Is It Safe? Identifying Malicious Apps Through the Use of Metadata and Inter-Process Communication

Rodrigo Lemos, Tiago Heinrich, Carlos A. Maziero
Computer Science Department
Federal University of Paraná State
Curitiba, Brazil
{rglemos, theinrich, maziero}@inf.ufpr.br

Newton C. Will
Computer Science Department
Federal University of Technology - Paraná
Dois Vizinhos, Brazil
will@utfpr.edu.br

***Abstract*—In recent years, the growth in the number of threats on Android has contributed to increasing user awareness and concern about security-related concepts. Due to the predominance of Android, the attacks present on the platform have also evolved, and new strategies for identifying threats are needed. A popular way to identify threats is the use of intrusion detection systems, which can exploit different strategies to carry out threat identification. Static analysis strategy aims to identify malicious apps by scanning their source code, and dynamic analysis uses the behavior monitor approach to classify benign and malicious apps. These two strategies can also be combined in a hybrid approach. This paper focuses on a hybrid strategy to identify threats in Android systems through the use of static metadata extracted from applications and dynamic data from inter-process communication, in order to train machine learning models to perform threat identification. Three machine learning algorithms were used to verify the efficacy of our strategy. Our approach showed to be viable, with the results presenting an identification rate of around 87%, demonstrating that the proposed model has benefits in identifying threats in Android mobile devices. We also point out attributes that differ between malicious and benign apps and highlight the impact on the use of inter-process communication to identify threats.**

***Index Terms*—Android Security, Intrusion Detection, System security, Malware Detection.**

I. Introduction

Social engineering techniques are often used by attackers to compromise mobile devices, often offering malicious apps such as hot apps at the moment [1]. The complexity of getting access to mobile devices remotely implies this need for user actions (installing an application, clicking on an *Uniform Resource Locator (URL)*, or connecting to a wireless network) to compromise the system. Thus, is important to evaluate the applications to be installed in Android systems to maintain their security.

During the Covid-19 lockdown period it was possible to observe the growth in these types of attacks. One example is *covidLock*, which is a ransomware that tricks users by promising to provide real-time monitoring information for coronavirus. After deceiving users, it locks the device and requests a ransom [2]. An Avira report [3] highlights that 59% of Americans install some type of user-safety-oriented software, with 81% of users stating that security on mobile devices is essential. This same study highlights a 28% growth in concern for the safety of mobile devices during the pandemic. This growth is also presented by Kaspersky [1].

Distinct malware identification strategies exist in the Android environment. Despite the security checks found in app stores (such as Google Play), there are malicious apps that pretend to be legitimate and are not detected by the stores' detection engines [4].

In the literature, a wide range of approaches are presented to carry out the identification of threats on Android. For malware identification we can find works that use system calls, *hardware* device usage information [5], network traffic [6], and *Intents* calls [7].

Our focus on identifying threats is to utilize application metadata and inter-process communication (IPC) [8] data obtained from application executions. These data will be used to represent behaviors expected from applications, thus using machine learning strategies to learn to distinguish between harmless and harmful behaviors.

The following contributions are presented in this work:

- A new strategy as a discussion of using hybrid data for Android threat identification;
- A strategy for observing behavior and identifying threats using machine learning techniques for Android; and
- A dataset named AndroBlend [9], containing hybrid data, in order to identify threats on Android.

The remainder of this paper is organized as follows. Section II presents the background concepts for understanding the paper. Section III presents our proposal. Section IV present the evaluation results and discusses our findings. Section V presents the related work and compares our proposal with the literature. Finally, Section VI concludes the paper.

II. Background

This Section presents a brief description of important concepts used in this work, contextualizing how threat identification is performed on Android and what are the most common strategies to assess data.

A. Malware Detection on Android

The detection of malware threats on Android is an essential task, and over the years new strategies have been proposed to improve user security. *Android Runtime (ART)* is responsible

for implementing the virtual machine used by each Android application and executes its bytecode, in a sandbox having an unique *User ID (UID)* inside its own process [10]. This feature is similar to the *Java Virtual Machine (JVM)*, however ART is register-based and uses an ahead-of-time compilation to improve runtime performance of apps [11].

In addition to the applications running in an isolated environment, Android provides a kernel-level secure communication channel between processes, known as *Binder* [12]. Apps also use signatures to ensure their authenticity and permissions to ensure an user driven level of security on Android, besides all Linux kernel resources, such as SELinux, that are used to provide more robust access control policies [13].

Although these security layers are designed to make Android a secure environment, there are threats able to find breaches on it and to circumvent these security measures. This problem is highlighted by the substantial growth in threats in this environment [1].

In order to minimize the damage of such threats, several approaches are explored. Malware detection systems are one of them, aimed to monitor and identify malicious applications. Such approaches are very relevant, since Android has became the top one target to malware creators [14].

