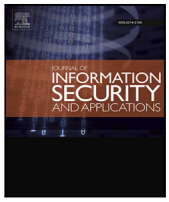
Journal of Information Security and Applications 80 (2024) 103691

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa



Enhancing android malware detection explainability through function call graph APIs

Diego Soi *, Alessandro Sanna, Davide Maiorca, Giorgio Giacinto

Department of Electrical and Electronic Engineering, University of Cagliari, Piazza d'Armi, Cagliari, 09123, Italy

ARTICLE INFO

Keywords:
Malware analysis
Deep learning
Explainability
Android

ABSTRACT

Nowadays, mobile devices are massively used in everyday activities. Thus, they contain sensitive data targeted by threat actors like bank accounts and personal information. Through the years, Machine Learning approaches have been proposed to identify malicious Android applications, but recent research highlights the need for better explanations for model decisions, as existing ones may not be related to the app's malicious functionalities.

This paper proposes an explainable approach based on static analysis to detect Android malware. The novelty lies in the specific analysis conducted to select and extract the features (i.e., APIs taken from the DEX Call Graph) that immediately provide meaningful explanations of the model functionality, thus allowing a significant correlation of the malware behavior with its family. Moreover, since we contain the number and type of features, the distinct impacts of each one appear more evident. The attained results show that it is possible to reach comparable results (in terms of accuracy) to existing state-of-the-art models while providing easy-to-understand explanations, which may yield significant insights into the malicious functionalities of the samples.

1. Introduction

Android OS is the main operating system for mobile devices. According to Stat Counter [1], 70.93% of the devices sold in 2022 were Android-based. Modern devices are not limited to basic operations, like messages and phone calls; they can be employed as a key for MFA (Multi-Factor Authentication) purposes, such as to unlock a car or a safe deposit box or to access bank or work accounts. Hence, mobile malicious software poses a great risk to users' security and privacy. Recent reports show that Android was the target of choice for more than 33 million in new malware samples [2,3].

This threat is often mitigated through anti-malware systems. Modern research on Machine Learning and Deep Learning showed that it is possible to accurately discriminate malicious samples from benign ones [4–8]. Nevertheless, a critical challenge that research has to address is the increasing variability, in terms of behavior, of malicious samples. For example, banking malware, which relies on Command&Control servers, significantly differs from ransomware, which directly encrypts files on the system's memory or locks the screen, making the smartphone unusable. This is especially evident when analyzing the decompiled code of such applications and examining the related APIs. These vast differences between malware families might lead to the extraction of a huge quantity of different features (e.g. the applications' sizes, the number of resources within the APK, etc.), which may be less relevant for detection purposes. As it is often done when dealing with this kind of ML application, the burden of selecting the relevant characteristics is left to the underlying machine-learning system.

This aspect may baffle analysts, who often believe that high detection accuracy scores are due to an effective *understanding* of the malicious characteristics of the apps.

Research has been focusing on *interpretability* to understand better the decisions taken by machine-learning classifiers. In particular, the focus is mostly on deep learning, which is the most complex technique within ML due to the poor understanding of what happens at the hidden layers that may constitute the network. We refer to techniques capable of associating specific features with *scores*, representing their influence in deciding the related class. Previous works have focused on evaluating popular state-of-the-art systems, showing that features relevant to the classifier are often not representative of the functionalities of the sample. The reason for these often disappointing results is that most systems, as underlined previously, employ feature sets that do not clearly encapsulate the behavior of malicious samples.

* Corresponding author.
E-mail addresses: diego.soi@unica.it (D. Soi), alessandro.sanna96@unica.it (A. Sanna), davide.maiorca@unica.it (D. Maiorca), giacinto@unica.it (G. Giacinto).

https://doi.org/10.1016/j.jisa.2023.103691

Available online 2 January 2024

Table 1
Comparison between Java and equivalent DEX and SMALI code.

Java	DEX	SMALI
`int x = 42`	`13 00 2A 00`	`const/16 v0, 42`

As a consequence, interpretability techniques fail to precisely identify elements that are responsible for maliciousness.

For this reason, in this paper, we aim to advance state of the art by proposing three main contributions: (i) the careful selection of features (i.e. set of critical API calls) that can be effectively used to explain complex models, such as deep-learning classifiers, and to gain valuable insight both about the malicious characteristics of malware samples and the impact of each feature on the classifier's decision; (ii) the design of a deep learning model able to classify Android applications starting from the extraction of the Function Call Graph (FCG) obtained from the static analysis of the Android packages (APK) and its subsequent scan to select the list of APIs; (iii) the enforcement of a SHAP-based explainable model to understand what the malware is capable of doing (local explainability) and summarize the actions of samples that belong to the same family (global explainability);

We evaluated our model by considering over 40,000 malicious and benign samples extracted from various resources. The attained results show that API calls can effectively provide reasonable explanations of detections while keeping a very good accuracy and false positive rate. We additionally perform an experimental evaluation to assess the impact of concept and time drift on our system.

This work is structured as follows:

- We analyze the current State of the Art on detection and explainability applied to Android Malware (Section 2).
- We provide a general technical background (Section 3).
- We illustrate a Deep Learning model to identify malicious samples based on features statically extracted, and we include an explainability stage that lists the basic operations performed for added clarity (Section 4).
- We perform an experimental evaluation of the whole model on a consistent dataset (Section 5).
- We discuss this work's contributions and limitations, giving some insight into future research directions (Section 6).

2. Related work

This section outlines the state-of-the-art strategies to understand what has been achieved in Android Malware detection and what this work adds to research. The approaches employed over the years differ by analysis type, extracted features, chosen classifiers, and, possibly, the explainability techniques that may be applied.

We can trace three main types of malware analysis approaches: (i) *static*, (ii) *dynamic*, and (iii) *hybrid*. Static analysis studies, in the case of Android applications, the APK by parsing the Android Manifest, the DEX file, or the SMALI code, a human-readable way to write DEX machine code, as shown in Table 1.

This way, the system can extract features like API calls, instruction opcodes, permissions, and activities. Conversely, dynamic approaches require the execution of the application in a controlled environment to monitor API calls, network flow, and other kinds of run-time features. They can counteract obfuscation,[1] but it is time and resource-intensive. Finally, the hybrid-based analysis combines both static and dynamic ones.

Regarding classical ML models, Scalas et al. employed a Random Forest that is useful for multiclass classification problems [9]; Han et al. evaluate the application of SVM to API calls represented by a tuple `(name, arguments, returned type)` extracted from APKs [10]; Ref. [11], instead, extract permissions, SMALI size, and permissions rate (i.e. the ratio between the total number of permissions required and the SMALI size) applying an SVM model.

Concerning DL, the main models employed in Android Malware analysis can be summarized as follows:

- **Recurrent Neural Networks/Long Short-Term memory**: they focus on the sequentiality of operations, e.g. API calls, done by the application [8,12,13];
- **Graph Neural Networks**: they avail of a graph to represent the features, which is then fed to the network for classification [14,15];
- **Convolutional Neural Networks** aim to represent the features as a matrix. In this case, the features, such as API calls, permissions, activities, and network packets, can be embedded with different techniques [4–7].

Alternative methods employ different networks [16] for each feature, which allows them to be retrained separately whenever necessary.

Regarding explainability, Drebin [17] is one of the first explainable approaches applied to classical machine learning for Android Malware detection. Specifically, the authors extracted the weights of the features involved during classification. In this way, they could list the top k features by weights and explain the samples. Other works, instead, evaluate the possibility of leveraging gradient-based techniques to classical ML model (specifically, on Support Vector Machines): Gradient, Gradient*Input, and Integrated gradients, which may be useful not only to explain models [18,19] but also to assess the robustness of ML against adversarial attacks [20].

Other techniques applied to Android malware analysis with Deep Learning mainly depend on the type of model used. Kinkead et al. [21] use the LIME (Local Interpretable Model-agnostic Explanations) model to produce relevance scores for all the features employed in a 1D CNN highlighting which is the activation of each program part. Some employ visualization approaches like GRAD-CAM to produce heatmaps highlighting bytecode relevant for the decision [22], others an attention mechanism to a Multilayer Perceptron that uses as features API calls and permissions [23] denoting how a general overview of an application functionality can be achieved.

Additionally, recent approaches employed Transformer-based models [24]. Ullah et al. [25] applied BERT (Bidirectional Encoder Representations from Transformers) to extract features from dynamically extracted network traffic applying SHAP to explain how the features contribute to malware or benign labeling, while Jo et al. [26] employed Visual Transformers on image representations of dex code using the intrinsic attention mechanism to produce a relevance score for each feature (i.e. the pixels of the image). However, they introduced the possibility of automatically retracing the code starting from the highlighted sections of the heatmaps.

