Before	No	General
Gao et al. [18]	Yes	Static	API calls	Before	No	General
Sanz et al. [19]	No	Dynamic	Network traffic	After	No	General
Sihag et al. [12]	No	Dynamic	Network traffic	After	No	General
Gharib and Ghorbani [20]	Yes	Hybid	Strings with specific content, API calls, permissions	Before and after	No	Specific (ransomware)
Alzaylaee et al. [13]	Yes	Hybid	API calls, intents	After	No	General
Pierazzi et al. [11]	Yes	Hybid	Certificate hash, permissions, app components, network traffic, SMS sent, read/write operations	After	Yes*	Specific (spyware)
Mahindru and Sangal [21]	Yes	Static	Permissions, API calls, app ratings, number of downloads	Before	Yes*	General
Kim et al. [22]	Yes	Static	API calls	Before	No	General
Jerbi et al. [23]	Yes	Static	API calls	Before	No	General
Atacak [24]	Yes	Static	Permissions, intents, activities	Before	No	General
Wu et al. [25]	Yes	Static	API calls	Before	No	General
Sahin et al. [26]	Yes	Static	Permissions	Before	No	General
MADS	No	Static	Permissions, intents, hardware components, app components	Before and after	Yes	Specific

* The process is performed in an external server; therefore, device energy is not consumed.

an APK file may not occur on time for preventing malicious actions (e.g., from a *ransomware* application).

Other authors rely on the combination of static and dynamic analysis. Gharib and Ghorbani [20] propose a solution for the detection of Android *ransomware*, called DNA-Droid, using ML techniques. The solution first analyzes an application employing static features related to strings (having content regarding encryption, locking, threats, pornography, or money), images (logos), API calls, and *permissions*. It classifies an application as *benign*, *suspicious*, or *malware*. In case the application is marked as *benign* or *malware*, the analysis is finished. For suspicious applications, the solution uses dynamic features (API call sequences) to detect malicious behavior. The static features used by DNA-Droid requires disassembling the APK file, which is prohibited by licenses and terms of use of Android applications. Besides, it also suffers from the limitations regarding efficiency of using dynamic analysis.

Another proposal that shows the benefits of combining both static and dynamic analysis is due to Alzaylaee et al. [13], who use a deep learning system to detect Android *malware*. The authors first employ dynamic features (i.e., API calls and *intents*) to evaluate a deep learning-based neural network model. After this, they combine the dynamic features with static features to create another model and compare it with the previously evaluated model. Their results show that the second model outperformed the first one. By relying on dynamic analysis, their proposal is not suitable for practical scenarios, especially to analyze several applications at the same time. Moreover, it is limited for detecting malicious applications only after their installation.

Pierazzi et al. [11] propose an off-device detection approach based on static and dynamic features obtained by running Android applications on *Koodous* platform [27]. Multiple classifiers are used and their results are combined by an Ensemble Late Fusion algorithm. The static features employed are the hash of the certificate used to sign the application, the set of used *permissions*, and the number of some application components. The dynamic features correspond to read and write operations, started background processes, load of DEX files, cryptographic operations, outgoing and incoming network activity, and SMS messages sent.

Another off-device approach is proposed by Mahindru and Sangal [21], who design a web-based Android *malware* detection framework. In order to analyze an application, its APK file is sent to the web-based system, which executes the application to extract *permissions* and API calls. The system also collects ratings and number of user downloads related to the application. The obtained features are evaluated by using models constructed with different ML techniques (e.g., deep learning and farthest first clustering). The problems of both off-device proposals are related to the implementation on real world scenarios. They require the analyzed applications to be sent to external servers in order to be executed and analyzed. This is inefficient and may require users to wait long time periods for the analysis result, leading to negative user experience.

Kim et al. [22] propose the use of convolutional neural network (CNN) to find common features of API call graphs of Android *malware*. When classifying an application, the system extracts the graph of the application and compares it with the graphs created with the CNN in the training phase. This comparison indicates whether the application is malicious or not. A similar idea is proposed by Wu et al. [25], who also leverage API call graphs for detection. Jerbi et al. [23] employ API calls to create rules to detect Android *malware* through a bi-level optimization problem, wherein the upper level designs a set of effective *malware* detection rules, and the lower level generates a set of artificial patterns of each rule with the aim of improving detection performance. By using API calls, these proposals not only are inefficient, since they require performing reverse engineering, but also infringe licenses and terms of use of applications.

Atacak [24] introduces a detection system comprising a fuzzy inference approach that combines and interprets the output scores of several ML models created with different algorithms. The use of multiple models is an approach due to Sahin et al. [26] as well, who propose a set of linear regression classifiers. In addition to employing several models in the classification task, which is not efficient, the authors of both works propose the decompilation of APK files to analyze them, infringing applications' licenses and terms of use.

To overcome the limitations of the literature, we propose in this paper MADS. By obtaining features from the APK file via the operating system, MADS is more efficient and does not infringe licenses and terms of use of Android applications. Besides, while related work is based on a single ML detector and one stage only, our detection solution is based on a lightweight and multi-stage approach using specific-type detectors, aiming to reduce the number of misclassified applications. MADS is able to analyze applications before their installation in an on-device fashion, and does not require (although the last analysis step can be performed right after the installation or may involve) the execution of the application. This prevents some applications (e.g., *ransomware*) from performing their malicious actions, which can be dangerous and too late if an analysis is performed during or after their execution.

III. PROPOSED SOLUTION

A. OVERVIEW

In this section, we present *Malware APK Detection Solution* (MADS), a lightweight, on-device, and multi-stage approach to detect *malware* of specific types created to run on Android devices, without the need of internet connection. The aim of MADS is to efficiently and accurately analyze Android applications without infringing their licenses or terms of use that forbid reverse engineering, decompilation, or disassembly of the APK file. More specifically, MADS consists of one or more detectors, wherein each detector is carefully designed to identify a single *malware* type (e.g., *ransomware*, *banking*) using APK features obtained from the operating system in a non-invasive fashion.

For the sake of simplicity, here we will first present MADS having a single detector employed on it; later, we will cover a use case of MADS with multiple detectors working together. Figure 1 depicts, in the offline steps of MADS (leftmost part), an overview of how to build a detector to be employed in MADS. The first step consists of creating a set of rules to cover the indispensable characteristics and behaviors that an application must present to be considered as a potential *malware* of a certain type of interest. These rules will compose the *lightweight* analysis module of a detector. Next, a dataset of APK files is created, comprising samples of the *malware* type of interest and *benign* samples. This dataset is employed to train ML models using only features that can be obtained employing Android native functions (which does not infringe licenses and terms of use of applications). These models will be components of the *deep* analysis module of a detector.

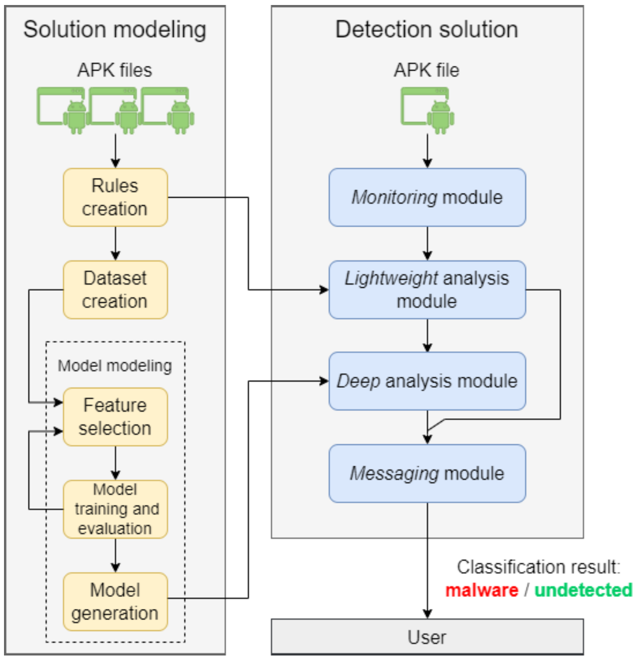


FIGURE 1. Design overview of MADS (with a single detector).

By having at least one detector modeled, MADS can be deployed on an Android device, as shown in Figure 1 (rightmost part). During its execution, MADS *monitoring* module monitors the device for the presence of APK files. Upon identifying a file, MADS employs, in sequence, its two analysis modules to analyze the application for malicious traces. Each module obtains a specific set of features from the APK file, using Android native functions, and outputs a classification result for the application. First, the *lightweight* analysis module is executed to check whether the application is a potential *malware* of interest, by comparing the APK features with the *lightweight* rules. If no potential *malware* behavior is identified, an *undetected* label is returned (meaning that non-malicious behavior was found). Otherwise, the analysis goes on; i.e., the *deep* analysis module classifies the application, based on ML models, and outputs a classification result (*malware* or *undetected*) for the analyzed application. The result of the analyses is forwarded to the *messaging* module that finally presents it for the user.