Malware can be defined as any code added, modified, or removed from a software system with the aim of causing damage or using differently a certain functionality of the system [15]. In this context, we consider an Android malware any app having this purpose.

When analysing an app, a malware detection system may behave in different ways due to its design. For example, its mechanism may work online or offline, process data locally or in a cloud environment, and use anomaly or signature detection methods. These design choices are relevant for the performance of the method and also by the impact they cause on the devices performance and use of resources.

The main way to differentiate these approaches are according to the origin of the analyzed data; they are divided into static, dynamic, and hybrid analysis methods [16], where hybrid analysis stands for the use of a combination of both static and dynamic data.

B. Malware Detection Strategies

Two strategies are used to identify threats on Android systems, these being:

1) Static analysis: aims to study the source or binary code of a threat (malware) without executing it [17]. This evaluation on Android is done by exploring the `.apk` files, which refers to the compressed application data. This file contains the resources, configuration files, and the bytecode (`.dex`) that will be interpreted in ART. This strategy ends up being fast and lightweight [18], but it may not be able to represent all the behaviors of an application. Obfuscation techniques or runtime code generation tend to escape this type of evaluation [19].

2) Dynamic analysis: explores application execution to capture information and perform threat identification. This evaluation can take into account a varied set of characteristics and operations that an application performs, such as: use of resources, type of communication operations performed, system calls, and communication between applications. As a result, this strategy is not susceptible to obfuscation, in addition to extracting more information [18]. However, it has the need for more resources and recent threats already highlight attackers' strategies to detect and/or circumvent this type of strategy [17].

III. Proposal

The constant evolution of threats in the Android environment ends up representing how attackers are improving over time. Because of this, new security measures are needed aiming to keep up with new attacks. Our work aims to improve security in the Android environment, through the identification of malicious applications.

Through a hybrid set of data, consisting of inter-process communication data and application metadata, we aim to identify threats in applications published in the app store. The identification procedure will focus on the use of machine learning algorithms to define expected behavior for malicious apps.

Looking at the state of the art mechanisms for detecting malware it is clear that machine learning approaches are used in most of them and are accurate in detecting malware [16]. The use of machine learning strategies to identify threats on Android has gained attention in the last decade. It is possible to highlight a wide range of strategies aimed at solving problems found in the mobile device environment, with better results than classical strategies [20]. It can be implied that the popularity of such methods is due to their performance and flexibility, since they can deal with any kind of data and can be ported to any platform such as the mobile device itself or outsourced to a cloud.

A. Data Approach

The proposed evaluation strategy focuses on previously collect data, intended to identify malicious apps. This method of monitoring allows to identify threats without any limitation of device resources. It is advantageous to identify threats in apps before they are made available in the app store.

Android's architecture forces apps to use its inter-process communication mechanism to gain access to system and hardware resources. Communication between apps and system services also requires the use of this feature. Fig. 1 shows how two apps communicate through the use of `Intents`. A message will be generated by an activity of `App1`, this message is forwarded to a `handler` that will be responsible for exchanging messages with the Android Binder IPC facility. When Binder receives this message, it forwards it to the `handler` of `App2` until it reaches the intended service. As such, tracking and understanding communications on this channel is a key to understanding Android-specific behaviors [21].

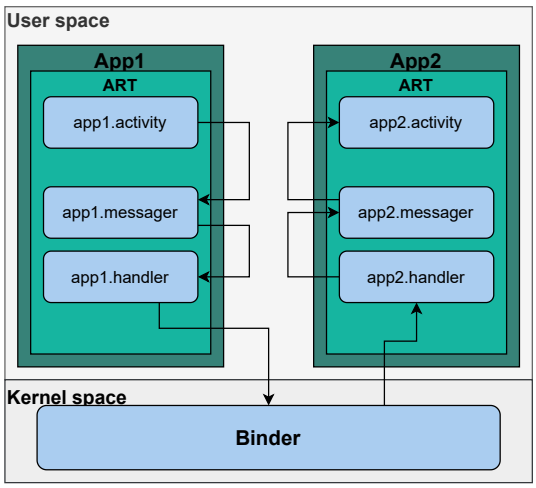

Fig. 1. Inter-process communication engine on Android.

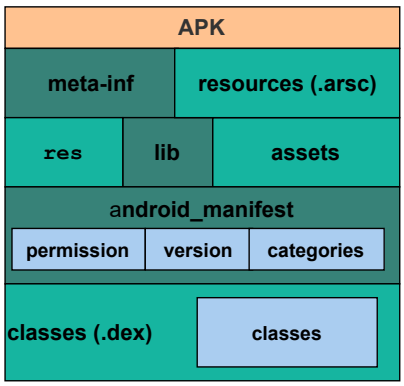

Fig. 2. Structure of the information found in an APK file.