The motivation for this work is two-folded. Firstly, the features employed in other state-of-the-art systems are, in most cases, either unrelated to the application's functionality or inflated due to the diverse kinds of malware to identify. This may mask the distinct impact of each feature on the classification, resulting in explanations that lack clarity and relevance to the sample being analyzed. Secondly, while visual explanations, as demonstrated in Refs. [22,26], can be beneficial, they pose challenges for the analyst in terms of interpretability. Even though automatic approaches may be employed, the uncertainties and complexities in studying in depth the underlying code, starting from a heatmap (see Fig. 1), to understand the degree of relevance of the highlighted sections, remain.

In contrast, this work aims to introduce an explainable classification approach that relies only on a specific type of feature, namely API calls, which exhibit a strong correlation with critical packages, as detailed in subsequent sections. This approach allows us to achieve both good

[1] Obfuscation is a technique to create source code difficult to understand for humans or computers.

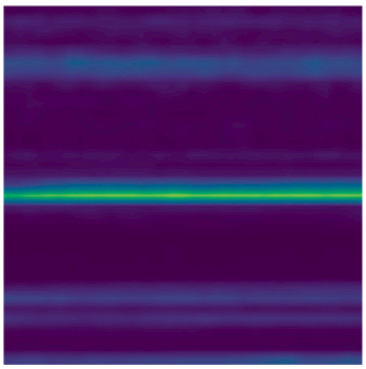


Fig. 1. Example of visual explanation using GradCAM technique on a CNN approach [22]. The central green line is highlighted because it is the most relevant part of the bytecode for the classification.

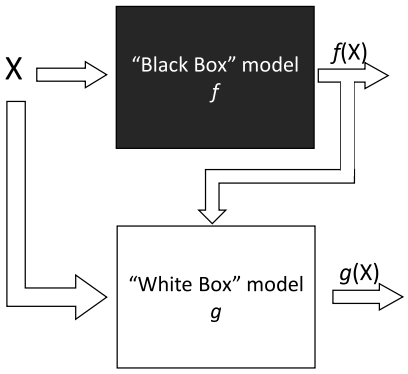


Fig. 2. Model distillation schema. The distilled model takes as input features X, the original model's parameters, and the output of each sample for training the model $g(X)$.

performance and a clear understanding of the functional operations performed by an application.

In Section 5.4, a full comparison with some of the approaches presented here [11,22,23,26] is done concerning both classification performance and the different kind of explanations that state-of-the-art systems proposed (i.e. visualization and text-based).

3. Background

This section discusses the technical background necessary to read and understand this paper. In particular, we provide key concepts of Android applications and Explainability.

3.1. Android

Android is an operating system for mobile devices developed by Google, based on the Linux kernel, and available as free and open-source software. For that reason, it has been widely adopted by different mobile vendors that adapt the Operating System to their necessities, maintaining the core unvaried.

3.1.1. APK

Android applications are distributed as APK files [27], that are akin to a ZIP archive that contains all the resources required by the devices to install and execute the application: (*i*) `lib`, which is a directory holding the libraries for each different CPU; (*ii*) DEX[2] file, that contains the actual app code executed by ART (Android Runtime); (*iii*) `assets`, and `res` directories to save external and raw resources; (*iv*) `AndroidManifest.xml`, which holds data such as *permissions* (i.e. the rights required to use restricted APIs), *services*, *intents*, and *components*.

3.1.2. Entry points

In Android OS, the entry points of an app are the ways in which either the system or the user can interact with the application. It is fundamental to understand what they are to get an insight into how Android apps can start their execution. These are primarily categorized into four distinct groups, defined in the `AndroidManifest.xml` file [28]. The **activities** facilitates user interaction through the UI; the **services** execute background actions[3]; **broadcast receivers** enforce a publish–subscribe pattern enabling the communication between the app and the underlying system; **content providers** manage resources, facilitating data access.

[2] Dalvik Executable Format: https://source.android.com/docs/core/runtime/dex-format

[3] *Foreground services* - noticeable to the user; *Background services* - not directly noticed to the user; *Bound services* - to implement components interactions

3.2. Explainability

Explainability has a crucial role in machine learning (ML) because it is the extent to which the mechanisms inside the ML models can be explained in human terms [29]. In some cases, the term *explainability* is confused with *interpretability*, which, on the contrary, is the extent to which a human can predict a model's result.

A learning model can be *intrinsically explainable* when it is explainable by design (e.g. small decision trees), or *post-hoc explainable* when further processing is needed.

Various approaches to explainability have been proposed in the literature [30], typically falling into three main categories:

- **Visualization** approaches whose common representation are heatmaps that highlight the most relevant features. They differ in the generation methodology: *perturbation-based* by perturbing the input to verify how the model prediction changes; *backpropagation-based* that analyzes the gradients from output to input during the training phase[4];
- **Model Distillation**, whose goal is to generate a model capable of mimicking the behavior of the original one (see Fig. 2). SHAP (*SHAPley additive explanations*) methodology, first introduced in 2017 [31], is one of the most representative examples. SHAP values are *the mean marginal contribution of each feature across all possible values in the feature space*, to measure the relevance of the features used in the model;
- **Model Intrinsic** approaches enforce the use of intrinsic characteristics of the model. An example is the *Attention Mechanism* in which the weights of each unit are extracted to define the importance of a feature.

4. Methodology

This section presents an overview of the methodology applied during this work with respect to chosen classification and explainability approaches. In particular, the goal is to develop a model that is both effective in identifying the malicious applications among the benign ones and explainable in the sense that the intrinsic malicious functionalities should be acknowledged. To achieve this, we need to work with human-understandable features intrinsically correlated with the behavioral characteristics of the application.

Classification. As explained in Section 3.1, an Android application contains the DEX code, which holds classes and APIs called by the Android Runtime during execution. Starting from this, our approach,

[4] Examples are Class Activation Maps (CAM) and Gradient CAM (Grad-CAM) applied to CNNs models.

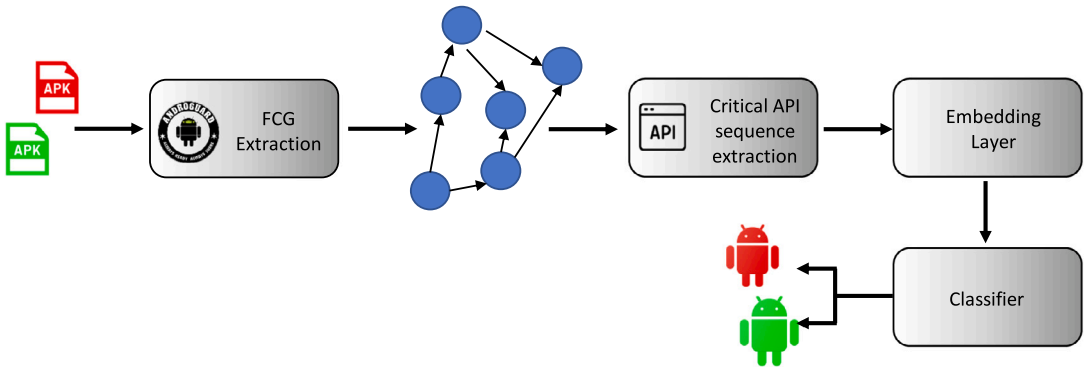


Fig. 3. Overall classification schema which consists of different steps: Function Call Graph extraction, Critical API extraction, Embedding Layer, and the actual classifier.

depicted in Fig. 3, is to analyze the APK to extract the so-called **Function Call Graph (FCG)**, whose aim is to represent in a graph all API calls that are performed within the application code. By scanning that graph, a list of APIs belonging to critical packages is pulled out. Then, an embedding layer is applied to obtain suitable features for the subsequent CNN network to decide whether the sample is benign or malicious.

The choice to extract only APIs statically derives from the fact that these features are immediately attributable to the behavior of the samples. Therefore, the successive explainability methodology may deal with meaningful features, producing meaningful explanations. On the contrary, features such as the permissions from the AndroidManifest or the opcodes from SMALI analysis, although strictly connected to the sample execution purposes, are not as significant as the APIs.

Now that the general idea is clear, a detailed illustration of each step is necessary.

Function call graph (FCG) extraction. The FCG, as the name suggests, is a graph of all the API calls found in the DEX file. Specifically, a node is an API, while edges represent the relationship between them, i.e. a call. The idea is to scan the graph, starting from the apk entry points, so that we can trace the calls execution path.

Various techniques exist in literature to extract dynamic or static function calls. In the former, the sample is run in a controlled environment to trace all the API calls, while in the second case, the code is scanned. In this work, we enforce a static analysis using the Androguard tool to produce the FCG. This way, we are sure to produce the entire call graph without leaving out any relevant API call, which, in the case of dynamic analysis, may not be reached throughout the application execution because of the limited run time. Additionally, dynamic approaches are hindered by anti-sandboxing and anti-debugging techniques used in malware; these techniques modify the program's behavior at run time and, consequentially, the sequence of performed API calls.

Another static method involves SMALI code analysis by looking at the `invoke-*` opcodes [9,32]. However, this kind of systematic approach may require a huge analysis of the SMALI files, which may be numerous in the case of larger applications.

