An Explainable Convolutional Neural Network for Dynamic Android Malware Detection

Francesco Mercaldo[1,2], Fabio Martinelli[2] and Antonella Santone[1]
[1]*University of Molise, Campobasso, Italy*
[2]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*

Keywords: Malware, Security, Deep Learning, Explainability, Android, Testing.

Abstract: Mobile devices, in particular the ones powered by the Android operating system, are constantly subjected to attacks from malicious writers, continuously involved in the development of aggressive malicious payload aimed to extract sensitive and private data from our smartphones and mobile devices. From the defensive point of view, the signature-based approach implemented in current antimalware has largely demonstrated its inefficacy in fighting novel malicious payloads but also old ones, when attackers apply (even simple) obfuscation techniques. In this paper, a method aimed to detect malware attacking mobile platforms is proposed. We exploit dynamic analysis and deep learning: in particular, we design the representation of an application as an image directly generated from the system call trace. This representation is then exploited as input for a deep learning network aimed to discern between malicious or trusted applications. Furthermore, we provide a kind of explainability behind the deep learning model prediction, by highlighting into the image obtained from the application under analysis the areas symptomatic of a certain prediction. An experimental analysis with more than 6000 (malicious and legitimate) Android real-world applications is proposed, by reaching a precision of 0.715 and a recall equal to 0.837, showing the effectiveness of the proposed method. Moreover, examples of visual explainability are discussed with the aim to show how the proposed method can be useful for security analysts to better understand the application malicious behaviour.

1 INTRODUCTION AND RELATED WORK

Mobile devices are present in a plethora of aspects of our life for instance, to keep in touch with our relatives but also to make payments through an app provided from our bank or to apply a digital signature on a document.

Considering the huge amount of sensitive and private information that every day are stored on our mobile devices, it is clear that these devices are really of interest for attackers, devoted to develop more and more aggressive malicious payloads to steal and obtain illicit gains from our sensitive information.

From the security point of view, the current antimalware detection mechanism, mainly signature-based, is not able to recognize new threats: as a matter of fact a malicious payload can be detected only whether its signature is stored into the antimalware database signature and, for this reason, a malicious payload must be already analysed from security analysts in order to be detected (Mercaldo and Santone, 2021).

Moreover, even when a malware signature is stored into the antimalware repository, malicious writes typically employ obfuscation techniques aimed to evade the detection preserving the malicious payload business logic (Canfora et al., 2015).

From the mobile operating system diffusion, in 2022 Android is the most popular operating system in the world, with over 2.5 billion active users spanning over 190 countries[1].

Google Play, the official platform for downloading mobile applications for Android-powered devices, has grown enormously in the past decade, reaching $38.6 billion revenue in 2020. There were over 2.9 million apps available on the store in 2020, which were downloaded 108 billion times.

Android is the dominant platform in most countries, although it has had trouble surpassing Apple in Japan and the United States. In countries like Brazil,

[1]https://www.businessofapps.com/data/android-statistics/

Mercaldo, F., Martinelli, F. and Santone, A.
An Explainable Convolutional Neural Network for Dynamic Android Malware Detection.
DOI: 10.5220/0011609800003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 305-312
ISBN: 978-989-758-624-8; ISSN: 2184-4356

India, Indonesia, Iran and Turkey, it has over 85 percent market share.

From these considerations, it urges to design new methods for effective mobile malware detection. For this reason, in this paper we propose a novel approach based on dynamic techniques for the detection of malicious behavior.

We design a convolution neural network (CNN) to discriminate between malware and trusted applications. The CNN input is represented by images directly generated from the system call traces of a set of (malicious and trusted) Android applications. Moreover, with the aim to explain the model decision, we adopt the Gradient-weighted Class Activation Mapping (Grad-CAM) algorithm in order to automatically generate a localization map by highlighting the image regions responsible for a certain prediction.

As mobile platform for the evaluation analysis we consider the Android one anyway (i.e., the most diffused one), the proposed method in operating system independent.

Current state-of-the-art research papers proposed several methods in the mobile malware detection context, for instance, authors (Jerbi et al., 2020) consider the API call sequences' population to find new malware behaviors, applying some well-defined evolution rules. The malware models obtained in this way are inserted into the set of unreliable behaviors, to create diversification between malware samples in order to increase the detection rate.