The interaction between applications and with system services is carried out through `ioctl` system calls to Binder [22]. A simple example would be an application that wants read and write access to a system resource, which ends up sending a `BINDER_WRITE_READ` call to Binder, and consequently, Binder forwards this request to the corresponding service. Two types of requests can be issued using in this system call, (1) requests sent by services to Binder (`BC_`) and (2) requests sent by Binder to services (`BR_`).

The sequence of requests issued by an application through `ioctl` calls is explored to extract dynamic features using two distinct approaches. The first one is to count the number of requests issued by the application, resulting in a table of requests frequency. The second one captures the relative sequence of requests using n-grams[1], which are processed using TF-IDF (*Term Frequency–Inverse Document Frequency*) to create a table of frequent request subsequences.

Static information can be extracted from *Android Application Package (APK)* files using reverse engineering tools. Fig. 2 highlights the information contained in an APK file. Our focus at this stage is the extraction of attributes from the `AndroidManifest` and the bytecode (`.dex` file).

From the `AndroidManifest` it was extracted the declaration of permissions, categories and minimum, target, and maximum versions of the *Software Development Kit (SDK)*. Categories are strings that define the kind of component that should handle an intent [24]. Permissions are used by an app to request access to a restricted data or resource [25]. SDK versions are relevant to have knowledge about which APIs are available to use and if an app is targeted for outdated and less protected devices.

Since `.dex` file contains the application's bytecode, we can extract the names of the classes present in the application's source code from it by using [26]. This data brings information about which APIs are used in the app and may also indicate whether the app source code was obfuscated.

The dataset used in this approach consists of a vector containing both dynamic and static attributes for describing each app, in order to have all attributes analyzed simultaneously by the classification algorithms [9].

B. Methodology

Since we performed an offline evaluation, the entire set of selected apps must be executed and analyzed beforehand. Thus, a controlled execution of the apps is performed to capture the information discussed in Section III-A. Subsequently, these characteristics are combined into a single characteristic vector that is used to classify the apps as malicious or benign, according to the trained machine learning model.

The dataset consists of the labels presented by the Androzoo dataset [27]. Due to the constant updating of this dataset, characteristics of 622 apps launched between 2018 and 2020 were collected, 319 being benign apps and 303 malicious apps, thus representing a balanced dataset.

The dataset was normalized and divided into 80% for training/validation and 20% for testing. The training data were used for training and validation of the models through *k-fold* cross validation, with $k = 5$. An attribute selection strategy was applied in order to select the 10% of the attributes with the highest relevance. Finally, three machine learning strategies were explored: *Multilayer Perceptron*, *Random Forest* and *AdaBoost*. The choice of these strategies was due to the popularity in the literature and the low cost of training the models.

The tests where performed using the Android Studio environment emulator, version 3.6.3. The *Android Virtual Device (AVD)* used has a Nexus 4.0 hardware profile and an Android Q (Android 10.0) system image, with API level 29 and x86 architecture. In addition, AVD was configured with 8 GB of RAM and 50 GB of storage. Android 10.0 was chosen because it is the most used version of Android in the world since August 2020, with 32.5% of market share in August 2021 [28].

[1] n-grams are overlapping sequences of consecutive requests of size n obtained through a sliding window approach [23].

IV. Evaluation

This Section presents our evaluation process and a discussion about the use of metadata and inter-process communication for the identification of malicious apps. Our dataset was split into training and testing bases in a 80:20 ratio, maintaining a balanced distribution between classes. The training dataset was also split into training and validation bases using the same ratio.

From the dataset, it is important to highlight that two pieces of information enrich the application profile. The communication between processes that is dynamic information, extracted from an application execution and discloses how an application interacts (communicates) with services and other applications; and metadata that present static information such as application categories and classes.

Four key points in Android threat identification using hybrid data will be evaluated in this section. Such assessments take into account key information to distinguish malicious apps. The evaluations are: (i) to identify among the attributes extracted from `AndroidManifest` which ones are representative for malicious apps; (ii) to identify which attributes have a statistically relevant difference between malicious and benign apps; (iii) to assess the impact on the use of process communication information to identify threats; and (iv) to discuss the impact of using machine learning strategies to identify threats in the app store.

A. Feature Analysis

When evaluating the most popular features, the first factor that stands out is the difference between the number of permissions related to accessing and changing network elements between benign and malicious apps. These features are related to `WIFI` or `NETWORK` states, which are directly related to the communication of applications; malicious apps uses such features 24% and 18% more than benign apps, respectively. This tendency for malicious apps to communicate can be related to factors such as extracting data, discovering new victims, or even requesting advertisements.

Network access isn't the only popular permission that malicious apps focus on. The `PHONE` state also gets attention from malicious apps, as it allows access to communication resources as well as to device connections resources. 26.2% of observed malicious apps requested this permission, which means 40.3% more than benign apps. This permission allows malicious apps to monitor user actions and watch private information, such as calls. This difference is also shown in Figure 4, as one of the top differences between malicious and benign apps.