Critical API calls. By scanning the FCG, only API calls belonging to a list of critical packages specified in Table 2 are extracted. As described in [33], critical APIs are the ones that are controlled by run-time permissions, which give the application access to restricted data or instructions. Two examples are `android.telephony`, which contains APIs to monitor basic phone information (deviceID, IMEI, or SIM data) or to perform some actions (e.g. sending SMS), and `android.net`, which provides APIs that can manage network connectivity.

The choice is driven by two considerations: (i) minimizing the number of features, thereby reducing processing them; (ii) considering the entirety of API calls, including Java's, may not be representative for distinguishing between APK categories.

Table 2
Critical APIs list in which Android modules and their description is depicted.

Android module	Description
`android.accounts`	User's account management APIs
`android.app`	Application management APIs
`android.bluetooth`	Bluetooth management APIs
`android.content`	Publish, access and share device data APIs
`android.location`	Geolocation management APIs
`android.media`	Device media interface management APIs
`android.net`	Network access management APIs
`android.nfc`	NFC access management APIs
`android.provider`	Content provider access management APIs
`android.telecom`	Call management APIs
`android.telephony`	Monitor basic phone information APIs

Embedding layer. We assume the list of critical APIs extracted in the previous stage as a sentence, like in Natural Language Processing (NLP). Two procedures are needed to represent features in a form suitable for a CNN:

- **TF-IDF**[5]: we select the 20 most relevant strings. As explained later in Section 5.2, this number has been chosen because it represents the best compromise between performance speed and results validity. This is critical because each APK has its own list of APIs with varying sizes; the classifier, however, needs inputs of uniform length.
- **Word2Vec Embedding**: we derive an embedding vector for each API. A matrix of 20×25 is then produced. In practice, the context of words is considered and "remembered" during classification [34].

As stated previously, the methodology presented takes advantage of the research in NLP. Therefore, the adoption of TF-IDF and Word2Vec embedding is natural since it is a standard de facto for this kind of application [35,36]. Indeed, the first process considers only the most relevant words (i.e. APIs) in the list of critical calls extracted in the previous stage, while Word2Vec is needed to build an embedding between the selected APIs to produce the corresponding word vector, a vectorial representation of that API.

Classifier. Fig. 4 shows the classifier's scheme employed in this work. The core is a **CNN** consisting of 1D convolutional layers ReLu activated. We need this kind of convolution since the network must process each feature (API) alone so that no correlation between APIs is learned. A dropout layer is inserted to reduce overfitting that may arise. Then, **1D max pooling** and **Flatten** layers are required to produce a valuable input for the last fully connected network (i.e. the last three Dense layers) that can generate the classification score with a final sigmoid unit. Eventually, a threshold of 0.5 is chosen to weight the two classes

[5] TF-IDF stands for Term Frequency-Inverse Document Frequency.

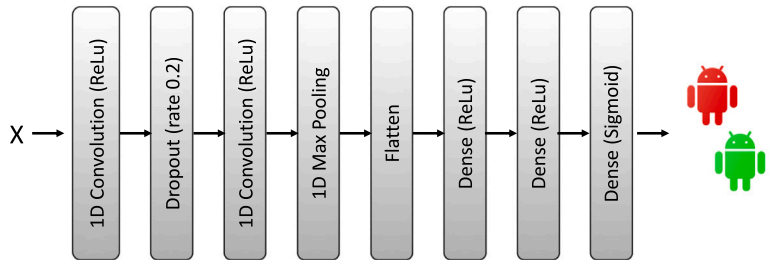


Fig. 4. Classifier Schema with all its layers. See Table 3 for further clarifications.

Table 3
Deep Neural Networks parameters. Layer field represents the type of Keras layer; Padding and Stride are two of the main attributes to specify in the case of convolutional layers; Filters/Units/Pool size specifies the number of filters or units in the case of different layers; and Output shape is the shape of the result produces by the layer.

Layer (Activation)	Padding	Stride	Filters/Units/Pool size	Output shape
Conv1D (ReLu)	None	1	32	(None, 20, 32)
Dropout (rate 0.2)	–	–	–	(None, 20, 32)
Conv1D (ReLu)	None	1	20	(None, 20, 20)
MaxPool1D (–)	None	–	4	(None, 5, 20)
Flatten (–)	–	–	–	(None, 100)
Dense (ReLu)	–	–	80	(None, 80)
Dense (ReLu)	–	–	50	(None, 50)
Dense (ReLu)	–	–	1	(None, 1)

identically and to yield the actual classification label: 0 for benign APKs and 1 for malicious ones (Eq. (1)).

$$label = \begin{cases} 0 & \text{if } score < 0.5 \\ 1 & \text{if } score > 0.5 \end{cases} \quad (1)$$

Explainability. The proposed approach employs a well-known methodology applied in other works [37–39]: the SHAP model. In particular, the surrogate model can compute relevance scores for each feature passed in input to the classifier (i.e. the APIs). The SHAP model produces a matrix of 20 × 25. A sum is then applied row by row to calculate the relevance of each API. SHAP-based explanations are chosen for their simplicity, intuitiveness, and ability to measure only the contribution to the classification without considering the model's performance. In addition, SHAP can be used for both local and global explainability.

5. Experiments

5.1. Dataset

The dataset consists of 48,372 APK samples in total (Table 4). In particular, most malicious samples are from VirusShare datasets,[6] while Androzoo collection[7] is used to obtain benign ones. To label uncategorized samples, VirusTotal APIs are used considering malware, those for which at least one security vendor or sandbox flagged it as malicious. Otherwise, it is considered a benign APK. We use VirusShare APIs and AVClass [40] to associate each extracted malware with its description, indicating the family it belongs to. For brevity, in Table 5, we report only the data about the 10 most popular families out of the total 432. Additionally, we pair these findings with a description of the general threats associated with these samples enumerated in 12 categories in Table 6.

As shown in Fig. 5 and Table 7, the samples' year ranges from 2008 to 2022, with malicious samples being prevalent until 2015, while the number of benign ones is more significant after 2016.

Table 4
Complete dataset specifics. It describes the number and the source of the APKs used in the work.

Description	Androzoo	VirusShare	Tot.
Benign	21,459 (100%)	–	21,459
Malicious	832 (3%)	26,081 (97%)	26,913

Table 5
Top 10 malicious families in the dataset.

Family	Amount
smsreg	3041
oimobi	568
dowgin	512
skymobi	507
smssend	506
gappusin	500
kuguo	458
smspay	385
igexin	338
hiddenapp	330

Table 6
Threat categories in the malicious dataset.

Threat	Amount
grayware	10 537
downloader	3058
virus	1537
rooter	277
ransomware	181
backdoor	171
clicker	140
spyware	79
bot	17
worm	3
dialer	2
hoax	1
unknown	10 910

[6] https://virusshare.com/
[7] https://androzoo.uni.lu/

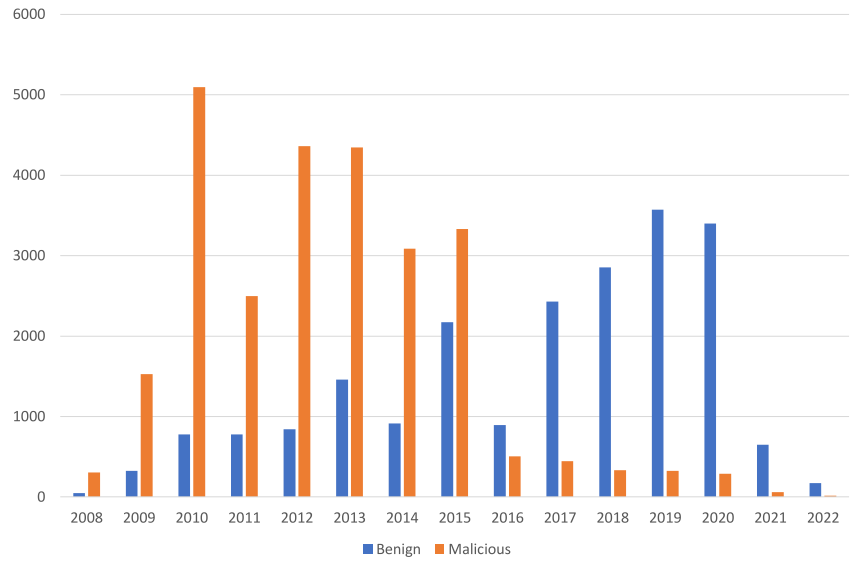


Fig. 5. Distribution of samples per year. The blue bars identify the benign sample, while the orange ones represent the malicious samples.

Table 7
Number of samples per year.

Year	Benign	Malicious
2008	46	213
2009	321	1495
2010	768	4707
2011	777	2229
2012	836	4160
2013	1457	3254
2014	909	2622
2015	2169	2854
2016	892	442
2017	2416	339
2018	2837	263
2019	3566	283
2020	3391	270
2021	650	53
2022	172	13

5.2. Evaluation of classification

We performed several tests in order to evaluate the model used to classify and explain Android APKs. Therefore, in this section, we present the experiments and the corresponding results.