In (Casolare et al., 2020) authors consider a static approach, based on formal methods by exploiting model checking to perform the detection of applications involved in colluding attacks.

Differently from these papers, we propose an approach to dynamically detect Android malware through the analysis of system calls by means of explainable deep learning. In detail, we propose to represent applications in terms of images obtained from the system call traces by taking into account a way to (visually) explain the rationale behind the model precision (i.e., the malware detection).

The paper proceeds as follows: in Section 2 the proposed method for dynamic mobile malware detection is discussed, in Section 3 with the aim to demonstrate the effectiveness of the proposed method we present the experimental analysis with real-world (malicious and trusted) Android applications and, finally, conclusion and future research lines are presented in the last section.

2 MOBILE MALWARE DETECTION THROUGH EXPLAINABLE CNN

In this section we present the method we designed for Android malware detection through dynamic analysis. In particular we extract, from a running application, the system call traces and we obtain an image directly from these traces to input a convolutional neural network designed by authors.

In the next subsection we describe the proposed method for malware detection, composed by three main phases: the first is aimed to obtain system call traces and to generate the system call images, the second one to build a model with a convolution neural network designed by authors and the last phase is devoted to evaluate the model built in the previous phase.

2.1 System Call Image Generation

Figure 1 shows how we obtain an image from a system call trace obtained from a running application.

The idea is to capture and store, in a textual format, the system call traces generated by running applications. For this aim, the *.apk* installation file of an Android application (*App under analysis* in Figure 1) is installed and initialised on an Android device emulator (*Android device* in Figure 1). Successively, a set of 25 different operating system events (Zhou and Jiang, 2012; Jiang and Zhou, 2013) is generated at regular time intervals of 10 seconds (*Operating System Event Injection* in Figure 1) and sent to the emulator with the aim to stimulate the malicious payload behaviour (*Application Running* in Figure 1) and, from this, the correspondent sequence of system calls is obtained (*System call Extraction* in Figure 1).

We exploit a set of different 25 operating system events because several previous papers (Zhou and Jiang, 2012; Jiang and Zhou, 2013; Casolare et al., 2021) demonstrated that these events are considered by malicious writers to activate the payloads in Android environment. In particular, we considered the operating system event aimed to trigger Android malware exploited by authors in (Casolare et al., 2021; Mercaldo et al., 2016; Medvet and Mercaldo, 2016).

The system call retrieval from the Android application is performed by a shell script (*Python Script* in Figure 1) developed by authors aimed to perform in sequence a set of actions below described:

- initialisation of the target Android device emulator;

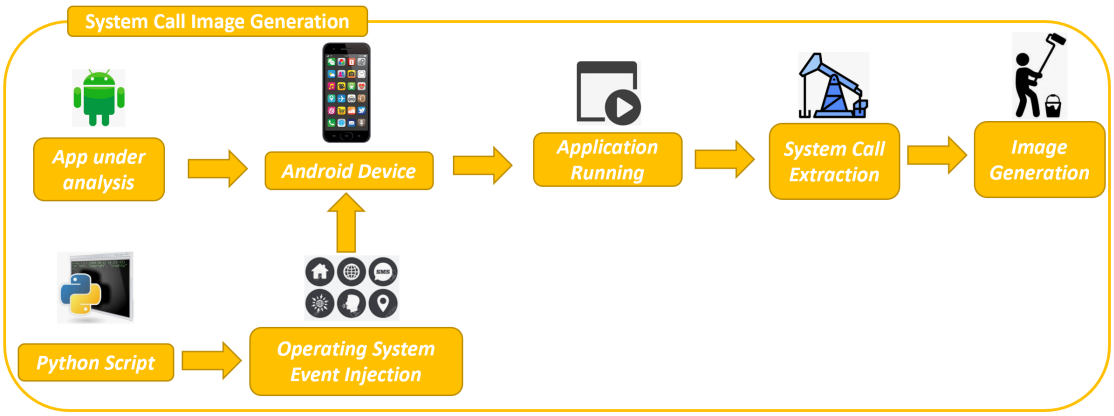

Figure 1: The system call extraction and the image generation step.

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

Moreover we exploit the monkey tool belonging to the Android Debug Bridge (ADB[2]) version 1.0.32, to generate pseudo-random user events such as, for instance, clicks, touches, or gestures (with the aim to simulate the user interaction with the Android application under analysis).