`LOCATION` and `INSTALLATION` permissions also have a higher presence in malicious apps; as an example, `BROADCAST_PACKAGE_INSTALL` and `ACCESS_LOCATION_EXTRA_COMMANDS` are used 4.1 and 3.1 $\times$ more in malicious apps than in benign ones. An install permission will allow a malicious app to install other features or even other applications on the device without notifying the user or requesting his explicit approval. Thus, such permissions need some attention from users.

Looking into the permissions only present in malicious apps, the ones related to running tasks based in context information, running instrumentation, and interacting between different users stands out. Such kind of permissions are very representative for malicious actions, since two of them allows an app to execute tasks automatically and the other opens a breach in the protection between users, which is one of Android's core security mechanisms.

Fig. 3 presents the classes used at least 5$\times$ more in malicious apps. Looking at the classes, it's highlighted that from the 36 `zxing` classes that appeared in the dataset, 29 of them are used at least 5$\times$ more in malicious apps. The `zxing` case gained notoriety in media recently as a barcode scanner app that was cloned and refurbished in a malicious one, affecting many people [29].

Another class that stands out is the `org/apache/cordova/inappbrowser`, which opens a window that behaves like a standard web browser, but is not subjected to any whitelist, and therefore is recommended to load third-party (untrusted) content [30]. This class is used 5.2 $\times$ more in malicious apps.

Finally, it was noted that many of malware applications uses the same kind of code obfuscation. This is due to many classes used mostly by malicious apps had the same pattern in their name of repeated single letters, such as `com/b/a/a/a` and `com/a/a/a/a`, for example. This fact shows the importance of having dynamic features for detecting malware, because many of them uses code obfuscation, especially the malicious ones.

Fig. 4 shows the static features with the greatest discrepancy in use between malicious and benign apps. These thirty attributes are the features with the biggest difference among around 14.7 k features. The X axis shows the attributes, and the Y axis shows the percentage of malicious and benign apps using it. It is noteworthy that the average difference between malicious and benign apps is -0.06%, with the third quartile having an average value of 0.04%.

Overall, the classes with the greatest discrepancy in use between malicious and benign apps show that many of malware apps are disguised as usual ones, such as fitness apps and even apps with paid options linked to Google Play Services.

The `com/stub` class found in Fig 4 also highlighted a different behaviour regarding the use of inter-process communication by malicious and benign apps, since it implements a remote interface, aiming at sending and receiving messages locally for services (similar to a Remote Procedure Call (RPC)). This information emphasizes the relevance of the dynamic data collected to characterize a malware behaviour.

Looking at the dynamic data collected, it was observed that malicious apps make 17.4% more calls per application into the inter-process communication mechanism than benign ones. This factor ends up highlighting the importance of observing calls between processes to identify threats in the Android environment.