5.2.1. 10-Fold cross-validation

Setup. The dataset is divided into training and test sets (80% and 20%, respectively), and a fold-cross test is performed to evaluate model classification performance over a small dataset. The train set is further split into 10 folds: one is employed as a validation test, while the others train the model. This process is repeated 10 times. The remaining 20% of the dataset will be used only in the test described in the following. The other reason for this test is to choose some fundamental parameters for the classification task: the number of APIs selected by TF-IDF (20 APIs per sample) and the number of elements per API by Word2Vec embedding (25 elements per API). Note that considering fewer APIs per sample would mean a drop in the performance, while more than 20 results in an increase in the computational time.

Results. Results are shown in Table 8. We computed several metrics, including accuracy, precision, and F1-score, to measure the quality of the test classification. As one can notice, the average measures over ten folds are satisfactory since the achieved accuracy is 87.3%±0.63, and the F1-score is 87.3%±0.48. This means that the model can recognize both malware and benign samples since F1 takes into account both precision and recall. In addition, the standard deviation (i.e. the measure of the variation or dispersion inside a set of data) remains low, meaning that over the ten folds, the measures are clustered around the mean.

Table 8
Results over 10 folds. The table shows the average and standard deviation of the measures done to evaluate the performance over 10 folds.

	Acc. (%)	Prec. (%)	F1 (%)	AUC
μ	87.3	87.0	87.0	0.94
σ	0.63	1.68	0.48	0

Table 9
Classification report where Precision, Recall, and F1-score are the measures produced based on the considered class (i.e. Benign or Malicious) while the accuracy is computed over all the test set.

Class	Prec. (%)	Recall (%)	F1 (%)	Support
Benign	87.0	85.0	86.0	4318
Malicious	86.0	88.0	87.0	4563
Acc. (%)			87.0	8881

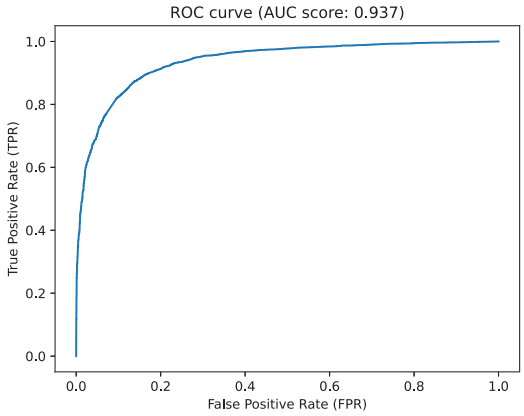


Fig. 6. ROC curve describing the experiments' performance with the whole dataset.

5.2.2. Classification with the whole dataset

Setup. A separate test has been done on the whole dataset by using the complete training test, not divided into folds, to train a newly created model with the same parameters set in the previous test, and the test set (20% of the entire dataset, as described before) to evaluate the performance. The training is done over 8 epochs with a batch size of 20 samples and a validation set composed of 15% of the training set.

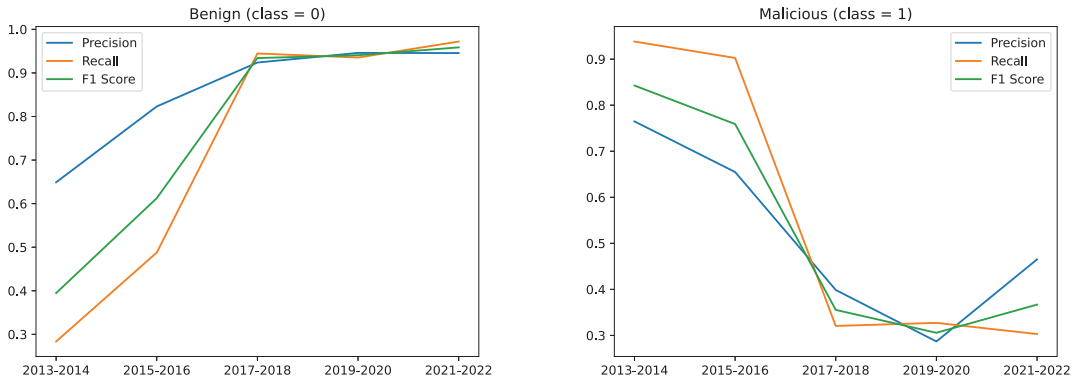


Fig. 7. Time-aware classification test. The figures show the metrics (i.e. precision, recall, and f1 score) computed considering as testing year range the values in the x-axis while the training set consists of samples from 5 years earlier.

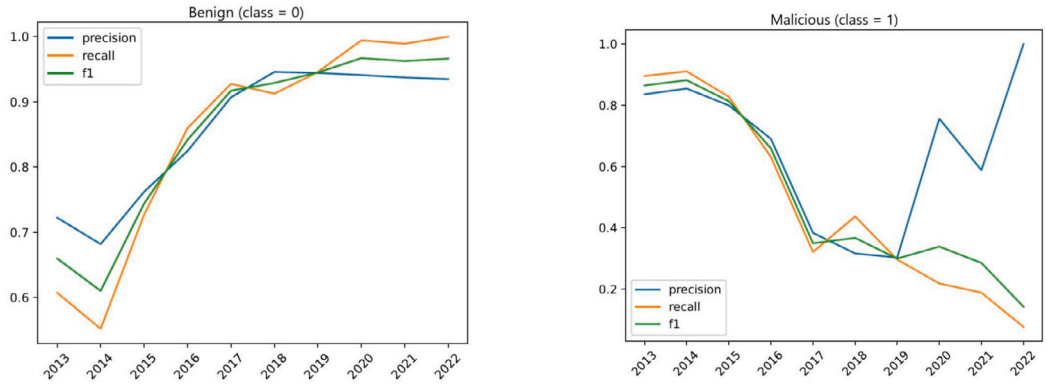


Fig. 8. Sliding window test. The figures show the metrics (i.e. precision, recall, and f1 score) computed considering as testing year the values in the x-axis, while the training set consists of samples from 5 years earlier.

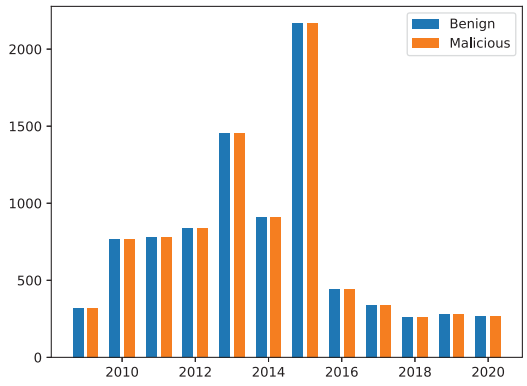


Fig. 9. Enforcing the same malicious-benign rate per analysis year.

Results. Fig. 6 shows the ROC curve from which we can state that $TPR = 0.87$ while $TNR = 0.86$. The ROC considers the trade-off between precision and recall, so it is a more indicative metric than accuracy, considering only how many predictions are correct. In addition to the ROC curve, Table 9 shows the classification report produced to measure precision, recall, and F1-score for both classes, and the overall accuracy reaches about 87.0%.

5.2.3. Time-aware classification

Setup. We carried out four different tests, employing the TESSERACT library [41], in which the samples' development year is considered because different works denoted how temporal biases could affect the dataset, leading to better performance [41,42]. This bias refers to unrealistic evaluations of the samples in the test set due to the wrong integration of future knowledge in the training set; namely, if samples from 2019 are in the training data, the model may learn some features that can only be seen in the future (e.g. API calls introduced only in 2019) simplifying the identification of past data. Conversely, in the case of malware analysis, the objective is to identify future samples starting from past knowledge.

The first test is done by incrementally training the model with recent data. So, starting from a time window of five years (2008–2012), we tested the model on future data (i.e. APK from 2013 and 2014). Then, we re-trained the model, adding data that once were used for testing (i.e. 2008 to 2014 applications), and re-testing it with 2015-2016's samples. That is done until the test consists of 2021-2022's applications.

Results. Fig. 7 shows a degradation of performances over time. Interestingly, however, the benign class presents a rise in performance scores. This can be addressed by collecting more recent data, especially for malware.

5.2.4. Time-aware sliding window classification

Setup. This test has been done to understand how model training is affected by the number of samples per year within the dataset, as shown in Fig. 5. The training set comprises samples in a time window of five years, while the test consists of only the samples from one year next to the last in the training window. For example, if the training window is 08–12, as depicted in Fig. 8, the training samples belong to the development year range 2008–2012, while the test applications are from 2013. Besides, a sliding window of one year is chosen to notice how the results change based on the different testing years.