To collect the system call traces we consider strace[3], a tool freely available on Linux operating systems. In detail, we invoke the command *strace -s PID* to hook the running Android application process to intercept only syscalls generated by the application under analysis process.

Once obtained a log of system calls, we extract one by one the single calls and respecting the order given by the log we build the image (*Image Generation* in Figure 1). We assign a certain RGB pixel to a certain system, in this way we create the images, pixel by pixel.

With the aim to better understand how the images are obtained from system call traces, in Figures 2 and 3 we show two examples of images obtained from the system call traces of two malware belonging to the Opfake family. In detail in Figure 2 is represented the first malware sample[4] detected as malware by 35 antimalware on 63 provided by Virustotal[5].

In Figure 3 is represented another Opfake malicious sample[6] detected as malware by 33 on 63 Virustotal antimalware[7].

Figure 4 shows an example of images obtained from the system call trace of a legitimate application. In particular, we show the image related to the Voice Changer [8] application, an app aimed to record a sound

[2]https://developer.android.com/studio/command-line/adb

[3]https://man7.org/linux/man-pages/man1/strace.1.html

[4]identified by the 1b41f7f3d25b916c42d91e9561afeb6f6a927d3549381aa57ab684728f0e245dhash

[5]https://www.virustotal.com/gui/file/1b41f7f3d25b916c42d91e9561afeb6f6a927d3549381aa57ab684728f0e245d

[6]identified by the 1b81427b68c71cc3242ed0b86c2dd93154b1b8c31fe9261a9eb886b04ae21058hash

[7]https://www.virustotal.com/gui/file/1b81427b68c71cc3242ed0b86c2dd93154b1b8c31fe9261a9eb886b04ae21058

[8]https://play.google.com/store/apps/details?id=com.meihillman.voicechanger&hl=en_US&gl=ES

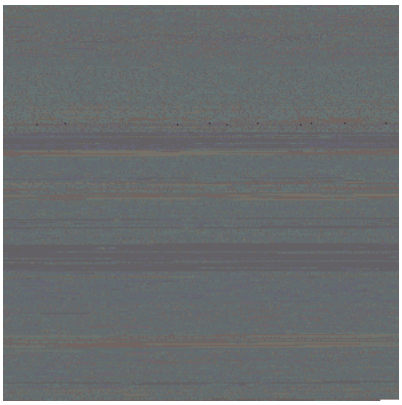

Figure 2: A first example of image generated from the system call trace of a malware belonging to the Opfake family.

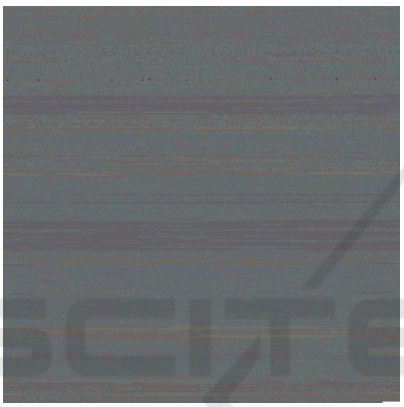

Figure 3: A second example of image generated from the system call trace of a malware belonging to the Opfake family.

or select an existing audio file and to convert the audio using a set of effects.

As we can see, in the image we can see that to each color is associated with the relative system call. In particular, both the malware samples showed in Figures 2 and 3, are belonging to the same family i.e., Opfake. In the image in Figure 2 and in the image in Figure 3, we can see the representations about two different malware applications, that in this case they look similar to each other but different from the trusted application representation in Figure 4. In fact, the malware images have some common parts like the two brown bands, that on the contrary are absent in the trusted image. In this way, we already have a visual impact that allows us to notice the differences between trusted and malware applications. Hence this method of representation through image generation could be used to discern between malware applications and trusted ones.

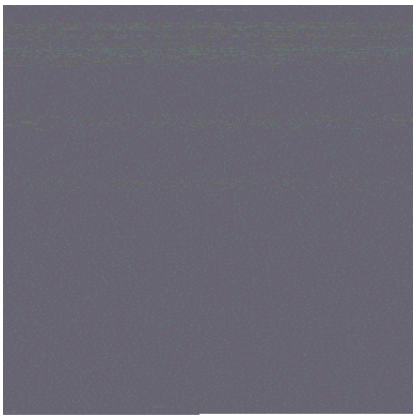

Figure 4: An example of image generated from the system call trace obtained from a legitimate Android application.