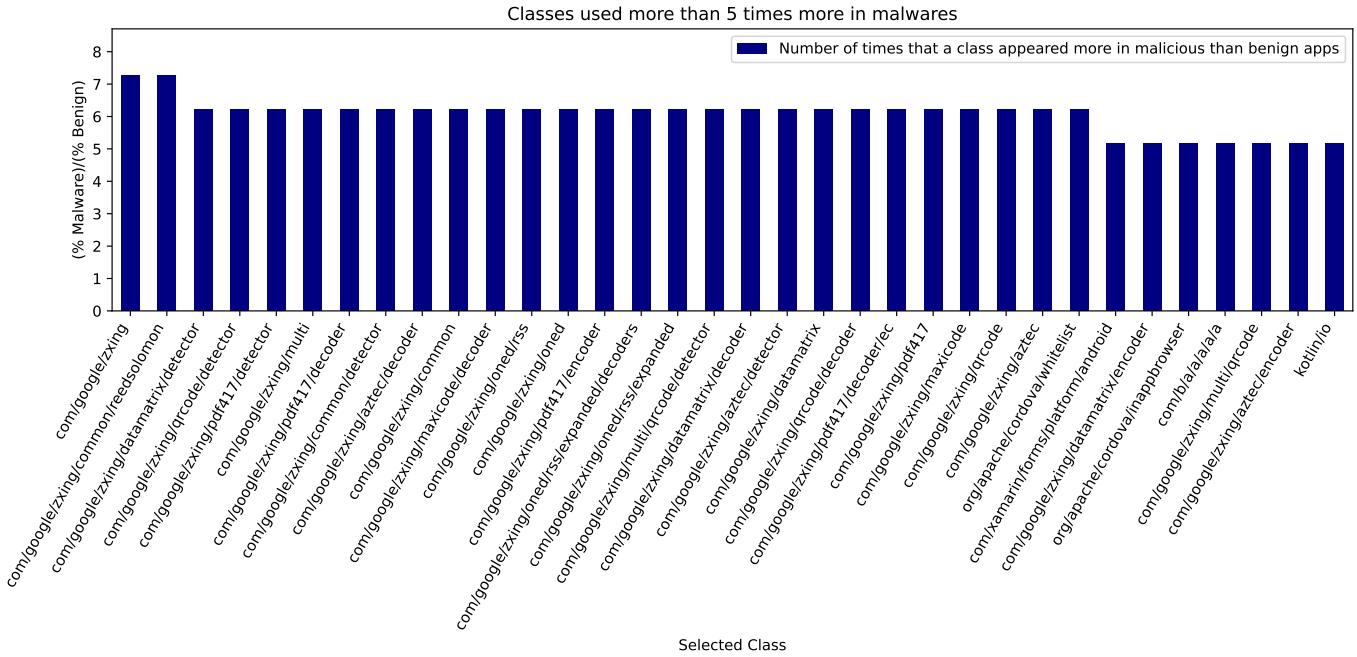

Fig. 3. Ratio between use of classes between malicious and benign applications.

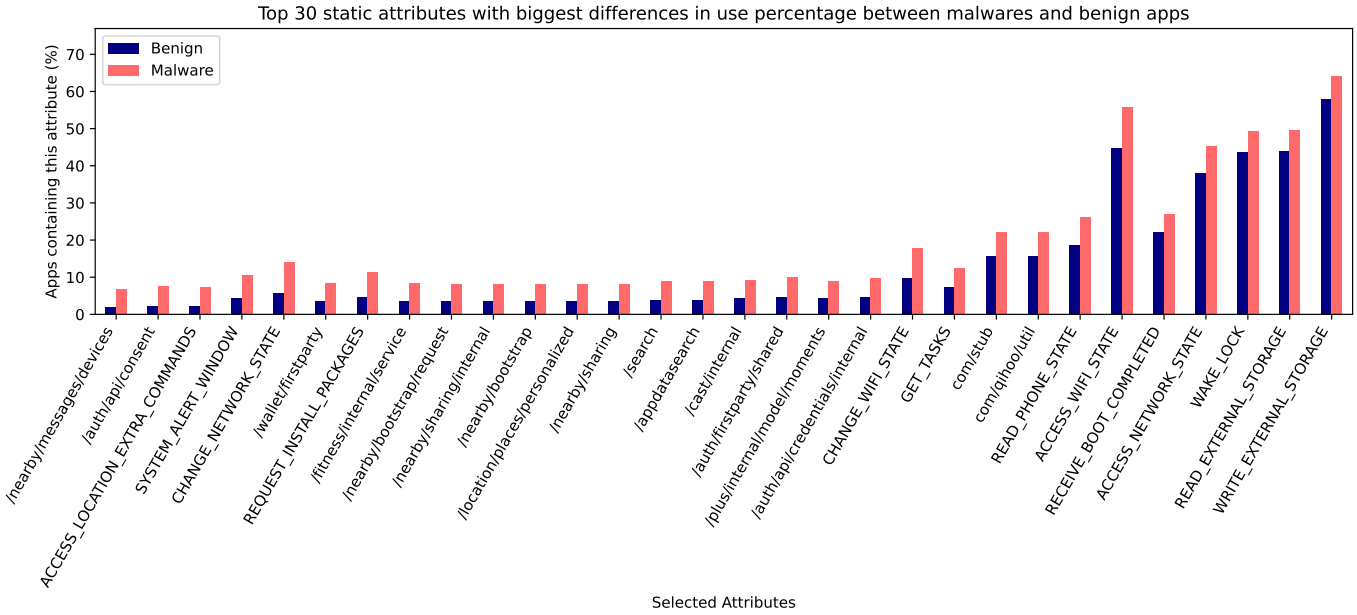

Fig. 4. Percentage of malicious and benign apps using a feature (ordered by relative differences).

This difference is much higher regarding the commands related to registering threads to the Binder service, which suggests that this apps intends to send many messages to other processes simultaneously. The reason behind this statistics is likely to be that besides the normal app tasks, malware also intends to perform their malicious actions while being executed, such as launching unwanted adds, accessing files, sending data to the internet, etc.

B. Results

The results obtained are presented in Tables I and II. They contain are the average values obtained after five runs using cross-validation for both representation approaches, using the validation dataset (as presented in Section III-B). The metrics were chosen focusing on the best description of the behavior found.

The results presented in Tables I and II highlight that the use

TABLE I
AVERAGE PERFORMANCE ON VALIDATION (V) AND TEST (T) DATASETS USING THE NUMBER OF CALLS REPRESENTATION FOR DYNAMIC FEATURES.