Results. Fig. 8 shows the results of the sliding window test. The consideration we can make is that if the number of samples per class is low, the results are poor. Let us take the case of malicious class; all

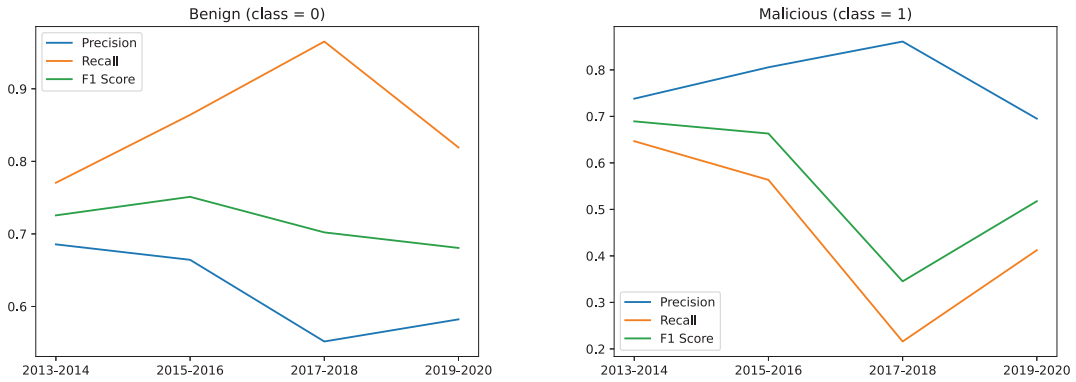


Fig. 10. Time-aware same-distribution classification test. The figures show the metrics (i.e. precision, recall, and f1 score) computed considering as testing year range the values in the x-axis while the train set consists of samples from 5 years earlier.

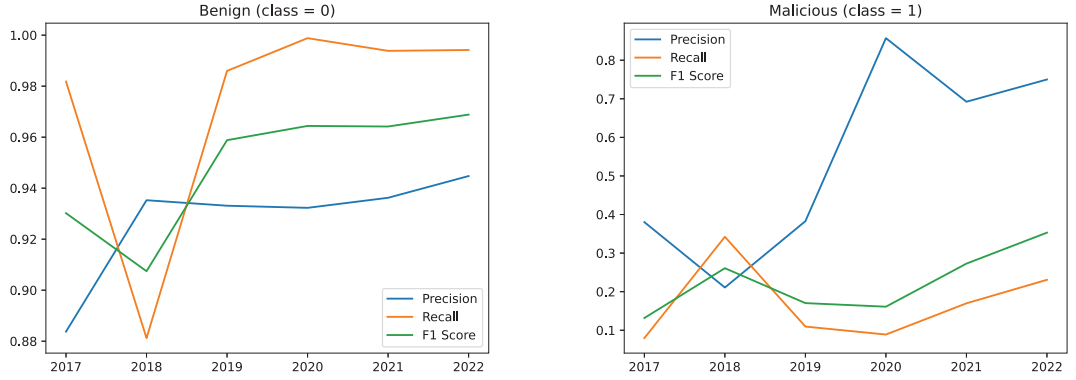


Fig. 11. Time-aware classification from 2016. The figures show the metrics (i.e. precision, recall, and f1 score) computed considering as testing year the values in the x-axis while the train set consists of samples from 1 year earlier.

metrics are significant in the first four ranges (i.e. 2013 to 2016), and then the performance drops. The same results for the benign class; when the samples are lower in number (i.e. 2013 to 2015), the performance are poor, while when the number of sample increases, the computer metrics are better.

These results are coherent with the consideration that we made before; the dataset should be yearly balanced in the number of samples.

5.2.5. Time-aware same-distribution classification

Setup. To address irregular time distribution in our data, we verified our results with an additional experiment concerning a downsampled set considering a yearly balanced data set. Specifically, we take the minimum number of samples in each class each year (e.g., since 2012 has 4160 malicious samples, but only 836 benign ones, we take 836 malicious samples and all the benign ones). Years 2008, 2021, and 2022 had to be cut for lack of sufficient samples in one of the classes. This was done to be sure that the model was not just learning the temporal distribution of the data, albeit it had not explicitly access to it. We train the model in the same fashion as in 5.2.3. Having discarded 2008 data, however, the initial training batch's size is four years (from 2009 to 2012).

Results. Fig. 9 shows the new data distribution. As Fig. 10 shows, the model's performance degrades in the same fashion as the previous tests, showing that the features are the main learning item. However, the difference in performances between the two classes is less severe, which probably can be imputed to the more balanced set. The influence of features towards classification is detailed further in Section 5.3.

5.2.6. Time-aware classification from 2016

Setup. Having noticed that all previous temporal evaluations indicated a sudden negative spike in performances when the year 2016 was selected for testing, we investigated if there could be an event that could have introduced a sudden change in the data distribution. As it turns out, in that year, Android Nougat was released, which featured OpenJDK as the main development kit in lieu of the defunct Apache Harmony.[8] Even if carried out seamlessly at the time, such a change may well have introduced completely new features in data, drastically shifting the data distribution.

Results. As Fig. 11 shows, using data from 2016 and ongoing to perform the same test described in 5.2.3 yields fairly better results, with an increase of overall results in the case of benign class. Nonetheless, due to the malicious data scarcity signaled before, the positive class performed poorly. For this reason, we advise that systems that use features like ours should be retrained using a significant amount of more modern data, considering this significant concept drift.

In light of the time-aware experiments, it is essential to differentiate between *recent* and *unknown* malware. The first refers to samples developed after training time, while the latter encompasses malware that, despite not being recent, has never been encountered by the system in the training stage. Having clarified this, the system performance sensibly drops only if test data falls in the first category due to the drift found in 2016.

5.3. Evaluation of explainability

In this section, we present the experiments and the corresponding results for the explainability stage. Particularly, we focused on local and global explainability of the samples comprising the dataset.

[8] https://shorturl.at/drsLM

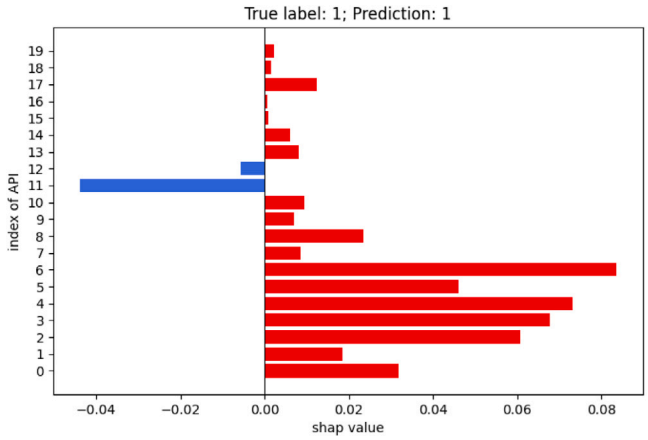


Fig. 12. Bar plot of the SHAP values computed for a malicious sample correctly classified where SHAP values and API index (row in the feature matrix) are shown. Red bars, on the right, are associated with a malicious operation, while blue ones, on the left, are associated with a benign operation.

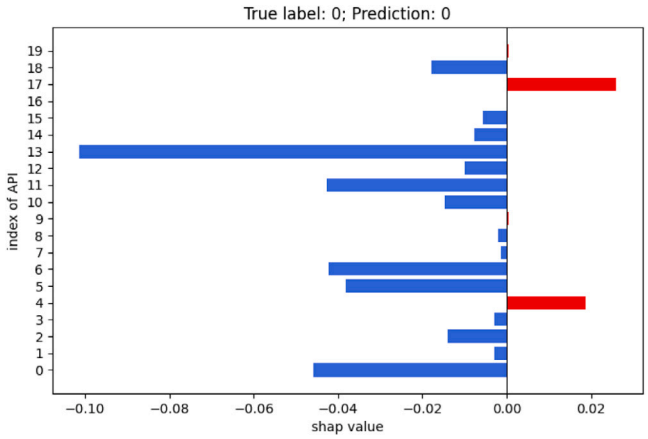


Fig. 13. Bar plot of the SHAP values computed for a benign sample correctly classified where SHAP values and API index (row in the feature matrix) are shown. Red bars, on the right, are associated with a malicious operation while blue ones, on the left, are associated with a benign operation.

Table 10
Top 5 APIs of the sample in Fig. 12 based on SHAP value. Bold APIs are those related to a malicious operation.

API index	API	SHAP ↑
6	android.content.res.AssetManager-open	0.085
4	**android.net.wifi.WifiInfo-getMacAddress**	**0.073**
3	**android.telephony.gsm.GsmCellLocation-getCid**	**0.067**
2	**android.telephony.gsm.GsmCellLocation-getLac**	**0.061**
5	**android.net.wifi.WifiManager-getConnectionInfo**	**0.046**

Table 11
Top 5 APIs of the sample in Fig. 13 based on SHAP value.