It is important to highlight that while the *lightweight* analysis module is always executed, the *deep* analysis module runs depending on the outcome of its preceding module, which can save resources and processing time, as we will show in section IV-E. In the rest of this section, we first present in more detail how a MADS detector is designed. Next, we describe MADS workflow in detail while executing in an Android device with multiple detectors.

B. RULES CREATION

While designing a MADS detector, the first step consists of creating a set of rules (which will compose the *lightweight* analysis module) to map the characteristics of the *malware* type or specific behavior of interest for the detector. Here, we focus on identifying, by the use of rules, potential *malware* with specific behaviors related to certain attack mechanisms (e.g., encryption of users data). Each detector contains a different set of rules according to the *malware* type it aims at identifying. In essence, the rules should encompass lightweight features that can be obtained before the application installation using Android native functions. An example of rule could be, for instance, the presence of certain Android *permissions* used by the application that are indispensable for a specific type of *malware* to perform its malicious behaviors.

To conceive the rules, for instance, based on only *permissions*, the *malware* type of interest must be studied to understand which *permissions* (and their combinations) it requires to perform a malicious action. The rules designer may opt to create a preliminary dataset with samples of the *malware* type of interest to analyze real samples in order to obtain insights about the characteristics and behaviors of the *malware* type. These insights may help the designer to learn which features can be obtained and used for analysis before the APK file installation.

Suppose the threat of interest of a detector is *phone scam* [28], whose main activity is to intercept phone calls and redirect them to attackers. One can create rules that include this type of activity, which will be applied during an APK file analysis. One example of *permission* that can be used is *PROCESS_OUTGOING_CALLS* that allows applications to block and redirect phone calls. This way, an application having this *permission* is a possible candidate to be *phone scam*. Other types of *permissions* (and their combinations), or even other elements that can be obtained from the APK files in a lightweight fashion, can compose the rule set of the *lightweight* step of the *phone scam* detector.

C. DATASET CREATION

Another step of the design of a MADS detector consists of creating a dataset of *malware* and *benign* APK files that should be used to construct the ML models (components of the *deep* analysis module). The dataset elaboration must take into account the specificities introduced by the MADS analyses modules, as follows.

As introduced before, the *lightweight analysis* of a MADS detector identifies whether an application is a potential *malware* based on a set of rules encompassing characteristics of the *malware* type of interest. As we will discuss in more detail later, the *lightweight* analysis is the first step of a detector and the *deep* analysis only executes if the *lightweight* module flags the application as potential *malware*. Therefore, the *deep* analysis module will only analyze applications having the characteristics defined in the *lightweight* analysis.

For the aforementioned reasons, the dataset employed to design the ML models must be elaborated to comprise only APK files with the characteristics covered by the *lightweight* rules defined for a particular detector. Suppose again we are building a detector for *phone scam malware*. Both the *malware* and *benign* samples collected to model a detector of this threat must encompass phone call activities, since the main activity of such *malware* type is to intercept phone calls. This dataset creation requirement guarantees that the ML models can be built to better distinguish between *malware* and *benign* applications that present the characteristics of interest. In section IV, we show the impact of this design choice, which helps reduce the number of false alarms triggered by ML-based detectors.

With the dataset established, the corresponding APK files are given as input to a feature extraction tool that employs Android native functions to obtain features from the applications. These features must represent characteristics of the applications that can be obtained before and/or after the application installation on an Android device. The feature obtainment process of MADS does not infringe licenses and terms of use of applications since all the features are obtained from information provided by the Android operating system. Besides, the features should undergo through appropriate preprocessing steps according to the dataset designer expectations about a suitable processed dataset. For example, missing feature values should be handled, duplicate samples based on feature values may be removed, and samples with outliers on specific feature values may also be dropped. In section IV, we also show that using a restricted set of features, compared to state-of-the-art solutions that adopts a broader set of features (employing API calls, for instance), does not reduce the detection capabilities of our detectors.

D. MODEL MODELING

In addition to lightweight rules, a MADS detector is also comprised of ML models that make part of the *deep* analysis module. This module comprises two steps, wherein each step is based on an ML model. Except for the set of features obtained from the APK files, the ML models of both *deep* steps are generated according to the same workflow, as presented in Figure 1 (leftmost part) and explained as follows.

Recall that, at this point, a dataset of *benign* and *malware* APK files has been already created and properly preprocessed, resulting in a set of samples, wherein each sample comprises two sets of features obtained via Android native functions. One set consists of static features that can be obtained before the APK installation and is used to build a first ML model (component of the first *deep* analysis step). One can employ different categories of features for this model, including those related to *permissions* [29], *hardware components* [30], and *application components* [31]. Common types of Android *application components* are *activities*, *services*, *receivers*, and *providers*.

The other set of features can be obtained after the APK installation and is employed to construct a second ML model, which will compose the second *deep* analysis step. This set of features can be obtained from static (e.g., features related to *intents* [32], *assets*, etc.) and/or dynamic analysis (e.g., features related to behaviors monitored during the application execution, such as *system calls* performed). Notice that this ML model may use all the features that can be extracted from the APK file using Android native functions, including those employed to build the first model. Thus, the second model is intended to present stronger analysis capacity compared to that of the first model, since the second model may employ a broader set of features.

After obtained, the features are given as input to a feature selection technique to filter out those that are not informative to the classification task, as a form to decrease the complexity of the ML model by using a smaller number of features. There are many techniques to this end (e.g., *Recursive Feature Elimination* [33], *SelectFromModel* [34], among others) and a combination of them is also a suitable alternative. The same or a different approach of feature selection used to build the first model can be applied to construct the second model. Next, the selected features are provided to an ML training algorithm such that a model can be trained. The outputted model is evaluated and, if it has suitable classification results (which depend on the designer's criteria), the corresponding model data is generated and can be deployed on the Android device. Otherwise, feature selection and model training and evaluation are repeated until a suitable model is found.

For the two generated models, the *confidence level* (*CL*) of the models regarding the classification labels given to the dataset samples should be analyzed in order to establish a threshold *CL* value. This threshold will indicate when a *malware* label outputted by the model should be accepted as the final label for an analyzed application. For instance, in an ML approach that employs an ensemble of classifiers of the same type (e.g., Random Forest), the *CL* value may consist of the proportion of classifiers that labeled the analyzed APK file with the *malware* label. In case the proportion is higher than the chosen threshold, the *malware* label is accepted; otherwise, the label *undetected* is returned instead.

E. MADS DETECTORS WORKFLOW

After properly designed, a MADS detector can be deployed on an Android device. MADS can work with one or more detectors, wherein each detector aims at detecting applications of a particular *malware* type or behavior. The detailed workflow of MADS with multiple detectors is illustrated in Figure 2. First, the user invokes the installation of an application on her/his device. MADS identifies an application installation event and starts the analysis of the application before it is installed.

The application is given as input for the MADS detectors deployed on the device. Although we present here a scheme where the detectors are disposed in a sequential order, operating one after the other, for efficiency reasons (as explained later), we emphasize that all the detectors can also be used in parallel to analyze a given application.

Considering that the detectors are executed sequentially, the application under analysis is first given as input for *Detector 1*. Employing Android native functions, *Detector 1* obtains a first set of features related to the rules created for the *lightweight* analysis of the type of *malware* of interest. Next, the detector performs the *lightweight* analysis of the APK file and verifies whether the rules of the corresponding *malware* type of interest apply to the analyzed application. In a positive case, the application is forwarded to the *deep* analysis module of *Detector 1* to continue the analysis. However, in case the *lightweight* analysis rules of *Detector 1* does not apply, the APK is forwarded to the next detector (i.e., *Detector 2*), which will repeat the same process, but using its specific set of rules and components, comparing to a possible different set of features obtained from the APK. This analysis process is performed by all MADS detectors available. If no rule of any of the *lightweight* analysis from all *N* detectors apply to the APK, then this application is labeled as *undetected* and the analysis ends, as no malicious traces of interest were found, and the APK can be immediately installed on the device.

Notice that the *lightweight* analysis does not always provide a final classification for the analyzed application. Instead, it indicates whether the application has characteristics of a known type of *malware* of interest and it is the *deep* analysis that can finish an analysis procedure by providing a final result. Recall that this second analysis module is composed of two ML models, wherein each model makes part of one step of the module execution, i.e., *deep* analysis before installation and *deep* analysis after installation. The steps are executed as explained next.