2.2 The CNN Training

Once obtained a set of images from the system call traces we can use these images to input the CNN designed by authors in the *Training* step, as shown from Figure 5.

We obtain the images related to a set of trusted and malware Android applications. In addition to the images dataset we need the labels (*Labelled Image Dataset* in Figure 5) related to each application i.e., malware or trusted.

Once obtained all the images for each (malware and trusted) application, with the related label they represent the input for the CNN we designed (*Deep Learning Network* in Figure 5). We designed a deep learning model composed of 14 different layers carried out by the utilization of the following six layers: *Conv2D*, *MaxPooling2D*, *BathNormalization*, *Flatten*, *Dropout* and *Dense*.

The Python code related to the developed model is shown in Listing 1.

By exploiting the designed deep learning network we build the model (i.e., *Model* in Figure 5) that will be considered to discriminate between malware and trusted (unknown) Android applications.

2.3 The CNN Testing

Once built the model, we evaluate its effectiveness in Android malware detection in the testing step, depicted in Figure 6.

In this step we consider a set of Android applications not exploited in the previous step (i.e., *Unknown Application* in Figure 6): through the System Call Image Generation step we extract the system call traces, we obtain the related images used as input for the model built in the previous step. The model will

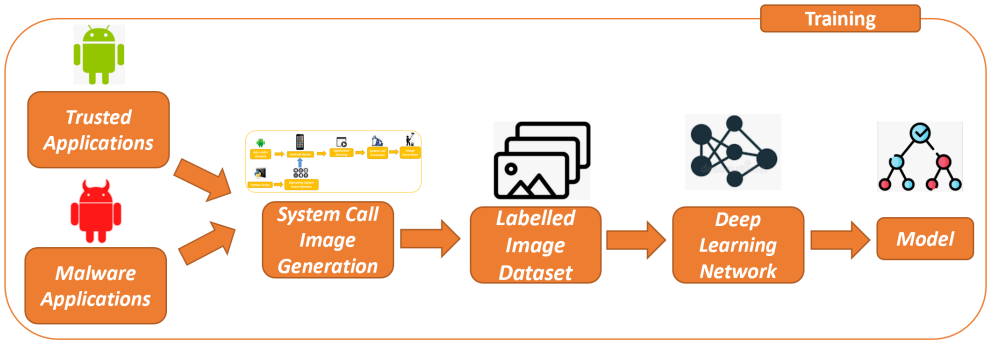

Figure 5: The convolutional neural network training step.

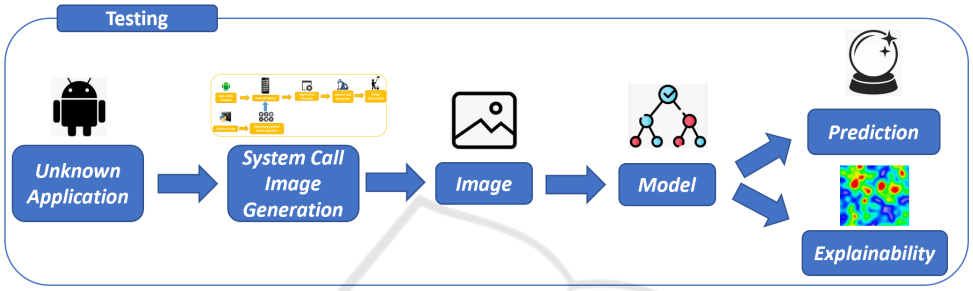

Figure 6: The model testing step.

```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3),
activation='relu',
input_shape=(50, 50 ,1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization(renorm=True))
model.add(layers.Conv2D(128, (3, 3),
activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization(renorm=True))
model.add(layers.Conv2D(256, (3, 3),
activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization(renorm=True))
model.add(layers.Flatten())
model.add(layers.Dense(256,
activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(self.num_classes,
activation='sigmoid'))
model.compile(loss='binary_crossentropy',
optimizer=Adam(self.learning_rate),
metrics=['acc', Precision(name="prec"),
Recall(name="rec")])
```

Listing 1: Python code snippet of the deep learning network.

output a label (i.e., malware or trusted) (i.e., *Prediction* in Figure 6) and will provide a visual explanation related to the prediction (i.e., *Explainability* in Figure 6) by highlighting the areas of the image related of a certain prediction: this makes the proposed CNN explainable. To highlight the areas of the images symptomatic of certain prediction, we generate the activation maps for a subset of images by exploiting the Grad-CAM algorithm (Selvaraju et al., 2017), used with the aim to provide a visual explanation behind the model prediction (i.e., malware or trusted). The rationale behind the adoption of this approach is not only to acquire a high prediction accuracy but also to explore the portions of the image under analysis that are responsible for a specific prediction. The aim is to explain the forecast by localizing malicious-symptomatic regions. This allows to determine whether the model is looking in the proper spot to generate a prediction.