API index	API	SHAP ↑
13	android.content.res.Resources-getXml	−0.085
0	android.content.res.ColorStateList-isStateFul	−0.046
11	android.content.res.Resources-getDrawable	−0.043
6	android.content.res.Resources-getValue	−0.043
5	android.content.res.TypedArray-getColor	−0.038

5.3.1. Local explainability

Setup. As described in Section 3.2, local explainability is intended to explain a single sample. Hence, this test aims to ascertain the system's capability to generate distinct explanations by associating each sample (one malicious and one benign) with the most representative APIs. We present the results through bar plots and tables. The first shows the SHAP values for each API extracted for the corresponding sample. Red bars correspond to positive SHAP values associated with APIs relevant to malicious (class "1") classification. Vice versa, blue bars indicate negative SHAP values, therefore associated with APIs that are more interesting for benign (class "0") classification. The tables show the top 5 APIs with the corresponding SHAP values sorted in descending order of their absolute value. This is to know the most relevant APIs and, derivatively, the most significant operations performed by the sample.

Results. Fig. 12 and Table 10 show the results for a spyware **malicious application**.[9] SHAP values are mostly positive: the classifier considers the associated features to indicate maliciousness. The top 5 APIs are critical operations and among them, some can retrieve sensitive information about cell location employed by the smartphone (`getLac` and `getCid` which retrieve cell ID and location), and connectivity state (`getConnectionInfo`) that, probably, is fundamental because the malware needs it to connect on the Internet.

Fig. 13 and Table 11 show the results for a **benign application**.[10] In this case, one can notice immediately that features extracted for the sample shift the classification score closer to the benign because blue bars are prevalent. Table 11 shows that the operations performed are not so critical: they access application resources from the content provider without leaking valuable information.

5.3.2. Multi-class global explainability

Setup. This test aimed to understand which are the most discriminative APIs from a global perspective. Therefore, we took only malicious samples from the test set to highlight behavior that can be considered malicious. More than one sample can contain the same APIs, so SHAP values are computed by averaging the values obtained for each APK in the test.

Results. Table 12 refers to the top 10 APIs based on SHAP value. There are APIs related to internet connectivity checks, SMS sending functionality, and WiFi information. There are also APIs not related to maliciousness because test malware samples belong to heterogeneous families, so samples are quite varied in their functionalities.

5.3.3. Same-class global explainability

Setup. To counteract the variety of samples in the test set, we performed a second test related to global explainability taking samples from the same family from another set different than training and test. The set consists of ten samples from the LockerPin family taken from the CICAndMal2017 dataset.[11] In this way, we could evaluate the model classification on samples the model did not see and if it can correlate samples and explain them.

Results. Table 13 shows explanations for three samples belonging to the LockerPin ransomware family. From it, we can state that is possible to find a correlation between considered applications since the top 5 APIs are similar:

- `lockNow()` to lock immediately the device;
- `resetPassword()` to change the unlocking PIN/password;
- `isAdminActive()` to check if the administrative component is enabled.

[9] MD5: 65f37e366b9ba5000eba50834f87fed9
Name: com.mobistartapp.win7imulator.apk (spyware)

[10] MD5: ed8b7e9ad3f00dd02d7e1c755de98393
Name: com.resultsdirect.eventsential.branded.cues

[11] https://www.unb.ca/cic/datasets/andmal2017.html

Table 12
Top 10 APIs for the malicious samples in the test set. Bold APIs are those related to a malicious operation.

API	SHAP ↑
android.net.wifi.WifiInfo-getMacAddress	**0.157**
android.content.res.TypedArray-getFloat	−0.137
android.app.admin.DevicePolicyManager-isDeviceOwnerApp	**0.130**
android.content.res.Resources-getIdentifier	−0.129
android.telephony.gemini.GeminiSmsManager-sendDataMessageGemini	**0.128**
android.telephony.gsm.SmsManager-getDisplayOriginatingAddress	**0.123**
android.content.pm.PackageManager-setComponentEnabledSetting	0.118
android.content.res.XmlResourceParser-getAttributeFloatValue	−0.113
android.content.res.AssetManager-openXmlResourceParser	−0.118
android.net.wifi.wifiManager-getConnectionInfo	**0.109**

Table 13
Approach applied to three malware from the LockerPin family. Bold APIs are those related to a malicious operation.

MD5	API	SHAP ↑
Locker 1[a]	**android.app.admin.DevicePolicyManager-isAdminActive**	**0.058**
	android.app.admin.DeviceAdminReceiver-onPasswordFailed	0.055
	android.content.res.AssetManager-open	0.047
	android.app.admin.DevicePolicyManager-resetPassword	**0.044**
	android.app.admin.DevicePolicyManager-lockNow	**0.037**
Locker 2[b]	android.content.pm.PackageManager-getApplicationIcon	0.095
	android.app.admin.DevicePolicyManager-lockNow	**0.059**
	android.app.admin.DeviceAdminReceiver-onPasswordFailed	0.047
	android.app.admin.DevicePolicyManager-resetPassword	**0.045**
	android.content.res.AssetManager-open	0.028
Locker 3[c]	**android.app.admin.DevicePolicyManager-isAdminActive**	**0.062**
	android.content.pm.PackageManager-getApplicationIcon	0.061
	android.content.pm.PackageManager-setComponentEnabledSetting	0.060
	android.content.res.AssetManager-open	**0.043**
	android.app.admin.DevicePolicyManager-resetPassword	**0.038**

[a] MD5: a063292d8667cf3d83ff9365dfb8650a
[b] MD5: d335f22545505783e473b42259253d36
[c] MD5: aa2be7fd72752dffa89fb903cb70392e

5.4. Comparison with other approaches

A comparison with four other approaches is discussed to understand the pros and cons of the proposed model.

Akbar et al. [11] use a classical machine-learning approach to classify Android applications. In particular, the employed statically-extracted features are diverse (i.e. permissions, SMALI size, and permission rate). The classifier employed is an SVM model trained with 10 000 applications in total. In terms of performance, the proposed approach is comparable (87% vs 89% of accuracy), without considering the time-aware dataset as in [11]. Even though explainability is not a key point in the compared work, this has been done to underline that the kind of features employed are not so easily correlated with the app behavior as stated in previous sections.

Iadarola et al. [22] examine the possibility of using a CNN on DEX files represented as black/white images. Also, in this case, the accuracy is higher (87% vs 95% of accuracy). Concerning explainability, they employ GRAD-CAM to obtain heatmaps highlighting the most relevant parts of the dex code for the classification. The problem is that it is not as immediate as our explainable approach to understanding the functionalities of a sample because the analyst should always examine the code.

An interesting comparison can be made with a similar approach to [22], proposed by Jo et al. [26]. As mentioned in Section 2, the approach uses novelties on transformers-based models since they applied a Vision Transformer (ViT) on image representations of dex files. One of the main results is the possibility of using the intrinsic attention mechanism to produce a heatmap similar to the one made by GradCAM. However, to counteract the difficulties related to heatmaps' analysis, they implemented an automatic reversing methodology to understand which part of the code is related to the "critical" pixels. The approach seems quite good, but the main problems remain the resources required to do that and the uncertainty of the pixel-to-dex method, which may result in not providing sufficient details for the application at hand. Therefore, the same consideration done in the previous comparison can be made: the approach presented in this work is much more intuitive and straightforward.

At last, we performed a practical comparison to highlight the differences between our method and Xmal [23] by testing the two approaches with samples belonging to different malware families. The approach is akin to ours as it aims to offer human-understandable explanations regarding malware functionality. However, a notable distinction lies in the features employed (i.e. both API calls and permissions). A comparative analysis is warranted to discern the differences and similarities in the produced explanations that are generated in different ways for both the kind of method and the features applied.

Table 14 summarizes the results for the 4 most representative malware.

For the LockerPin sample,[12] our approach lists a set of APIs, among which `lockNow`, `resetPassword`, `onEnable`, and `getWho` are associated with the malicious behavior of a LockerPin malware. Specifically, they are employed to obtain administrative privileges, reset the password and lock the screen immediately preventing the user from accessing the device. On the other hand, Xmal can only retrieve one relevant permission: `WAKE_LOCK`. This allows to keep the smartphone alive even though the device appears asleep. This is reasonable since Xmal is not extracting any feature related to locking capabilities, be it an API or permission.

[12] MD5: 4BD33BA8957168DCCBEADBBEA45C6843

Table 14
Comparison between our approach and Xmal with four samples. Bold features represent those that are more associated with true malicious behavior, while Xmal permissions are highlighted in bold italics to distinguish them from API calls clearly. All APIs within the left column are in `android` package.