Suppose the rules of the *lightweight* analysis of *Detector 1* apply to the APK file. In this case, the detector's *deep* analysis module obtains a second set of features before the APK installation using Android native functions. With this new set of features, *Detector 1* executes the first ML model (from the *deep* analysis before installation). The model outputs a classification label for the analyzed APK file and a *CL* value

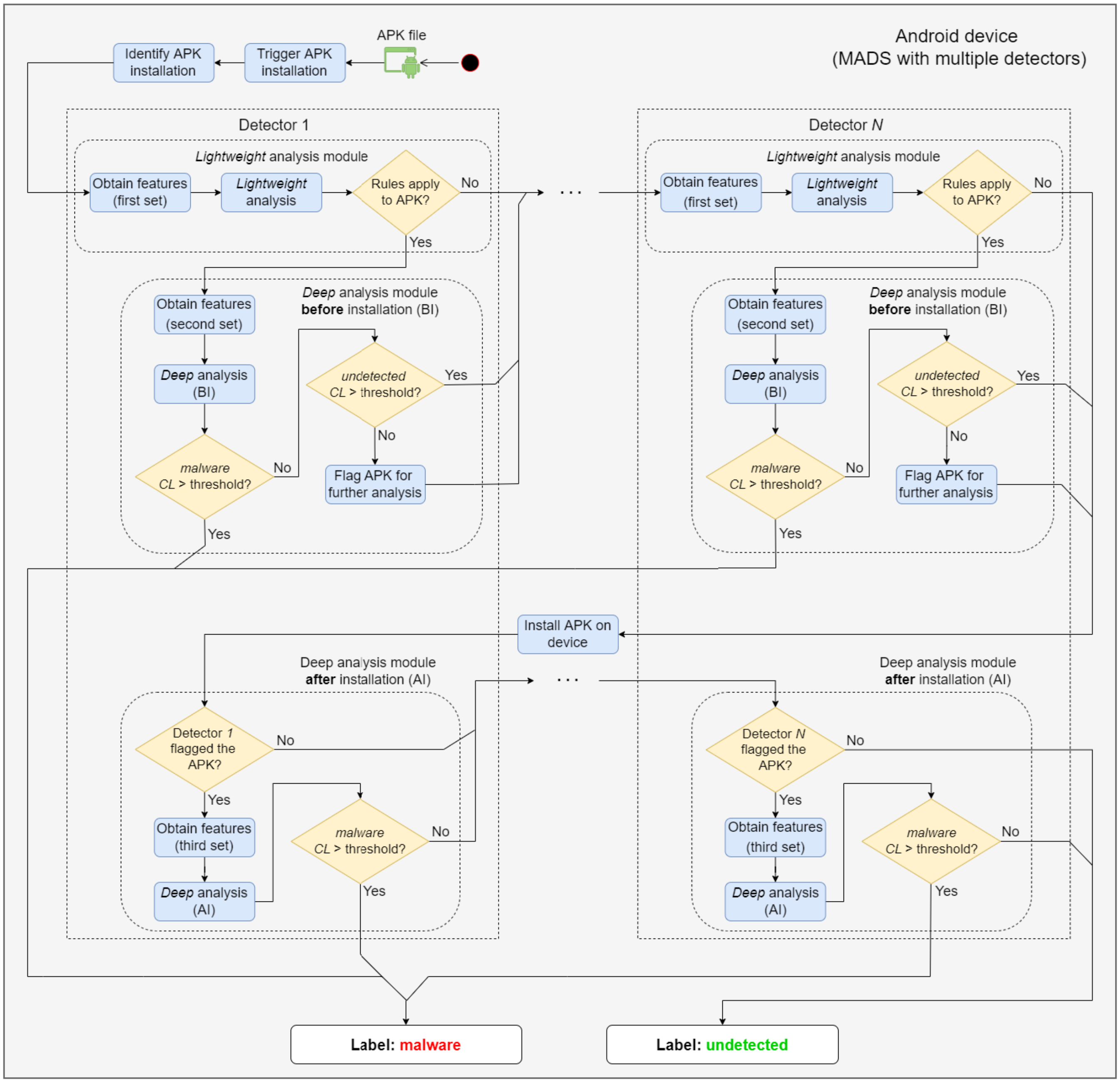

Android device
(MADS with multiple detectors)
APK file
Trigger APK installation
Identify APK installation
Detector 1
Lightweight analysis module
Obtain features (first set)
Lightweight analysis
Rules apply to APK?
No
Yes
Deep analysis module before installation (BI)
Obtain features (second set)
Deep analysis (BI)
malware CL > threshold?
undetected CL > threshold?
Flag APK for further analysis
Detector N
Install APK on device
Deep analysis module after installation (AI)
Detector 1 flagged the APK?
Obtain features (third set)
Deep analysis (AI)
malware CL > threshold?
Detector N flagged the APK?
Label: malware
Label: undetected

FIGURE 2. Workflow of the Android *malware* detection solution proposed.

indicating how confident the model is about the outputted label. If the *CL* value is above the threshold predefined for the first model, then the outputted label is accepted. Moreover, in case the classification of *Detector 1* is accepted and the result is the *malware* class, the analysis is finished right away and the label returned to the user, alarming him about the danger of continuing with the installation process of this particular APK file. Notice that in such a case, the execution of the other detectors available in MADS will not be necessary, since a malicious behavior was already found. On the other hand, if the label returned is *undetected* with the classification being accepted due to a sufficient *CL* value, *Detector 1* does not execute the next step of its *deep* analysis module (after installation) and finishes its analysis procedure.

When the *CL* is below the predefined threshold for both classes, the classifier is not confident about the classification for the APK file, thus the application is flagged to be analyzed later by the *deep* analysis after installation of that particular detector that returned this result. In this case, the following step is to forward the APK to the next MADS detector to continue the analysis procedure and so on. If all detectors that perform the *deep* analysis before installation output no confidence enough about a *malware* label, the APK can be installed in the device. The next analysis, also part of the *deep* analysis module, is executed in a different moment (after the application installation), using a different set of features obtained about the application after its installation, as described in the following.

Suppose *Detector 1* was not confident enough about its outputted classification label during the *deep* analysis before installation. In such a case, *Detector 1* collects a third set of features that can only be obtained after the APK installation using Android native functions. The third set of features is given as input to a second ML model, from the *deep* analysis after installation, which performs the last application analysis step. Similar to the first model, the second one outputs a classification label and a *CL* value. If a *malware* label is outputted and the *CL* is above a threshold predefined for the second model, then the outputted label is accepted and the analysis provides the respective result to the user. Otherwise, the subsequent detectors will proceed with the analysis by executing their *deep* analysis after installation in case they flagged the APK in their previous analysis step. If no detectors output a *malware* label with *CL* above the predefined thresholds, the analysis has been completed with no concrete confidence about the application label, thus the APK file is finally classified as *undetected*. Alternatively, one can ignore the *CL* returned by the second model and accept its result as final. We emphasize that when one of the detectors flag the APK for further analysis and there is still other detectors to evaluate the APK, in case a *malware* label with high confidence is found, the APK file is flagged as such and no further analysis is necessary anymore.

We use two threshold values for accepting a classification returned by the ML model, one for the *malware* class and another for the *undetected* one. These thresholds are specific for each detector and may impact the results of the solutions regarding *false positive* and *false negative* results. Their choice is a trade-off between these metrics, and for that reason, must be chosen based on a study of the detectors classification results and users' needs (which will cause less impact on users: a false alarm or being infected by a *malware*?).

We highlight that some of the steps presented in Figure 2 are redundant and can be easily performed just once. For instance, the feature obtainment process can be performed one time and used by all detectors in the later steps. We present the figure with such redundant steps for the sake of understanding and to highlight the specificities of each detector. During the prototype creation (detailed in the next section), we will optimize the steps to avoid redundant work.

IV. EVALUATION

In order to assess the benefits of our proposal, we developed two Android applications: one representing MADS and having three different detectors working together, one after the other and based on different threats of interest (e.g., *banking*, *ransomware*, and *phone scam*), and another representing a general approach used by most state-of-the-art solutions (referred as *general solution*), using only a single model to detect all of the same threats addressed by the MADS application.

For the *general solution*, we used the *APKParser* tool [35] for feature extraction, since it obtains features using the state-of-the-art's most common approach, i.e., performing the reverse engineering, disassembly, and decompilation of the APK files. In turn, for MADS, we used Android native functions [16], our proposed alternative. We optimized the feature extraction process of MADS (based on Figure 2) to extract the features from the APK file only once and then use these features during the whole classification cycle, feeding each detector with the features needed for the classification task of each one of them.

We tested MADS and the *general solution* regarding detection capabilities and resources consumption, using a dataset we designed composed of 2,944 APK samples (736 for each class: *benign*, *ransomware*, *banking*, and *phone scam*). This dataset was used to train the ML models used by the two solutions. In this section, we present details about our evaluation and results. We first describe the use case scenario considered to perform the evaluation and draw research questions we aim at answering. Next, we detail how the two solutions were designed. Finally, we present the tests performed and evaluation results.

A. USE CASE AND RESEARCH QUESTIONS

To measure the effects of MADS in a practical scenario, we consider a use case of detection of three types of Android *malware*, as follows:

- *Phone scam*: this threat consists of malicious applications that can intercept users' outgoing calls and forward to cybercriminals, who will be able to perform voice phishing attacks (e.g., to gain access to personal and financial information from users).
- *Banking*: *malware* that aims to steal banking credentials of users, at rest in the device or given as input by users, and send to attackers for financial gain.
- *Ransomware*: type of *malware* that demands a sum of money from victims while promising to "release" a hijacked resource of the device in exchange.