3 EXPERIMENTAL ANALYSIS

In this section we describe, respectively, the real-world dataset involved in the experimental analysis and the classification results.

3.1 The Dataset

The dataset considered in the experimental analysis was gathered from two different repositories: relating to the malicious samples we obtained real-world Android malware from the Drebin dataset (Arp et al.,

2014; Michael et al., 2013), a very well-known collection of malware largely considered by malware analysis researchers, including the most widespread Android families. The malware dataset is freely available for research purposes [9]. The malware dataset is also partitioned according to the *malware family*; each family contains malicious samples sharing several characteristics: the payload installation, the kind of attack and the events triggering the malicious payload (Zhou and Jiang, 2012). The following malware families are considered in the experimental evaluation, with the aim to try to cover the full spectrum of mobile malicious behaviours: *Geinimi*, *Plankton*, *Basebridge*, *Kmin*, *GinMaster*, *Opfake*, *FakeInstaller*, *DroidDream*, *DroidKungFu* and *Adrd* (we considered the most populous 10 malware families in the Drebin dataset).

To gather legitimate applications, we crawled the official app store of Google[10], by using an open-source crawler[11]. The obtained collection includes samples belonging to all the different categories available on the market.

We analyzed the dataset with the VirusTotal service[12]: this analysis confirmed that the trusted applications did not contain malicious payload while the malicious ones were actually recognized as malware. The (malicious and legitimate) dataset is composed by 6817 samples: 3355 malicious (belonging to 10 different malicious families) and 3462 trusted. From the malicious and legitimate dataset we gathered the system call sequences with the procedure explained in the previous section. Subsequently, from each system call trace we generated the relative image representation used for the training and the testing of the designed CNN.

3.2 Classification Results

In this section we present the results of the experimental analysis aimed to verify whether the designed deep neural network is able to discriminate between malicious and trusted Android applications.

In order to evaluate the results of the classification following metrics are computed: Precision, Recall, Accuracy and Loss.

For replication purposes, we report the hyper-parameters we considered for model training in Table 1.

To perform the experiment we considered a machine with a i7 8th Generation Intel CPU and 16GB RAM memory, equipped with Microsoft Windows 10 and running the Windows Subsystem for Linux. The CNN was implemented with the Python programming language (version 3.6.9) by exploiting the Tensorflow 2.4.4 library. We train the CNN by considering 70% of the dataset, 15% is considered for validation, while the remaining 15% is exploited for the CNN testing.

Table 1: The hyper-parameters considered for the CNN training step.

Input	Batch	Epochs	Learning Rate
50x50X3	128	10	0.001

Table 2: Experimental analysis results.

step	Loss	Accuracy	Precision	Recall
training	0.780	0.565	0.567	0.579
testing	0.612	0.781	0.715	0.837

The results of the experimental analysis are provided in Table 2.

Figure 7 shows the accuracy trend for the 10 epochs we considered, in green the accuracy train trend, and in grey the accuracy testing one. As shown from Figure 7 after seven epochs (in testing) the performances in detection are increasing.

Figure 8 shows the loss trend for the 10 epochs we considered. As shown from the loss trend in Figure 8 the loss exhibits an opposite trend with respect

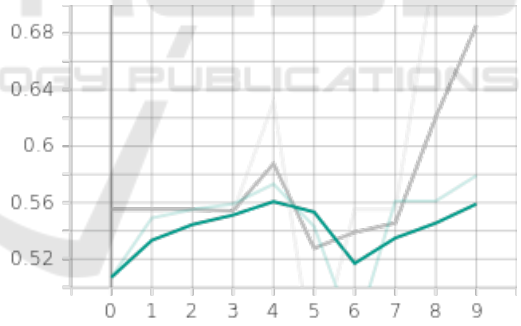

Figure 7: The accuracy trends for the 10 epochs: in green the accuracy train trend, in grey the accuracy testing one.