Algorithm	F1 Score		Recall		Precision		Brier Score		ROC AUC	
	V	T	V	T	V	T	V	T	V	T
Random Forest	91.7%	86.1%	90.2%	76.9%	93.6%	97.9%	7.9%	13.8%	92.2%	87.4%
AdaBoost	93.4%	87.3%	91.3%	77.5%	95.7%	100.0%	6.3%	12.6%	93.7%	88.7%
Multilayer Perceptron	92.9%	87.0%	93.7%	82.0%	92.2%	92.8%	6.9%	13.7%	93.2%	86.9%

TABLE II
AVERAGE PERFORMANCE ON VALIDATION (V) AND TEST (T) DATASETS USING THE 3-GRAMS REPRESENTATION FOR DYNAMIC FEATURES.

Algorithm	F1 Score		Recall		Precision		Brier Score		ROC AUC	
	V	T	V	T	V	T	V	T	V	T
Random Forest	89.2%	84.8%	90.5%	78.6%	88.0%	92.1%	10.7%	15.7%	89.4%	85.0%
AdaBoost	89.0%	85.1%	90.9%	78.9%	87.3%	92.4%	10.9%	15.4%	89.2%	85.3%
Multilayer Perceptron	91.9%	86.1%	93.0%	80.3%	91.1%	93.1%	7.9%	14.5%	92.4%	86.2%

of metadata and communication between processes to identify threats can achieve satisfactory results. F1Score demonstrates that all algorithms were able to fit both in the test and validation datasets. Recall would be the only metric to which a feature selection could contribute to improving its result. Taking into account the value achieved by the ROC AUC and Brier Score, it is possible to observe the learning and adequacy of data in relation to the models, validating this strategy for identifying threats.

Comparing the results from both Tables it is noted that the 3-grams approach got slightly worse results, with a decrease in performance for all algorithms tested. This decrease is probably due to the greater complexity of its data, since this approach generated 4.4 $\times$ more features. A greater complexity in data also explains why `Multilayer Perceptron` obtained the best results in this approach, since it is a neural network and therefore designed for more complex data.

Besides this difference in results, the 3-grams approach have some advantages: (i) with the greatest focus on dynamic data, it is less susceptible to obfuscation techniques; (ii) the dynamic data in it represents how communication between processes is performed. The success in this approach, where most of the data was from the dynamic source, shows the relevance of the inter-process communication in malicious behaviour.

The approach using only the number of calls as dynamic data (Table I) and the *AdaBoost* algorithm obtained the best results considering the adequacy of the models. But it still has values close to *Multilayer Perceptron*, that obtained a similar result in both data approaches. Overall, we can say that the algorithms managed to adapt to the dataset and it was possible to identify malicious apps.

The ROC curves representing the average performance in cross-validation folds for both approaches are represented in Figs. 5 and 6. The curves demonstrate that the performance remained consistent in all stages of the cross-validation, as the discrimination threshold varied.

With the results achieved, we can say that the use of meta-data together with information for inter-process communication

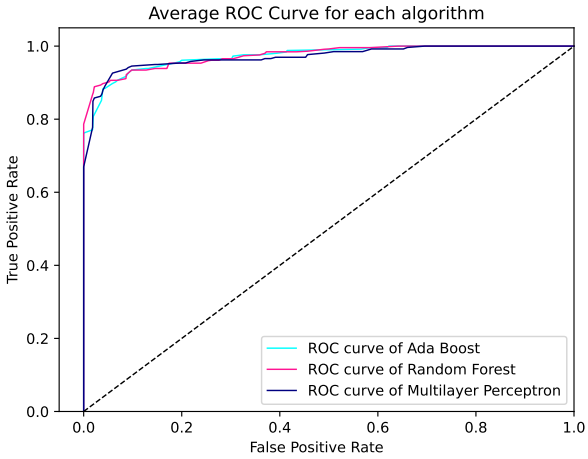

Fig. 5. Average ROC curves using the number of calls as dynamic data.

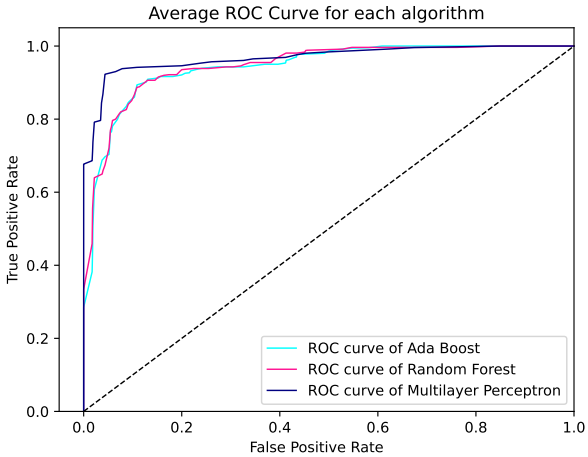

Fig. 6. Average ROC curves using 3-grams as dynamic data.

is a feasible strategy for identifying threats in the Android environment. However, some points regarding the use of hybrid data to identify threats should be discussed.