Considering this use case scenario, we created a proof-of-concept application of MADS. This solution is composed of one detector for each *malware* type of interest (with the behaviors described above), wherein each detector comprises two analysis steps (*lightweight* and *deep* analysis before installation). Notice that we did not take into account the second model, from the *deep* analysis after installation, in our evaluation. The reason is threefold. First, the goal of the evaluation is to draw a fair comparison between MADS and the *general solution*, which usually employs a single ML model. Second, the use of the *CL* threshold of the first model may vary according to criteria of the solution designer and the first model results on the *testing*

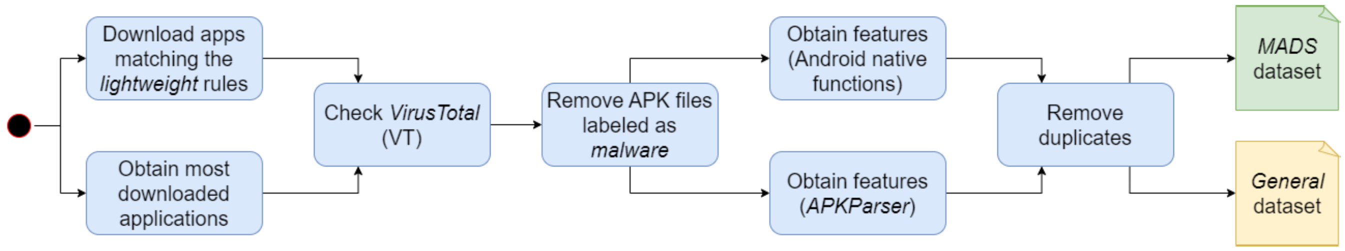

Download apps matching the *lightweight* rules
Obtain most downloaded applications
Check *VirusTotal* (VT)
Remove APK files labeled as *malware*
Obtain features (Android native functions)
Obtain features (*APKParser*)
Remove duplicates
MADS dataset
General dataset

FIGURE 3. Workflow adopted to collect and process the *benign* samples used in our evaluation.

TABLE 2. Examples of rules created for our evaluation.

Malware type	Rule based on the permission the app requests	Potential malicious capability
Phone scam	*PROCESS_OUTGOING_CALLS*	The app can redirect telephone calls to attackers.
Phone scam	*BIND_INCALL_SERVICE*	The app intends to become the default telephone call app, controlling aspects of phone calls, such as call redirection.
Banking	*BIND_ACCESSIBILITY_SERVICE*	The app can control aspects of the device, being able to log users' inputs and monitor the screen for banking credentials.
Banking	*SYSTEM_ALERT_WINDOW*	The app can display a fake window, mimicking the screen of a legitimate banking app to obtain credentials typed by users.
Ransomware	*DISABLE_KEYGUARD*	The app can change the PIN code of the system, blocking the use of the device by the user.
Ransomware	*WRITE_EXTERNAL_STORAGE*	The app can access data of the device storage, being able to encrypt them.

dataset. Third, using the *deep* analysis after installation would only improve the results obtained with the first *deep* analysis.

Different from MADS, state-of-the-art solutions usually adopt a single analysis step comprising an ML model to detect several types of *malware* with different behaviors. Thus, we designed a single ML model to represent this general detection approach. We emphasize that we decided to create such solution to compare results, since many works proposed in the literature do not make their solutions available for tests. Besides, the datasets used for the experiments are not shared with the community in the majority of the cases, which makes the process of replicating their experiments impossible. To overcome this limitation, some authors use publicly available datasets, such as Drebin [7], but this set contains outdated applications from 2010 to 2012, which do not correspond to the characteristics of applications seen nowadays.

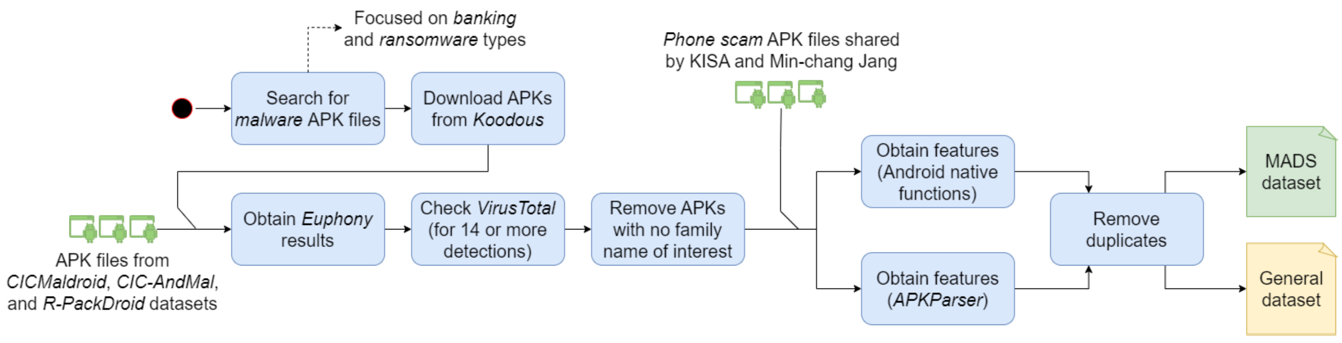
By testing MADS and the *general solution* to detect *phone scam*, *banking*, and *ransomware*, our evaluation aims at answering three research questions:

(i) By using specific-type detectors in a multi-stage fashion, can we improve the classification capabilities of ML-based solutions in relation to general classifiers?
(ii) Even though MADS uses additional steps during classification, can we have a more efficient analysis process, requiring less time than the *general solution*?
(iii) Can we perform a more efficient analysis process, consuming less resources (CPU and RAM) than the *general solution* during Android application analysis?

B. RULES AND DATASETS CREATION

The first design step of MADS is the creation of a set of rules for the *lightweight* analysis module of each detector (refer to section III-B). We studied papers and technical reports of security specialists and analyzed many applications to understand the well-known and main malicious behaviors of *phone scam*, *banking* and *ransomware* applications. Based on our studies, we created a set of rules to identify possible applications of each threat. Table 2 provides examples of rules we created for our evaluation.

The second design step of MADS is the creation of a dataset of APK files (refer to section III-C) used to build the models of the *deep* analysis. To create a model for each of the three detectors, MADS requires a specific dataset having *malware* applications for the threat type or behavior of interest, and *benign* applications with similar characteristics to the malicious ones, i.e., the *benign* applications must match the rules of the *lightweight* analysis for the specific threat. In contrast, the *general solution* tries to detect the three types of *malware* with a single detector and, therefore, requires a dataset of *benign* applications with as many characteristics as possible, and *malware* applications of the three types. In view of the specificities of each solution, we created two



Focused on *banking* and *ransomware* types
Search for *malware* APK files
Download APKs from *Koodous*
APK files from *CICMaldroid*, *CIC-AndMal*, and *R-PackDroid* datasets
Obtain *Euphony* results
Check *VirusTotal* (for 14 or more detections)
Remove APKs with no family name of interest
Phone scam APK files shared by KISA and Min-chang Jang
Obtain features (Android native functions)
Obtain features (*APKParser*)
Remove duplicates
MADS dataset
General dataset

FIGURE 4. Workflow adopted to collect and process the *malware* samples used in our evaluation.

datasets composed of *benign* and *malware* Android applications. The first one, named MADS dataset (*MD*), was used to design the detectors that embody the idea of the proposed solution. The second one, named general dataset (*GD*), was used to model the detector based on the *general solution* approach.

We collected and processed *benign* applications according to the workflow presented in Figure 3. First, we crawled the *Google Play Store* and downloaded samples that match the rules of the *lightweight* analysis modules of MADS to compose *MD*. In turn, most solutions of the state-of-the-art use *benign* datasets having the most downloaded applications from the *Play Store*, since it is believed that such a dataset ensures the variability required by a general model to perform accurate classification of different threats having a diverse range of behaviors. For this reason, to collect samples for *GD*, we used the *AppBrain* [36] platform to search for the package name of the applications with the highest number of downloads in several countries (e.g., Brazil, USA, India, France, etc.) and downloaded the corresponding applications from the *Play Store*. After this, we uploaded the downloaded APK files to *VirusTotal* (VT) [37], a popular platform capable of analyzing the files using several antivirus (AV) engines, wherein each engine outputs a label indicating whether the file is malicious or *benign*. To guarantee *benign* datasets without malicious traces in our evaluation, we removed every file to which at least one engine outputted a malicious label.

Afterwards, we obtained features about the remaining *benign* APK files. As discussed in detail later, we employed a single process and the same input features to generate the ML models of both MADS and the *general solution*'s approach. Besides, since only the *deep* analysis before installation is considered for our solution in the experiments reported in this paper, the use case presented here encompasses only static features that can be obtained before the APK installation. As MADS obtains features from the Android operating system, we adopted Android native functions to obtain features about the APK files of *MD*. In turn, the *general solution*'s approach obtains features directly from APK files. Thus, we used *APKParser* to obtain features from applications of *GD*.