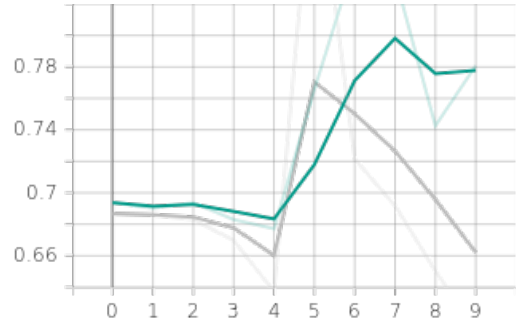

Figure 8: The loss trends for the 10 epochs: the green line is related to the loss train trend, while the grey line is related to the loss testing trend.

[9] https://www.sec.cs.tu-bs.de/~danarp/drebin/
[10] https://play.google.com/store
[11] https://github.com/liato/android-market-api-py
[12] https://www.virustotal.com/

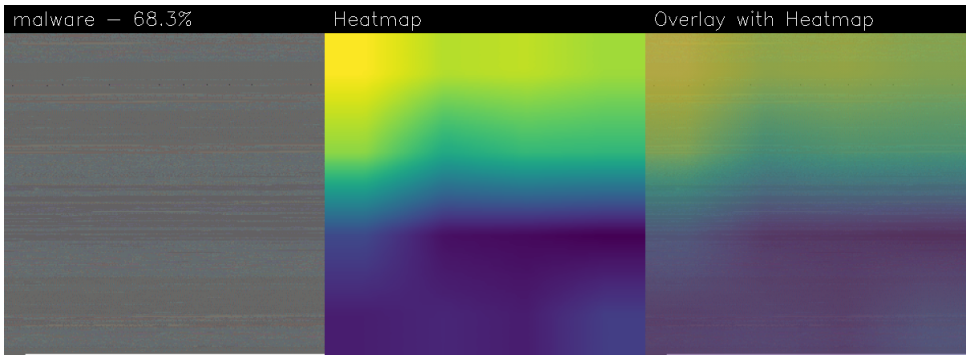

Figure 9: An example of explainability provided by the proposed method for the malware identified by the 1f0133014fac23a55fec3f56c9a09a2f0b7d02fc8f0c6c5b703397562e57098f hash.

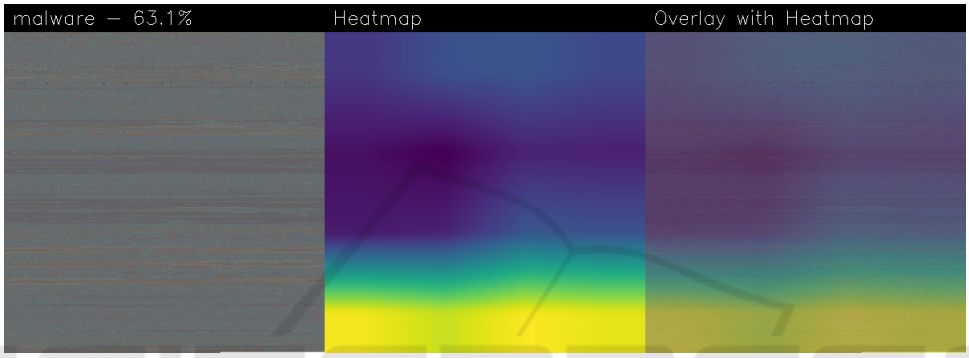

Figure 10: An example of explainability provided by the proposed method for the malware identified by the 2bba69f5994c871380d5633e37305e7c09db17bb3ebaeeb94163625609d866d9 hash.

to the accuracy, and this is what is expected when a deep learning model is correctly learning the classification task: this happens for both the training and the testing.mIn Figures 9 and 10 two examples of explainability proposed by the proposed method are shown. In each figure the first part of the image represents the image obtained from the system call trace, the middle image is the heatmap generated by the Grad-CAM with highlighted the areas symptomatic of a certain prediction and, in the right image, the overlay of the image generated from the system call trace with the heatmap provided by the Grad-CAM, in order to understand which part of the system call image is highlighted by the heatmap and it is, consequently, responsible for the (malware or trusted) prediction. In detail the yellow areas of the heatmap are related to part of the images that contributed to the detection with a high percentage, the green areas contributed with a medium percentage while the blue area did not contribute to the detection thus, from the explainability point of view they are not considered of interest.