The use of dynamic data in the Android environment is impacted by its high cost in resources, which are limited in mobile environments. Working with an approach that focuses on processing and verifying applications outside the mobile environment solves the impact that the use of dynamic data would have. In addition, this approach prevents malicious apps from propagating in the app store. This contribution is relevant, since the stores aren't free from malware samples and even `Google Play` makes some of them available, as shown in the Androzoo dataset [27].

Finally, the use of hybrid data contributes to a deeper insight into the expected behaviors of an app, as we can assess both the metadata declared in the manifest and the actual resource usage when the app runs.

V. Related Work

The detection of malicious apps on Android systems is a research field that receives a lot of attention in the literature, with several techniques being proposed. A static approach is proposed by [19], extracting information from applications and building domain-specific models, which are used to analyze security settings and identify vulnerabilities in the application. This solution is vulnerable to code obfuscation and requires a specific model to identify each vulnerability.

Several works use permissions to detect Android malware, such as [31] that uses pairs of permissions that may be dangerous to identify malicious apps. Another solution performs a permission usage analysis to identify malware, mining permission data and identifying the most relevant permissions to distinguish benign and malicious apps, and using machine learning to classify malware families [32].

A dynamic approach is proposed by [33], using a set of features based on inter-component communication Intents and method calls to perform an application classification aiming to differentiate benign and malicious apps. A dynamic analysis of runtime application behavior is performed by [34], extracting data from system calls and Binder communications, and the authors described that data recovered from Binder calls improve the detection accuracy. These approaches can overcome obfuscation techniques and the use of reflection in source code by malicious apps, which are a challenge for static approaches.

Hybrid approaches are also adopted in the literature, as in [35], where authors use permissions and system calls as attributes, including their parameters. Arshad et al. [36] extracts data from *AndroidManifest.xml* and system calls logs and uses machine learning to process these data in remote servers, providing a real time analysis with minimum memory and battery overhead, but requires a constant network connection.

Our work focuses on a hybrid approach, collecting data from the `AndroidManifest`, such as permissions and categories, and extracting information from the application's bytecode , as described in Section III-A. This strategy overcomes the limitations of fully static approach, such as those presented by [19] and [31], and adds a new set of relevant data compared to fully dynamic approaches, such as those presented by [33] and [34], providing better results.

When compared with other hybrid approaches, such as those presented by [35] and [36], we can highlight the use of a more diverse set of static parameters and different machine learning algorithms, in addition to identifying the most relevant attributes for identifying malicious apps.

VI. Conclusion

This paper presented a new hybrid approach to detect malicious apps in Android systems, by collecting static and dynamic attributes from a new dataset, named AndroBlend [9], which is composed by the attributes extracted from 622 APK files.

We demonstrated that the attributes used to identify malicious apps presented feasible results using three different machine learning algorithms. It also demonstrated that our dataset is suitable for representing both benign and malicious apps. In addition, we discussed which attributes have greater relevance for identifying threats in the Android environment, highlighting permissions and recurring classes used by malicious applications.

As future work, we will improve the treatment of the dynamic data used, in order to more accurately represent the behavior of the analyzed applications. We will also study the correlation between static and dynamic attributes in identifying malicious apps, aiming to identify which combinations of attributes may be potentially dangerous. It is also possible to point out that the dataset can be expanded, aiming to cover a larger set of samples and, consequently, expanding the range of observed attacks. Consequently, new features used by malicious apps may be identified.

Acknowledgment

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil* (CAPES) – Finance Code 001. The authors also thank the UFPR and UTFPR Computer Science departments.

References

[1] V. Chebyshev, "Mobile malware evolution 2020," 2021. [Online]. Available: https://securelist.com/mobile-malware-evolution-2020/101029/

[2] CISA, "Alert (aa20-099a): COVID-19 exploited by malicious cyber actors," 2020. [Online]. Available: https://us-cert.cisa.gov/ncas/alerts/aa20-099a

[3] Avira, "Mobile security report," 2020. [Online]. Available: https://www.avira.com/en/mobile-security-report

[4] M. Rahman, M. Rahman, B. Carbunar, and D. H. Chau, "Search rank fraud and malware detection in Google Play," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 6, pp. 1329–1342, 2017.

[5] R. Zachariah, K. Akash, M. S. Yousef, and A. M. Chacko, "Android malware detection a survey," in *Proceedings of the International Conference on Circuits and Systems*. Thiruvananthapuram, India: IEEE, 2017, pp. 238–244.

[6] H. Kato, S. Haruta, and I. Sasase, "Android malware detection scheme based on level of SSL server certificate," *IEICE Transactions on Information and Systems*, vol. E103.D, no. 2, pp. 379–389, 2020.