We obtained more than 200 features from the *benign* APK files and then removed duplicates based on their feature values. Table 3 presents examples of features employed in our evaluation, which can be obtained before APKs installation. *Permissions* are used as binary categorical features, wherein possible values are 1 for *permissions* the application requests, and 0 for *permissions* the application does not request. Examples are *READ_SMS*, *CALL_PHONE* and *RECEIVE_SMS*. The same applies for *hardware components*, which are binary categorical features indicating whether the application requests access to a hardware component of the device. Example features are *hardware_camera*, *hardware_microphone* and *hardware_location*. In turn, for *app components*, we count the number of specific components the application contains. Example features are *activities_count*, *services_count* and *providers_count*. Table 4 presents an example of *benign* sample with the values of some obtained features.

TABLE 3. Examples of APK features used in our evaluation.

Feature type	Feature category	Quantity	Examples
Permissions	Categorical	166	*READ_SMS* *CALL_PHONE* *RECEIVE_SMS*
Hardware components	Categorical	60	*hardware_camera* *hardware_microphone* *hardware_location*
App components	Continuous	4	*activities_count* *services_count* *providers_count*

TABLE 4. Examples of dataset samples employed in our evaluation with the values of some of the obtained features.

label	*READ_SMS*	*CALL_PHONE*	*hardware_camera*	*activities_count*	...
malware	1	1	0	7	...
benign	1	0	1	13	...

Malware applications were collected and processed by following the workflow depicted in Figure 4. *Phone scam* applications were shared with us by the *Korea Internet Security Agency* (KISA) and the security researcher Min-chang Jang, who has reported the collection of samples in a previous work [28]. In turn, *banking* and *ransomware* samples were obtained from the following datasets: *CIC-AndMal2017* [38],

TABLE 5. Datasets used in our evaluation.

Dataset	Number of training samples				Number of testing samples			
	Benign	*Phone scam*	*Banking*	*Ransomware*	*Benign*	*Phone scam*	*Banking*	*Ransomware*
MD	736 (specific)	736	736	736	316 (specific) / 316 (general)	316	316	316
GD	736 (general)							

CICMalDroid 2020 [39] and *R-PackDroid* [40]. In addition, we searched for the hash of other samples in security reports (e.g., available on *Malpedia* [41]) and downloaded the corresponding APK files from *Koodous*, a popular web repository of APK files. Next, in order to ensure the obtained APK files are indeed *banking* or *ransomware* applications, we used analysis data from VT and *Euphony*, a tool that receives as input the AV engines' labels for an APK file and gives as output a family name to which the APK belongs [42]. We selected only samples that were labeled as malicious by at least 14 AV engines on VT and were labeled with a family name related to *ransomware* or *banking* by either *Euphony* or at least 4 AV engines. We created a list of families to consult from by researching the literature for known family names of *banking* and *ransomware*. Thereafter, we obtained the same set of features from all the selected *malware* APK files, as done for the *benign* samples, and removed duplicate ones (based on their feature vector). Notice that, for *phone scam*, the step of verifying that an APK file is malware or not, and to which family it belongs to, was not necessary, since a third party already confirmed this. Table 4 presents an example of *malware* sample with the values of some obtained features.

We used the processed APK files to construct the *MD* and *GD* datasets according to their specificities. Each dataset comprised *training* and *testing* sets balanced between the classes (*benign*, *phone scam*, *banking*, and *ransomware*). We chose to split each class of samples into 70% for the *training* set and 30% for the *testing* set. Since we obtained different numbers of samples for each class after the processing step described before, we decided to remove some samples from each set and keep the same number of samples for all sets. To determine the number of samples of each class allocated in each set, we took into account the smaller set of samples between the dataset classes. As *ransomware* was the class with the smaller number of samples obtained (1,052) after the dataset processing, we defined 736 (i.e., 70% of 1,052) as the number of samples the *training* set should contain for a given class and used this same number for the other classes too. In turn, we determined 316 (i.e., 30% of 1,052) as the number of samples for the *testing* set for each class.

Table 5 presents a summary of the datasets. The *MD training* set was comprised of 736 specific *benign* samples that matched the *lightweight* rules of all the three detectors (*phone scam*, *banking*, and *ransomware*). Each specific-type detector used the same 736 *benign* samples to train the ML models, in addition to the same number of *malware* samples of each detector. To perform a fair comparison between MADS and the *general solution*'s approach, *GD* was composed of the same samples of *malware* contained in *MD*, 736 for each threat. However, the *GD training* set differed from *MD* by having 736 *benign* samples from the most downloaded applications of the *Play Store*. With the aim of testing the two solutions with the same set of samples and drawing a fair comparison, both *MD* and *GD* had the same *testing* set, comprising 316 samples of each threat, 316 *benign* samples that matched the *lightweight* rules of all the three detectors, and 316 *benign* samples from the most downloaded ones of the *Play Store*. We stress that there were no duplicate samples among the *training* and *testing* sets of both datasets (*MD* and *GD*).

C. MACHINE LEARNING MODELS MODELING

After creating the datasets, we modeled the ML models of both MADS and the *general solution*'s approach. For MADS, we generated three models, one for each detector. For the *phone scam* detector's model, we used the 736 *benign* and 736 *phone scam* samples of the *MD training* set. To generate the models of the *banking* and *ransomware* detectors, we also adopted the *benign* samples of the *MD training* set, however, using the 736 *banking* samples and 736 *ransomware* applications, respectively. For the general model of the *general solution*'s approach, we employed all the samples from the *GD training* set (736 samples for each class: *benign*, *phone scam*, *banking* and *ransomware*).

To perform a fair comparison between MADS and the *general solution*, we used the more than 200 input features (available before the APK file installation) and a single workflow to generate all the models, employing the *scikit-learn* API [43]. First, we employed a feature selection to filter out features that do not effectively contribute to the classification process. Here, we used a model-based feature selection method named *SelectFromModel*. This method selects the most important features for the model by using importance values assigned by the model to the features according to their importance in the classification task. As ML algorithm, we selected *Random Forest* (with 50 estimators) since it is generally reported to be one of the ML algorithms with the best results in the Android *malware* literature [44], [45].

As a result, the MADS detectors had their models generated with 27, 30, and 31 features for the *ransomware*, *phone scam*, and *banking* models, respectively. In turn, the model for the *general solution*'s approach was modeled with 43 features. We can observe that the approach of our proposal results in the creation of simpler models compared to the *general solution*'s approach. As advantages of simpler models, the classification task can be performed more efficiently and the models can be easily interpreted in order to understand

TABLE 6. Results of testing MADS and the *general solution* on samples from the *testing* sets, wherein FP, TN, FN, and TP mean number of false positives, true negatives, false negatives, and true positives, respectively.

Metric	MADS models			State-of-the-art general model		
	Phone scam	*Banking*	*Ransomware*	*Phone scam*	*Banking*	*Ransomware*
FP / TN / FN / TP	0 / 632 / 2 / 314	1 / 631 / 1 / 315	6 / 626 / 1 / 315	12 / 1,252 / 4 / 312	28 / 1,236 / 18 / 298	24 / 1,240 / 18 / 298
False positive rate (FPR)	0.000%	0.001%	0.009%	0.009%	0.022%	0.019%
False negative rate (FNR)	0.006%	0.003%	0.003%	0.012%	0.057%	0.057%
Balanced accuracy	99.68%	99.76%	99.37%	98.89%	96.04%	96.20%
F1 score	99.68%	99.68%	98.90%	97.50%	92.83%	93.41%

which features are more important while classifying given applications. For our evaluation, all the models were created with a *CL* threshold of 50%, meaning that if a model outputs the *malware* classification label for an application, the label will only be accepted if the corresponding *CL* is above 50%.

D. DETECTION RESULTS

To answer research question (i), the created models were tested on the samples of the *testing* set. Recall this set comprises 316 samples of each threat (*phone scam*, *banking*, and *ransomware*), 316 *benign* samples that matched the *lightweight* rules of all the three MADS detectors and 316 samples from the most downloaded ones of the *Play Store*.

Each model was tested on the samples whose class is addressed by the model. For example, as the *phone scam* detector's model was trained with *benign* and *phone scam* samples, it was tested on samples of the same classes from the *testing* set. Although some rules of each detector may be similar in some specific cases, we argue that the *lightweight* analysis module will perform this filtering process during execution, and in practical scenarios, we expect this behavior. Notice that this choice does not affect the results, since the same *benign* samples will be evaluated by all detectors. In case of *malware* samples, the same occurs; a malicious sample may not be classified as *malware* by a detector specific for a different type of *malware* (if, otherwise, the sample is classified as *malware*, it will only improve the results), but when it reaches its intended detector, it will be properly classified as *malware*. Differently, the model of the *general solution*'s approach was tested on all the samples, since it was trained with all classes of APK files.