	Our approach ↑	Xmal ↑
LockerPin	**app.admin.deviceAdminReceiver-onEnabled** app.admin.deviceAdminReceiver-onReceive **app.admin.devicePolicyManager-lockNow** content.pm.packageManager-getLaunchIntentForPackage **app.admin.deviceAdminReceiver-onDisabled** app.admin.deviceAdminReceiver-onDisableRequested **app.admin.deviceAdminReceiver-getWho** app.admin.deviceAdminReceiver-onPasswordFailed content.res.resources-getConfiguration **app.admin.devicePolicyManager-resetPassword**	android.permission.READ_PHONE_STATE app.NotificationManager-cancel content.pm.PackageManager-checkPermission content.ContentResolver-query android.permission.ACCESS_NETWORK_STATE android.permission.INTERNET Ljava.lang.Runtime-exec *android.permission.WAKE_LOCK* Ljava.net.URL-openConnection
MazarBot	content.pm.packageManager-getLaunchIntentForPackage content.res.resources-getStringArray content.pm.packageManager-getInstalledApplications **content.ContentResolver-query** content.pm.packageManager-setComponentEnabledSetting **app.admin.deviceAdminReceiver-onPasswordSucceeded** **app.admin.devicePolicyManager-wipeData** **app.admin.deviceAdminReceiver-onEnabled** **app.admin.deviceAdminReceiver-onDisabled** content.pm.signature-toByteArray	*android.permission.READ_PHONE_STATE* app.NotificationManager-cancel **content.ContentResolver-query** *android.permission.INTERNET* android.permission.ACCESS_NETWORK_STATE **telephony.TelephonyManager-getDeviceId**
BeanBot	content.pm.binding-attachInterface app.enterprise.knoxCustom.customDevicemanager-getInstance **telephony.smsMessage-getDisplayMessageBody** app.admin.devicePolicyManager-isDeviceOwnerApp **telephony.telephonyManager-getDeviceId** android.media.session.mediaSession-setMediaButtonReceiver **telephony.gsm.smsManager-getDefault** **telephony.gsm.smsManager-sendTextMessage** android.content.pm.resolveInfo-getIconResource **app.enterprise.enterpriseDeviceManager-getFirewall**	android.permission.READ_PHONE_STATE Ljava.net.HttpURLConnection-getResponseCode **content.ContentResolver-query** android.permission.ACCESS_NETWORK_STATE **location.LocationManager-requestLocationUpdates** android.permission.INTERNET
Fake Installer	app.enterprise.multiUser.multiUserManager-getInstance **location.LocationManager-getLastKnownLocation** **telephony.telephonyManager-getDeviceId** content.pm.packageManager-getPackageInfo app.admin.devicePolicyManager-isDeviceOwnerApp **telephony.gsm.gsmCellLocation-getCid** **net.wifi.wifiManager-setWifiEnabled** app.enterprise.deviceSettings.deviceSettingsPolicy-getInstance **app.enterprise.enterpriseDeviceManager-getFirewall**	*android.permission.READ_PHONE_STATE* Ljava.net.HttpURLConnection-getResponseCode *android.permission.INTERNET* android.permission.ACCESS_NETWORK_STATE android.permission.SEND_SMS

The MazarBot malware[13] can obtain administrative privileges, get some information about the smartphone and wipe all data from the device. In the case of our approach, the explanations can detect some of the malicious behavior of the sample: *(i)* `query` that queries a URI, *(ii)* `onEnabled`, `onDisabled`, `onPasswordSucceded` that are associated with admin functionalities, and *(iii)* `wipeData` to delete all data inside the smartphone. Xmal is only listing features that highlight the internet and stealing data-related functionalities.

The other two samples, BeanBot[14] and Fake Installer[15] are two kinds of Trojan that can steal some information about the smartphone, send SMS, and connect to some remote server. The explanations of the two approaches are different because, in the case of Xmal, only some functionalities are listed since permissions are deemed more relevant than other APIs extracted as features. Conversely, in our approach, a list of some APIs can better represent malware capabilities (e.g. `getDeviceId`, `getDispalyMessageBody`, `sendTextMessaege`, `getCid`, etc.).

The four examples just presented lead to an interesting result: one can use both approaches in a complementary fashion since they highlight different features (i.e. API calls and permissions). In this way, the level of malware explainability is enhanced.

6. Conclusions and future works

In this work, we developed a novel methodology for explainable Android malware analysis. We selected API calls as a set of features strictly correlated with malicious behavior, and we performed several

[13] MD5: 1A4A6D135629D56917366D18C99ED316
[14] MD5: 9C1A90860302572A0D86BCF6C6A084EE
[15] MD5: 2CBADCC6A5474687A4F6A7F4183D835C

tests on a dataset of over 40 000 Android applications to check whether the explanations could detail malware behavior. The attained results showed that the approach could detect and explain malicious Android APKs with the set of APIs extracted. The SHAP values calculated on local and global explanations show that APIs constitute a winning choice to understand learning-based systems decisions in Android. In particular, experiments done with applications from the same family (LockerPin ransomware) demonstrate the possibility of stating a correlation between the apps at hand and possibly recognizing the maliciousness of the sample and its family automatically.

Different problems remain open and should be addressed in future works: (a) the classification performance is comparable but not superior to other approaches analyzed in Sections Section 2, and 5.4. Hence, a wider dataset can be used to evaluate classifier strength further. In addition, to address the problems related to the time-aware classification described in the previous sections, a dataset with more recent samples, in particular for the malicious class, should be needed to test the approach against temporal bias further; (b) The set of API extracted is very large. For that reason, an idea to improve classification and explainability would be to select a smaller feature set (APIs belonging to the same packages used in this work) that can represent maliciousness in the applications at hand in an equally discriminative fashion.

CRediT authorship contribution statement

Diego Soi: Conceptualization, Data curation, Software, Validation, Writing – original draft, Writing – review & editing. **Alessandro Sanna:** Software, Visualization, Writing – original draft. **Davide Maiorca:** Conceptualization, Supervision, Writing – review & editing. **Giorgio Giacinto:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

[1] StatCounter-GlobalStats. Mobile operating system market share worldwide. 2022, StatCounter. URL https://gs.statcounter.com/os-market-share/mobile/worldwide. (Online - Accessed 06 December 2022).

[2] AV-ATLAS. Total amount of malware and PUA under android. 2022, URL https://portal.av-atlas.org/malware/statistics. (Online - Accessed 06 December 2022).

[3] Kaspersky. Android mobile security threats. 2022, URL https://www.kaspersky.com/resource-center/threats/mobile. (Online - Accessed 03 November 2022).

[4] Feng R, Chen S, Xie X, Ma L, Meng G, Liu Y, et al. MobiDroid: A performance-sensitive malware detection system on mobile platform. In: 2019 24th international conference on engineering of complex computer systems. 2019, p. 61–70. http://dx.doi.org/10.1109/ICECCS.2019.00014.

[5] Karabey Aksakalli I. Using convolutional neural network for android malware detection. Comput Model New Technol 2019;23:29–35.

[6] Nicheporuk A, Savenko O, Nicheporuk A, Nicheporuk Y. An android malware detection method based on CNN mixed-data model. 2020, EasyChair.

[7] Wang Z, Li G, Zhuo Z, Ren X, Lin Y, Gu J. A deep learning method for android application classification using semantic features. Secur Commun Netw 2022;2022.

[8] Ravi V, Kp S, Poornachandran P, Kumar S S. Detecting android malware using long short-term memory (LSTM). J Intell Fuzzy Systems 2018;34:1277–88. http://dx.doi.org/10.3233/JIFS-169424.

[9] Scalas M, Maiorca D, Mercaldo F, Visaggio CA, Martinelli F, Giacinto G. R-PackDroid: Practical on-device detection of android ransomware, CoRR abs/1805.09563. 2018, arXiv:1805.09563.

[10] Han H, Lim S, Suh K, Park S, Cho S-J, Park M. Enhanced android malware detection: An SVM-based machine learning approach. In: 2020 IEEE international conference on big data and smart computing. 2020, p. 75–81. http://dx.doi.org/10.1109/BigComp48618.2020.00-96.

[11] Akbar F, Hussain M, Mumtaz R, Riaz Q, Wahab AWA, Jung K-H. Permissions-based detection of android malware using machine learning. Symmetry 2022;14(4). http://dx.doi.org/10.3390/sym14040718, URL https://www.mdpi.com/2073-8994/14/4/718.

[12] Chaulagain D, Poudel P, Pathak P, Roy S, Caragea D, Liu G, et al. Hybrid analysis of android apps for security vetting using deep learning. In: 2020 IEEE conference on communications and network security. 2020, p. 1–9. http://dx.doi.org/10.1109/CNS48642.2020.9162341.

[13] Vinayakumar R, Soman KP, Poornachandran P. Deep android malware detection and classification. In: 2017 international conference on advances in computing, communications and informatics. 2017, p. 1677–83. http://dx.doi.org/10.1109/ICACCI.2017.8126084.

[14] Lo WW, Layeghy S, Sarhan M, Gallagher M, Portmann M. Graph neural network-based android malware classification with jumping knowledge, CoRR abs/2201.07537. 2022, arXiv:2201.07537.

[15] Pengbin Feng TL, Ma X, Xi N, Lu D. Android malware detection via graph representation learning. Mob Inf Syst 2021;2021. http://dx.doi.org/10.1155/2021/5538841.

[16] Kim T, Kang B, Rho M, Sezer S, Im EG. A multimodal deep learning method for android malware detection using various features. IEEE Trans Inf Forensics Secur 2019;14(3):773–88. http://dx.doi.org/10.1109/TIFS.2018.2866319.

[17] Arp D, Spreitzenbarth M, Hübner M, Gascon H, Rieck K. DREBIN: Effective and explainable detection of android malware in your pocket. In: Symposium on network and distributed system security. 2014, http://dx.doi.org/10.14722/ndss.2014.23247.