In particular in Figure 9 we can note that the proposed model correctly detected the malware application, with a percentage equal to 68.3%. From the explainability point of view, we can note that this application is marked as malware because the upper area was considered malicious from the proposed model. This data can be of interest, as it can lead the security analysis to conduct further investigations. For example, by going back to the system calls highlighted in yellow, a fine-grained localization of the system calls invoked by the malicious payload could be carried out, as it would be possible to understand which event in particular allowed the generation of those particular system calls (considering in fact that the events of the operating system are generated in a predetermined order). Similar considerations can be done for the second example of visual explainability, shown in Figure 10. As a matter of fact, in the example shown in Figure 10 the model rightly classified the malicious application with a percentage equal to 63.1% but, differently from the example shown in Figure 10, the heatmap localised the yellow areas of interest in the lower part of the image. This can be symptomatic that this malicious payload was triggered by one of the final events sent to the operating system. Also in this case it might be of interest to further investigate the highlighted system calls, to see if they are similar to those highlighted in other applications. In this way, it may be possible to understand the exact sequence of

system calls generated by the execution of malicious behavior.

4 CONCLUSION AND FUTURE WORK

Every day we store a lot of sensitive and private information on our mobile devices. This is the reason why the interest of attackers with regard to our smartphone and tables is day-by-day increasing, with the development of more and more aggressive malicious payload devoted to exfiltrate our sensitive data. From these considerations, a method aimed to detect mobile malware is proposed in this paper. We focus on the most widespread mobile platform i.e., Android, by designing a method aimed to perform a dynamic analysis by extracting the system call trace of an application under analysis.

We exploit a CNN designed by authors to analyse images directly obtained from the system call trace to discern malicious applications from legitimate ones by obtaining an accuracy equal to 0.781. Moreover, we resort to the Grad-CAM to highlight into the image representing the application system call trace the areas symptomatic of a certain prediction, thus providing explainability behind the model prediction.

As future work, we plan to consider more algorithms to provide explainability for instance, the Grad-CAM++ (Chattopadhay et al., 2018) and the Score-CAM (Wang et al., 2020), to compare visual explanations. Also other deep learning models will be considered, for instance, the VGG19 and the ResNet ones with the aim to increase malware detection accuracy. Moreover, considering that the proposed method is platform-independent, we will also consider a dataset of PC ransomware and legitimate applications .

ACKNOWLEDGEMENTS

This work has been partially supported by EU DUCA, EU CyberSecPro, and EU E-CORRIDOR projects and PNRR SERICS_SPOKE1_DISE, RdS 2022-2024 cybersecurity.

REFERENCES

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20.

Casolare, R., De Dominicis, C., Iadarola, G., Martinelli, F., Mercaldo, F., and Santone, A. (2021). Dynamic mobile malware detection through system call-based image representation. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 12(1):44–63.

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

Chattopadhay, A., Sarkar, A., Howlader, P., and Balasubramanian, V. N. (2018). Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pages 839–847. IEEE.

Jerbi, M., Dagdia, Z. C., Bechikh, S., and Said, L. B. (2020). On the use of artificial malicious patterns for android malware detection. *Computers & Security*, 92:101743.

Jiang, X. and Zhou, Y. (2013). *Android Malware*. Springer Publishing Company, Incorporated.

Medvet, E. and Mercaldo, F. (2016). Exploring the usage of topic modeling for android malware static analysis. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 609–617. IEEE.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Download malware? no, thanks: how formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2016, Austin, Texas, USA, May 15, 2016*, pages 22–28. ACM.

Mercaldo, F. and Santone, A. (2021). Formal equivalence checking for mobile malware detection and family classification. *IEEE Transactions on Software Engineering*.

Michael, S., Florian, E., Thomas, S., Felix, C. F., and Hoffmann, J. (2013). Mobilesandbox: Looking deeper into android applications. In *Proceedings of the 28th International ACM Symposium on Applied Computing (SAC)*.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626.

Wang, H., Wang, Z., Du, M., Yang, F., Zhang, Z., Ding, S., Mardziel, P., and Hu, X. (2020). Score-cam: Score-weighted visual explanations for convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 24–25.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*.