We emphasize that among the 316 samples from the most downloaded applications of the *Play Store*, there were samples which did not match the *lightweight* rules of some or any of MADS detectors. As these samples have no malicious traces regarding the threats of interest, they did not even need to be analyzed by the MADS models and, therefore, were immediately classified as *undetected*.

Table 6 presents the classification results obtained after testing MADS specific-type models (*phone scam*, *banking*, and *ransomware*, respectively), and the model of the *general solution*'s approach. The metrics presented are calculated based on number of false positives (FP), true negatives (TN), false negatives (FN), and true positives (TP), as follows:

$$False\ positive\ rate\ (FPR) = \frac{FP}{FP+TN}$$

$$False\ negative\ rate\ (FNR) = \frac{FN}{FN+TP}$$

$$Balanced\ accuracy = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right)$$

$$F1\ score = \frac{2*TP}{2*TP+FP+FN}$$

The results show that MADS specific detectors presented low numbers of false positive cases (seven in total - average FPR rate of 0.003%, considering the three threats) compared to the general model (64 in total - average FPR of 0.017%). This represents nine times less false positive cases than the general model. Moreover, one of the MADS detectors did not misclassify any *benign* sample, while the general detector misclassified several *benign* samples as *phone scam*, *banking*, or *ransomware*. This indicates a strong advantage of our proposal, i.e., decrease on the number of false positive cases. Reducing the number of false alarms is a strongly desirable feature in the literature of *malware* detection, which improves the credibility of the detection solution and causes a positive impact on user's experience.

The MADS detectors also presented better performance than the general solution when classifying *malware* samples. These detectors had only four false negative cases (average FNR of 0.004%, considering the three detectors), which represents ten times less false negatives than the general detector that presented 40 cases (average FNR of 0.042%). *Banking* and *ransomware* are the types of *malware* to which the general model mostly assigned wrong labels. This shows how challenging it is for the *general solution* to predict the true label of *banking* and *ransomware* applications. In contrast, the MADS detectors are able to learn the specificities of each *malware* type. The results also show that MADS presented an average balanced accuracy of 99,60% (considering the three detectors), which is 2.56% better than the general solution's average balanced accuracy (97.04%).

E. PERFORMANCE RESULTS

To answer research questions (ii) and (iii), we evaluated the performance of MADS and the *general solution*'s

TABLE 7. Information on the APK sets used for performance evaluation.

Set	Description	Sample ID	Size (MB)
#1	*Benign* samples that do not match MADS lightweight rules	1	145.9
		2	153.7
#2	*Benign* samples that match MADS *phone scam* detector lightweight rules	3	9.3
		4	5.9
#3	*Benign* samples that match MADS *banking* detector lightweight rules	5	7.4
		6	49.3
#4	*Benign* samples that match MADS *ransomware* detector lightweight rules	7	77.8
		8	80.9
#5	*Benign* samples that match lightweight rules of all MADS detectors	9	31.7
		10	8.9
#6	*Phone scam* samples	11	3.2
		12	6.1
#7	*Banking* samples	13	1.3
		14	4.2
#8	*Ransomware* samples	15	13.9
		16	11.8
#9	*Malware* samples that match lightweight rules of two MADS detectors	17	2.9
		18	2.5

approach regarding the execution time and resources (CPU and memory) consumed to perform classification. To collect performance results, we executed both solutions on a *Samsung Galaxy S21+* device, equipped with the *Qualcomm SM8350 Snapdragon 888* chipset and 8 GB of RAM. All tests related to execution time and resources consumption were performed for one APK file at a time. To collect execution time results, we used the *SystemClock* Android native class [46]. In turn, to collect resources consumption data, we plugged the Samsung device in a laptop with Android Studio installed and, upon running an APK analysis on the device, we observed the Android Studio's *Profiler window* [47], which shows information on CPU and RAM consumed by the solution application.

We selected some *benign* and *malware* samples with different characteristics from the *MD* and *GD* datasets to run the performance evaluation. Our aim was to observe how MADS and the *general solution*'s approach perform in different classification scenarios (e.g., when a sample matches the *lightweight* rules of a MADS detector and when a sample does not). We split the selected samples into sets of two (2) samples according to their characteristics, as shown in Table 7. The reason for using such a small number of samples in each set is due to the complexity of the resources consumption evaluation, which required to be manually performed to not invalidate the results.

Table 8 presents results for the execution time evaluation. Notice that, for MADS, we present the execution time of the feature obtaining procedure (using Android native functions), *lightweight* analysis, and *deep* analysis steps separately. For the execution times of the *lightweight* and *deep* analyses, we considered the corresponding times of all executed detectors. For instance, if the *deep* analysis of the *phone scam* and *banking* detectors were executed for an APK set, then the time shown is for the sum of the execution times of both detectors. Since the *general solution* presents a single analysis step based on a general ML model, Table 8 shows the execution time of its feature obtainment procedure (using *APKParser*) and its analysis procedure.

TABLE 8. Average time to obtain features and perform each analysis step of MADS and *general solution*.

Set	Average time (ms)				
	MADS			*General solution*	
	Obtainment	*Lightweight*	*Deep*	Obtainment	Analysis
#1	19.5 ± 0.5	0.1 ± 0.0	-	116.5 ± 4.5	6.5 ± 0.5
#2	19.5 ± 2.5	0.1 ± 0.0	9.5 ± 1.5	102.0 ± 15.0	7.0 ± 1.0
#3	16.5 ± 4.5	0.1 ± 0.0	9.5 ± 1.5	205.0 ± 77.5	7.0 ± 3.0
#4	19.0 ± 0.0	0.1 ± 0.0	8.0 ± 0.0	82.0 ± 12.0	8.0 ± 1.0
#5	20.0 ± 2.0	0.1 ± 0.0	31.5 ± 0.5	104.5 ± 0.5	4.0 ± 1.0
#6	17.5 ± 5.5	0.1 ± 0.0	8.5 ± 1.5	73.5 ± 1.5	6.0 ± 1.0
#7	16.0 ± 1.0	0.1 ± 0.0	8.0 ± 3.0	72.0 ± 5.0	7.5 ± 0.5
#8	14.5 ± 1.5	0.1 ± 0.0	10.0 ± 1.0	87.5 ± 0.5	7.5 ± 0.5
#9	14.0 ± 1.0	0.1 ± 0.0	23.0 ± 3.0	64.0 ± 8.0	5.0 ± 2.0

In Table 8, we can observe results showing that the feature obtainment procedure of MADS is at least four times faster than the *general solution*'s procedure. There are cases (e.g., for APK set #3) in which MADS feature obtainment is even twelve times faster. The reason is that while the *general*

TABLE 9. Execution time of the solutions to obtain features and analyze all samples from the APK sets.

Set	Total execution time (ms)	
	MADS	*General solution*
#1	39.2	246.0
#2	58.2	218.0
#3	52.2	424.0
#4	54.2	180.0
#5	103.2	217.0
#6	52.2	159.0
#7	48.2	159.0
#8	49.2	190.0
#9	74.2	138.0
	530.8	**1,931.0**

solution directly manipulates the APK file to obtain features, MADS obtains features using information available through Android native functions, having a significant advantage in relation to execution time and also not infringing licenses or terms of use of applications. The *general solution*'s analysis procedure and MADS *deep* analysis present similar execution times when the *deep* analysis of a single MADS detector (i.e., an ML model) is executed. However, in case the *deep* analysis of more than one MADS detector is executed (e.g., for APK sets #5 and #9), the *general solution*'s analysis procedure becomes faster, since more than one MADS model analyzes the application. In scenarios where the application does not match any of the MADS detectors' *lightweight* rules (e.g., for APK set #1), MADS is highly efficient, as only the *lightweight* analysis is executed. Since this analysis consists of a simple comparison between variables, it occurs in a short time period, close to zero, thus we round its time to 0.1 ms. MADS *lightweight* analysis is at least four times faster than the *general solution*'s analysis procedure.

In Table 9, we show that the whole process of our proposal (considering feature obtainment and analysis steps) is three times faster than the *general solution*'s approach (more specifically, 3.64 times), taking into account the analysis of all tested samples. Hence, the results demonstrate the lightweight characteristic of our detection solution with respect to execution time. The results also indicate that, mostly, the size of an APK file does not introduce significant impact on the execution time of MADS analysis (including the feature obtainment procedure), since most features used in the experiments are easy to obtain and do not require processing every element of the APK file (e.g., most features used in our evaluation are available in the *Manifest* file).