[7] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-Droid: Deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, p. 101663, 2020.

[8] N. C. Will, T. Heinrich, A. B. Viescinski, and C. A. Maziero, "Trusted inter-process communication using hardware enclaves," in *Proceedings of the 15th Annual International Systems Conference*. Vancouver, BC, Canada: IEEE, 2021, pp. 1–7.

[9] R. Lemos, T. Heinrich, N. C. Will, and C. A. Maziero, "AndroBlend," 2020. [Online]. Available: https://github.com/Rodrigo-Lemos/AndroBlend

[10] Google, "Application sandbox," 2021. [Online]. Available: https://source.android.com/security/app-sandbox

[11] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proceedings of the European Symposium on Security and Privacy*. Paris, France: IEEE, 2017, pp. 481–495.

[12] T. Schreiber, "Android Binder," Master's thesis, Ruhr University, Bochum, Germany, 2011.

[13] Developer Android, "Secure Android devices," 2020. [Online]. Available: https://source.android.com/security

[14] S. Kumar and S. K. Shukla, "The state of android security," in *Cyber Security in India*. Springer, 2020, pp. 17–22.

[15] T. Alsmadi and N. Alqudah, "A survey on malware detection techniques," in *Proceedings of the International Conference on Information Technology*. Amman, Jordan: IEEE, 2021, pp. 371–376.

[16] V. Kouliaridis, K. Barmpatsalou, G. Kambourakis, and S. Chen, "A survey on mobile malware detection techniques," *IEICE Transactions on Information and Systems*, vol. E103.D, no. 2, pp. 204–211, 2020.

[17] J. Qiu, S. Nepal, W. Luo, L. Pan, Y. Tai, J. Zhang, and Y. Xiang, "Data-driven Android malware intelligence: A survey," in *Proceedings of the International Conference on Machine Learning for Cyber Security*. Xi'an, China: Springer, 2019, pp. 183–202.

[18] Y. S. I. Hamed, S. N. A. AbdulKader, and M. S. Mostafa, "Mobile malware detection: A survey," *Proceedings of the International Journal of Computer Science and Information Security*, vol. 17, no. 1, pp. 56–65, 2019.

[19] A. Nirumand, B. Zamani, and B. T. Ladani, "VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique," *Software: Practice and Experience*, vol. 49, no. 1, pp. 70–99, 2018.

[20] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124 579–124 607, 2020.

[21] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proceedings of the Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2015.

[22] N. Artenstein and I. Revivo, "Man in the Binder: He who controls IPC, controls the Droid," *BlackHat Europe*, 2014.

[23] M. Zolotukhin and T. Hämäläinen, "Detection of anomalous HTTP requests based on advanced n-gram model and clustering techniques," in *Proceedings of the 6th Conference on Internet of Things and Smart Spaces*. St. Petersburg, Russia: Springer, 2013, pp. 371–382.

[24] D. Android, "Intents and intent filters," 2019. [Online]. Available: https://developer.android.com/guide/components/intents-filters

[25] ——, "App manifest overview," 2021. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro

[26] A. Desnos, "AndroGuard: Reverse engineering, malware and goodware analysis of Android applications," 2015. [Online]. Available: https://github.com/androguard/androguard/

[27] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*. Austin, TX, USA: ACM, 2016, p. 468–471.

[28] StatCounter, *Mobile Android Version Market Share Worldwide*, 2021. [Online]. Available: https://gs.statcounter.com/android-version-market-share/mobile/worldwide/#monthly-202008-202108

[29] S. Hollister, "The original barcode scanner app, seemingly mistaken for malware, is getting review-bombed," 2021. [Online]. Available: https://www.theverge.com/2021/2/9/22275024/android-barcode-scanner-app-zxing-malware-confusion-negative-reviews

[30] Cordova, "Inappbrowser plugin," 2021. [Online]. Available: https://cordova.apache.org/docs/en/10.x/reference/cordova-plugin-inappbrowser/

[31] A. Arora, S. K. Peddoju, and M. Conti, "PermPair: Android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2020.

[32] 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 Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[33] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective Android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.

[34] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "DroidScribe: Classifying Android malware based on runtime behavior," in *Proceedings of the Security and Privacy Workshops*. San Jose, CA, USA: IEEE, 2016, pp. 252–261.

[35] R. de Souza Polisciuc, L. C. Albini, A. Grégio, and L. C. Bona, "Análise de aplicativos no Android utilizando traços de execução," in *Proceedings of the Brazilian Symposium on Information and Computational Systems Security*. Petrópolis, RJ, Brazil: SBC, 2020.

[36] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu, "SAMADroid: A novel 3-level hybrid malware detection model for Android operating system," *IEEE Access*, vol. 6, pp. 4321–4339, 2018.