[18] Scalas M, Rieck K, Giacinto G. Chapter 11 - Improving malware detection with explainable machine learning. In: Benois-Pineau J, Bourqui R, Petkovic D, Quénot G, editors. Explainable deep learning AI. Academic Press; 2023, p. 217–38. http://dx.doi.org/10.1016/B978-0-32-396098-4.00017-X, URL https://www.sciencedirect.com/science/article/pii/B978032396098400017X.

[19] Melis M, Maiorca D, Biggio B, Giacinto G, Roli F. Explaining black-box android malware detection, CoRR abs/1803.03544. 2018, arXiv:1803.03544.

[20] Melis M, Scalas M, Demontis A, Maiorca D, Biggio B, Giacinto G, et al. Do gradient-based explanations tell anything about adversarial robustness to android malware? CoRR abs/2005.01452 2020, arXiv:2005.01452.

[21] Kinkead M, Millar S, McLaughlin N, O'Kane P. Towards explainable CNNs for android malware detection. Procedia Comput Sci 2021;184:959–65. http://dx.doi.org/10.1016/j.procs.2021.03.118, The 12th international conference on ambient systems, networks and technologies (ANT) / The 4th international conference on emerging data and Industry 4.0 (EDI40) / Affiliated workshops. URL https://www.sciencedirect.com/science/article/pii/S1877050921007663.

[22] Iadarola G, Martinelli F, Mercaldo F, Santone A. Towards an interpretable deep learning model for mobile malware detection and family identification. Comput Secur 2021;105:102198. http://dx.doi.org/10.1016/j.cose.2021.102198, URL https://www.sciencedirect.com/science/article/pii/S0167404821000225.

[23] Wu B, Chen S, Gao C, Fan L, Liu Y, Wen W, et al. Why an android app is classified as malware? Towards malware classification interpretation, CoRR abs/2004.11516. 2020, arXiv:2004.11516.

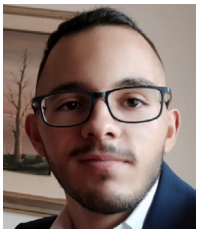
[24] Islam S, Elmekki H, Elsebai A, Bentahar J, Drawel N, Rjoub G, et al. A comprehensive survey on applications of transformers for deep learning tasks. 2023, http://dx.doi.org/10.48550/arXiv.2306.07303.

[25] Ullah F, Alsirhani A, Alshahrani MM, Alomari A, Naeem H, Shah SA. Explainable malware detection system using transformers-based transfer learning and multi-model visual representation. Sensors 2022;22(18). http://dx.doi.org/10.3390/s22186766, URL https://www.mdpi.com/1424-8220/22/18/6766.

[26] Jo J, Cho J, Moon J. A malware detection and extraction method for the related information using the ViT attention mechanism on android operating system. Appl Sci 2023;13(11). http://dx.doi.org/10.3390/app13116839, URL https://www.mdpi.com/2076-3417/13/11/6839.

[27] Kaliciński W. Smallerapk, part 1: Anatomy of an apk, medium - Android Developers. 2022, https://medium.com/androiddevelopers/smallerapk-part-1-anatomy-of-an-apk-da83c25e7003. (Online - Accessed 08 July 2022).

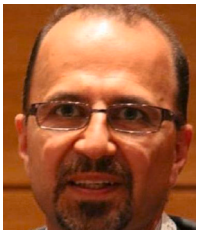
[28] Alvares S. App components: Entry points for Android applications. 2020, DevGenius. URL https://blog.devgenius.io/app-components-entry-points-for-android-applications-f3d0b0294af7. (Online - Accessed 18 October 2022).
[29] Gall R. Machine learning explainability vs interpretability: Two concepts that could help restore trust in AI. 2022, KDN nuggets. URL https://www.kdnuggets.com/2018/12/machine-learning-explainability-interpretability-ai.html. (Online - Accessed 10 July 2022).
[30] Xie N, Ras G, van Gerven M, Doran D. Explainable deep learning: A field guide for the uninitiated, CoRR abs/2004.14545. 2020, arXiv:2004.14545.
[31] Lundberg SM, Lee S. A unified approach to interpreting model predictions, CoRR abs/1705.07874. 2017, arXiv:1705.07874.
[32] Raul M, Pengbin F, Jiafeng M, Teng L, Xindi M, Ning X, et al. Android malware detection via graph representation learning. Mob Inf Syst 2021;2021/5538841.
[33] Yang Y, Du X, Yang Z, Liu X. Android malware detection based on structural features of the function call graph. Electronics 2021;10(2). http://dx.doi.org/10.3390/electronics10020186, URL https://www.mdpi.com/2079-9292/10/2/186.
[34] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. In: Proceedings of workshop at ICLR, vol. 2013. 2013.
[35] Kim Y. Convolutional neural networks for sentence classification, CoRR abs/1408.5882. 2014, arXiv:1408.5882.
[36] Yue W, Li L. Sentiment analysis using Word2vec-CNN-BiLSTM classification. In: 2020 seventh international conference on social networks analysis, management and security. 2020, p. 1–5. http://dx.doi.org/10.1109/SNAMS52053.2020.9336549.
[37] Morcos M, Al Hamadi H, Damiani E, Nandyala S, McGillion B. A surrogate-based technique for android malware detectors' explainability. In: 2022 18th international conference on wireless and mobile computing, networking and communications. 2022, p. 112–7. http://dx.doi.org/10.1109/WiMob55322.2022.9941515.
[38] Alani MM, Awad AI. PAIRED: An explainable lightweight android malware detection system. IEEE Access 2022;10:73214–28. http://dx.doi.org/10.1109/ACCESS.2022.3189645.
[39] Giannakas F, Kouliaridis V, Kambourakis G. A closer look at machine learning effectiveness in android malware detection. Information 2023;14(1). http://dx.doi.org/10.3390/info14010002, URL https://www.mdpi.com/2078-2489/14/1/2.
[40] Sebastián S, Caballero J. AVclass2: Massive malware tag extraction from AV labels. In: Annual computer security applications conference. New York, NY, USA: Association for Computing Machinery; 2020, p. 42–53. http://dx.doi.org/10.1145/3427228.3427261.
[41] Pendlebury F, Pierazzi F, Jordaney R, Kinder J, Cavallaro L. TESSERACT: Eliminating experimental bias in malware classification across space and time. In: 28th USENIX security symposium. Santa Clara, CA: USENIX Association; 2019, p. 729–46, URL https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury.
[42] Liu Y, Tantithamthavorn C, Li L, Liu Y. Explainable AI for android malware detection: Towards understanding why the models perform so well? In: 2022 IEEE 33rd international symposium on software reliability engineering. 2022, http://dx.doi.org/10.1109/ISSRE55969.2022.00026.



Diego Soi received his Master's degree in Computer Engineering, Cybersecurity and Artificial Intelligence from Università degli Studi di Cagliari, Cagliari, Italy in November 2022, and his Bachelor's degree in Electrical, Electronical and Computer Engineering from Università degli Studi di Cagliari, Cagliari, Italy in November 2020. Currently, he is a Ph.D. candidate for University degli Studi di Cagliari and his research interests are Malware analysis and detection for mobile systems, and the use of ML/DL on this kind of application.

Alessandro Sanna is a Corporate Ph.D. student for the University of Cagliari, where is a member of the Pattern Recognition and Application Laboratory, and Abissi Srl since April 2021. Currently researching on Malware Detection and Threat Intelligence, his studies focus primarily on Living-off-the-Land Malware and the use of ML/DL for Malware and Threat Detection.

Davide Maiorca received from the University of Cagliari (Italy) the M.Sc. degree (Hons.) in Electronic Engineering in 2012, and the Ph.D. in Computer and Electronic Engineering in 2016. He is currently a Senior Assistant Professor at the Department of Electrical and Electronic Engineering, University of Cagliari. His research fields include analyzing and detecting X86 and Android malware, malicious documents and multimedia applications (e.g., PDF, Microsoft Office), and Adversarial Machine Learning. Dr. Maiorca authored more than 25 research papers and has served as a Program Committee member and reviewer for international conferences and journals.



Giorgio Giacinto is a Professor of Computer Engineering at the University of Cagliari, Italy, where he serves as the coordinator of the M.Sc. degree in Computer Engineering, Cybersecurity and Artificial Intelligence.

His research interests are in machine learning for malware analysis and detection, and he published more than 180 papers in international conferences and journals. He is the Editor in Chief of the "Security Engineering & Applications" section of the Journal of Cybersecurity and Privacy He is a member of the managing committees of the Cybersecurity National Lab and the Artificial Intelligence & Intelligent Systems Lab within the CINI consortium, Italy. He also represents the Cybersecurity National Lab within the European Cybersecurity Organization (ECSO) He is a Fellow of the IAPR (International Association for Pattern Recognition) and a Senior Member of the IEEE Computer Society and ACM.