In Table 10, we present the results of device resources (CPU and memory) consumed by MADS in comparison to the *general solution*'s approach, considering initialization and idle state of the solutions applications, feature obtainment, and analysis procedures. It is important to highlight that CPU usage is proportional to energy consumption, thus the lower, the better. The results, shown as an average of the peak values obtained during each test, indicate that the CPU usage during initialization, idle state, and analysis procedures is similar between the solutions. However, MADS consumes less CPU while obtaining features, being even ten times more efficient than the *general solution*'s approach when considering the APK set #4. Therefore, MADS is able to perform a lighter application analysis compared to the *general solution*'s approach in terms of CPU consumption. Regarding RAM consumption, the results indicate that, in general, the solutions required similar amounts of memory to execute. Finally, the results show that the APK file size does not introduce significant impact on the amounts of memory consumed by MADS.

V. LIMITATIONS

In this section, we discuss the limitations of our work.

A. USE OF SPECIFIC-TYPE DETECTORS

One may argue that using specific-type detectors may not guarantee optimal detection performance. We argue that although this approach may suffer from scalability issues (it is hard to map every single *malware* type/class/family in the wild), for the *malware* types or behaviors of interest, the detection performance of this approach is superior to that of a general detector that tries to identify many *malware* types and behaviors at once (see section IV-E for more details). Besides, if we focus on *malware* behavior only, the number of possible malicious behaviors (e.g., locking a device, encrypting users' data) is restricted and easier to map.

Our goal by using such specific-type detectors is to allow for focusing on the most dangerous *malware* types or behaviors. Note that we can either have a detector specific to, for example, the *ransomware* type or one focused on detecting applications that perform unauthorized lock of devices. Besides, we showed in this work that by using specific-type detectors, we can also have a better performance regarding the use of device resources.

B. NECESSITY OF EXPERT KNOWLEDGE

Crafting rules is a task that requires expert knowledge on the threats of interest. Although this may seem a limitation, such task is performed by those implementing the specific-type detectors, who already have the required expertise. Besides, this task is executed once and then requires changes only when either the Android architecture changes or a new detector must be created.

C. LACK OF COMPARISON WITH PUBLIC DATASETS

Using public datasets is the preferred alternative to assess and compare ML solutions to understand their pros and cons regarding performance. However, many public *malware* datasets available contain outdated applications that are not possible to use anymore in recent versions of the Android operating system. For instance, the Drebin dataset contains apps from 2010 to 2012, which were built for old Android

TABLE 10. Resources consumption of MADS and the *general solution*'s approach during the classification steps.

Set	Metric	Resources consumption (average peak values) MADS / General solution				
		Initialization	Obtainment	Lightweight	Deep / Analysis	Idle state
#1	CPU (%)	20.0 / 17.0	2.0 / 12.0	1.0 / -	- / 2.5	0.0 / 0.0
	Memory (MB)	166.8 / 167.5	108.7 / 108.0	108.5 / -	- / 108.5	111.7 / 108.8
#2	CPU (%)	19.5 / 19.5	1.0 / 11.0	1.0 / -	1.5 / 3.5	0.0 / 0.0
	Memory (MB)	155.8 / 156.7	107.7 / 109.6	108.3 / -	108.2 / 110.2	110.8 / 110.5
#3	CPU (%)	20.0 / 20.0	2.5 / 14.5	0.1 / -	1.0 / 0.5	0.0 / 0.0
	Memory (MB)	159.6 / 159.3	106.1 / 116.7	106.8 / -	107.0 / 110.3	107.2 / 110.7
#4	CPU (%)	20.5 / 19.5	1.0 / 10.0	0.5 / -	1.0 / 1.0	0.0 / 0.0
	Memory (MB)	166.9 / 167.8	107.2 / 114.3	107.6 / -	107.9 / 104.9	108.2 / 105.3
#5	CPU (%)	20.0 / 18.0	2.0 / 11.0	0.1 / -	1.5 / 3.0	0.0 / 0.0
	Memory (MB)	162.9 / 166.5	106.2 / 110.5	107.3 / -	107.5 / 111.0	107.8 / 111.4
#6	CPU (%)	20.0 / 19.5	1.0 / 8.5	0.1 / -	0.1 / 1.5	0.0 / 0.0
	Memory (MB)	158.7 / 155.6	107.9 / 105.1	108.1 / -	108.4 / 105.6	108.7 / 106.0
#7	CPU (%)	18.0 / 18.0	0.1 / 9.5	0.5 / -	2.0 / 2.0	0.0 / 0.0
	Memory (MB)	159.4 / 160.0	108.5 / 101.9	109.2 / -	109.3 / 102.4	109.7 / 102.8
#8	CPU (%)	19.5 / 18.5	0.1 / 10.0	0.5 / -	1.0 / 3.5	0.0 / 0.0
	Memory (MB)	158.3 / 162.4	109.2 / 106.4	109.6 / -	109.9 / 106.9	110.3 / 107.3
#9	CPU (%)	20.5 / 19.5	1.5 / 9.0	0.5 / -	0.5 / 3.0	0.0 / 0.0
	Memory (MB)	158.7 / 158.3	106.7 / 100.8	108.7 / -	108.9 / 101.3	109.1 / 101.7

versions with different characteristics from the most recent versions.

Other datasets are unbalanced, and contain many applications with different characteristics and, in some cases, wrongly labeled. Another problem is that although the hash of the samples may differ in some cases, when we extracted the features of these applications, many were identical (considering more than 200 features extracted). Using many duplicate samples may create biased models. The existence of duplicate samples can be explained by considering a developer that creates a *malware* application and then slightly changes it so that it can appear to be different, evading signature-based detectors (e.g., detectors based on the samples hash).

In our work, we created datasets with specific characteristics of the threats of interest. Besides, we adopted practices during the dataset processing to avoid problems with wrongly labeled samples, unbalanced sets, and also with duplicate samples (considering the samples' feature vector values). Moreover, we collected more recently developed (or at least updated) *malware* and *benign* applications (created or released in the period of 2017 - 2022).

VI. CONCLUSION

In this work, we presented *Malware APK Detection Solution* (MADS), a novel, lightweight, and non-invasive approach to detect Android *malware*. We showed that using a set of specific-type detectors, in a multi-stage analysis fashion combining rules and machine learning techniques, we can overcome strategies often used by state-of-the-art solutions in terms of detection capabilities and performance (execution time and resources usage).

MADS specific-type detectors obtained a significant reduction of nine times less false positive cases than a general model, related to state-of-the-art solutions, which was trained with three different *malware* types at once. This benefit also comes with a reduction in the resources consumed by MADS in relation to the same state-of-the-art approach under comparison; the total execution time for all tested applications was three times faster, and MADS consumed ten times less CPU resources in some cases.

In addition to the performance advantages obtained, the proposed solution is also in compliance with licenses or terms of use adopted by mobile applications that forbid reverse engineering, decompilation or disassembly of the application under analysis. Future work encompasses a detailed analysis of how variations on the threshold for the *CL* (*Confidence Level*) values affect the classification results, as well as the impact of *concept drift* in relation to the classification of Android *malware*. An evaluation of adversarial machine learning-based attacks is also in the scope of next steps.

ACKNOWLEDGMENT

The authors would like to thank Allisson Andrade Franco, Valdir Souza Junior, and Gabriel de Alcantara Bomfim Silveira for their contributions.

REFERENCES

[1] P. Kotzias, J. Caballero, and L. Bilge, "How did that get in my phone? Unwanted app distribution on Android devices," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2021, pp. 53–69. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9519429

[2] J. Johnson. (2021). *Development of Android Malware Worldwide From June 2016 to March 2020*. Accessed: May 31, 2022. [Online]. Available: https://www.statista.com/statistics/680705/global-android-malware-volume/

[3] N. Peiravian and X. Zhu, "Machine learning for Android malware detection using permission and API calls," in *Proc. IEEE 25th Int. Conf. Tools with Artif. Intell.* Herndon, VA, USA: IEEE Computer Society, Nov. 2013, pp. 300–305.

[4] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proc. 12th USENIX Secur. Symp.* Washington, DC, USA: USENIX Association, Aug. 2003, pp. 169–186. [Online]. Available: https://www.usenix.org/legacy/events/sec03/tech/full_papers/christodorescu/christodorescu.pdf

[5] C. Bai, Q. Han, G. Mezzour, F. Pierazzi, and V. S. Subrahmanian, "DBank: Predictive behavioral analysis of recent Android banking trojans," *IEEE Trans. Depend. Sec. Comput.*, vol. 18, no. 3, pp. 1378–1393, May/Jun. 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8684321

[6] S. Alsoghyer and I. Almomani, "Ransomware detection system for Android applications," *Electronics*, vol. 8, no. 8, p. 868, Aug. 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/8/868

[7] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, vol. 14, 2014, pp. 23–26. [Online]. Available: https://prosec.mlsec.org/docs/2014-ndss.pdf

[8] M. Weizhi, Ed., *Protecting Mobile Networks and Devices: Challenges and Solutions*. Boca Raton, FL, USA: Taylor & Francis, 2016. [Online]. Available: https://repository.ubn.ru.nl/handle/2066/166088

[9] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8255798

[10] A. Salah, E. Shalabi, and W. Khedr, "A lightweight Android malware classifier using novel feature selection methods," *Symmetry*, vol. 12, no. 5, p. 858, May 2020. [Online]. Available: https://www.mdpi.com/2073-8994/12/5/858

[11] F. Pierazzi, G. Mezzour, Q. Han, M. Colajanni, and V. S. Subrahmanian, "A data-driven characterization of modern Android spyware," *ACM Trans. Manage. Inf. Syst.*, vol. 11, no. 1, pp. 1–38, Mar. 2020. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3382158

[12] V. Sihag, G. Choudhary, M. Vardhan, P. Singh, and J. T. Seo, "PICAndro: Packet inspection-based Android malware detection," *Secur. Commun. Netw.*, vol. 2021, pp. 1–11, Nov. 2021. [Online]. Available: https://www.hindawi.com/journals/scn/2021/9099476/

[13] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-droid: Deep learning based Android malware detection using real devices," *Comput. Secur.*, vol. 89, Feb. 2020, Art. no. 101663. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404819300161

[14] Android Sandbox. (2023). *Sandbox Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: https://source.android.com/security/app-sandbox

[15] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "EMULATOR vs REAL PHONE: Android malware detection using machine learning," in *Proc. 3rd ACM Int. Workshop Secur. Privacy Anal.*, Mar. 2017, pp. 65–72. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3041008.3041010

[16] Android Open Source Project. (2023). *PackageManager*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/reference/android/content/PM/PackageManager

[17] A. Pektaş and T. Acarman, "Deep learning for effective Android malware detection using API call graph embeddings," *Soft Comput.*, vol. 24, no. 2, pp. 1027–1043, Jan. 2020. [Online]. Available: https://link.springer.com/article/10.1007/s00500-019-03940-5

[18] H. Gao, S. Cheng, and W. Zhang, "GDroid: Android malware detection and classification with graph convolutional network," *Comput. Secur.*, vol. 106, Jul. 2021, Art. no. 102264. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821000882

[19] I. J. Sanz, M. A. Lopez, E. K. Viegas, and V. R. Sanches, "A lightweight network-based Android malware detection system," in *Proc. IFIP Netw. Conf.*, Paris, France, Jun. 2020, pp. 695–703. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9142796

[20] A. Gharib and A. A. Ghorbani, "DNA-Droid: A real-time Android ransomware detection framework," in *Network and System Security* (Lecture Notes in Computer Science), vol. 10394. Helsinki, Finland: Springer, Aug. 2017, pp. 184–198. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-64701-2_14

[21] A. Mahindru and A. L. Sangal, "MLDroid—Framework for Android malware detection using machine learning techniques," *Neural Comput. Appl.*, vol. 33, no. 10, pp. 5183–5240, May 2021. [Online]. Available: https://link.springer.com/article/10.1007/s00521-020-05309-4

[22] J. Kim, Y. Ban, E. Ko, H. Cho, and J. H. Yi, "MAPAS: A practical deep learning-based Android malware detection system," *Int. J. Inf. Secur.*, vol. 21, no. 4, pp. 725–738, Aug. 2022.

[23] M. Jerbi, Z. C. Dagdia, S. Bechikh, and L. B. Said, "Android malware detection as a bi-level problem," *Comput. Secur.*, vol. 121, Oct. 2022, Art. no. 102825.

[24] İ. Atacak, "An ensemble approach based on fuzzy logic using machine learning classifiers for Android malware detection," *Appl. Sci.*, vol. 13, no. 3, p. 1484, Jan. 2023.

[25] Y. Wu, J. Shi, P. Wang, D. Zeng, and C. Sun, "DeepCatra: Learning flow- and graph-based behaviours for Android malware detection," *IET Inf. Secur.*, vol. 17, no. 1, pp. 118–130, Jan. 2023.

[26] D. Ö. Sahin, S. Akleylek, and E. Kiliç, "LinRegDroid: Detection of Android malware using multiple linear regression models-based classifiers," *IEEE Access*, vol. 10, pp. 14246–14259, 2022.

[27] Koodous. (2023). *Collaborative Platform for Android Malware Analysts*. Accessed: Jun. 3, 2023. [Online]. Available: https://koodous.com/

[28] M.-C. Jang, K.-J. Kwak, J. Kim, and S. Kim. (2019). *When Voice Phishing Met Malicious Android App*. Accessed: May 31, 2022. [Online]. Available: https://www.blackhat.com/asia-19/briefings/schedule/

[29] Android Permissions. (2023). *Android Permissions Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission

[30] Android Hardware Components. (2023). *Android Hardware Components Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/guide/topics/manifest/uses-feature-element#hw-features

[31] Android Application Components. (2023). *Android Application Components Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/guide/components/fundamentals#Components

[32] Android Intent Filters. (2023). *Android Intent Filters Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/guide/components/intents-filters

[33] RFE. (2023). *Recursive Feature Elimination Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

[34] SelectFromModel. (2023). *SelectFromModel Documentation*. Accessed: Jun. 3, 2023. [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectFromModel.html

[35] Jared Rummler. (2023). *APKParser*, 2023. Accessed: Jun. 3, 2023. [Online]. Available: https://github.com/jaredrummler/APKParser

[36] AppBrain. (2023). *Top Android Apps and Games on Google Play*. Accessed: Jun. 3, 2023. [Online]. Available: https://www.appbrain.com/

[37] VirusTotal. (2023). *Analyse Suspicious Files, Domains, IPS and URLs to Detect Malware and Other Breaches, Automatically Share Them With the Security Community*. Accessed: Jun. 3, 2023. [Online]. Available: https://www.virustotal.com/

[38] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark Android malware datasets and classification," in *Proc. Int. Carnahan Conf. Secur. Technol. (ICCST)*, Montreal, QC, Canada, Oct. 2018, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8585560

[39] S. Mahdavifar, A. F. A. Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic Android malware category classification using semi-supervised deep learning," in *Proc. IEEE Int. Conf. Dependable, Autonomic Secure Comput., Int. Conf. Pervasive Intell. Comput., Int. Conf Cloud Big Data Comput., Int. Conf. Cyber Sci. Technol. Congr. (DASC/PiCom/CBDCom/CyberSciTech)*, Calgary, AB, Canada, Aug. 2020, pp. 515–522. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9251198

[40] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-PackDroid: API package-based characterization and detection of mobile ransomware," in *Proc. Symp. Appl. Comput.*, Marrakech, Morocco, Apr. 2017, pp. 1718–1723. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3019612.3019793

[41] Malpedia. (2023). *Resource for Rapid Identification and Actionable Context When Investigating Malware*. Accessed: Jun. 3, 2023. [Online]. Available: https://malpedia.caad.fkie.fraunhofer.de/

[42] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*. Buenos Aires, Argentina: IEEE Computer Society, May 2017, pp. 425–435. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7962391

[43] Scikit-Learn. (2023). *Machine Learning in Python*. Accessed: Jun. 3, 2023. [Online]. Available: https://scikit-learn.org/

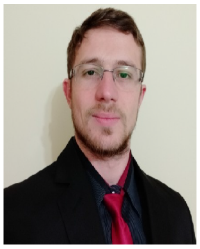
[44] P. Agrawal and B. Trivedi, "Machine learning classifiers for Android malware detection," in *Data Management, Analytics and Innovation*, N. Sharma, A. Chakrabarti, V. E. Balas, and J. Martinovic, Eds. Singapore: Springer, 2021, pp. 311–322.

[45] M. S. Rana, C. Gudla, and A. H. Sung, "Evaluating machine learning models for Android malware detection: A comparison study," in *Proc. VII Int. Conf. Netw., Commun. Comput.*, Taiwan, Dec. 2018, pp. 17–21.

[46] SystemClock. *SystemClock—Android Developers*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/reference/android/os/SystemClock

[47] Profile. (2023). *Profile Your App Performance—Android Developers*. Accessed: Jun. 3, 2023. [Online]. Available: https://developer.android.com/studio/profile

LEONARDO DA COSTA received the first M.Sc. degree in computer science from the Stevens Institute of Technology, USA, and the second M.Sc. degree in computer science from Universidade Federal do Pará (UFPA), Brazil, where he is currently pursuing the Ph.D. degree. Since 2020, he has been working as a Cybersecurity Researcher with the Samsung Research and Development Institute Brazil (SRBR). His main research interests include cryptographic protocols, blockchain, health records security, network security, android security, and machine learning applied to security.



VITOR MOIA received the Computer Engineering degree from the Centro Regional Universitário de Espírito Santo do Pinhal in 2013, the master's degree in computer engineering from the University of Campinas (Unicamp), Brazil, in 2016, and the Ph.D. degree from Unicamp in February 2020, working in the digital forensics field. During his master's thesis, he conducted a study about the security and privacy on cloud data storage. He worked for about three years as a Security Researcher with the Samsung Research and Development Institute, where he became a Project Leader and conducted applied research activities in mobile and network security. Since September 2022, he has been working as a Cyber Security Consultant at Eldorado Institute, Brazil, in red team activities and secure software development lifecycle (SSDLC) adoption.