DexRay: A Simple, yet Effective Deep Learning Approach to Android Malware Detection based on Image Representation of Bytecode

Nadia Daoudi, Jordan Samhi, Abdoul Kader Kabore, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein

SnT, University of Luxembourg
29, Avenue J.F Kennedy, L-1359, Luxembourg, Luxembourg
{nadia.daoudi, jordan.samhi, abdoulkader.kabore, kevin.allix, tegawende.bissyande, jacques.klein}@uni.lu

Abstract. Computer vision has witnessed several advances in recent years, with unprecedented performance provided by deep representation learning research. Image formats thus appear attractive to other fields such as malware detection, where deep learning on images alleviates the need for comprehensively hand-crafted features generalising to different malware variants. We postulate that this research direction could become the next frontier in Android malware detection, and therefore requires a clear roadmap to ensure that new approaches indeed bring novel contributions. We contribute with a first building block by developing and assessing a baseline pipeline for image-based malware detection with straightforward steps.
We propose DexRay, which converts the bytecode of the app DEX files into grey-scale "vector" images and feeds them to a 1-dimensional Convolutional Neural Network model. We view DexRay as foundational due to the exceedingly basic nature of the design choices, allowing to infer what could be a minimal performance that can be obtained with image-based learning in malware detection.
The performance of DexRay evaluated on over 158k apps demonstrates that, while simple, our approach is effective with a high detection rate (F1-score = 0.96). Finally, we investigate the impact of time decay and image-resizing on the performance of DexRay and assess its resilience to obfuscation.
This **work-in-progress paper** contributes to the domain of Deep Learning based Malware detection by providing a sound, simple, yet effective approach (with available artefacts) that can be the basis to scope the many profound questions that will need to be investigated to fully develop this domain.

Keywords: Android Security · Malware Detection · Deep Learning

1 Introduction

Automating malware detection is a key concern in the research and practice communities. There is indeed a huge number of samples to assess, making it challenging to consider any manual solutions. Consequently, several approaches have been proposed in the literature to automatically detect malware [1–4]. However, current approaches remain limited and detecting all malware is still considered an unattainable dream. A recent report from McAfee [5] shows that mobile malware has increased by 15% between the first and the second quarter of 2020. Moreover, Antivirus companies and Google, the official Android market maintainer, have disclosed that malware apps become more and more sophisticated and threaten users' privacy and security [6–8].

To prevent the spread of malware and help security analysts, researchers have leveraged Machine Learning approaches [9–15] that rely on features extracted statically, dynamically, or in an hybrid manner. While a dynamic approach extracts the features when the apps are running, a static approach relies exclusively on the artefacts present in the APK file. As for hybrid approaches, they combine both statically and dynamically retrieved features. Both static and dynamic approaches require manual engineering of the features in order to select some information that can, to some extent, approximate the behaviour of the apps. Also, the hand-crafted features might miss some information that is relevant to distinguish malware from benign apps. Unfortunately, good feature engineering remains an open problem in the literature due to the challenging task of characterising maliciousness in terms of app artefacts.

Recently, a new wave of research around representation of programs/software for malware detection has been triggered. Microsoft and Intel Labs have proposed STAMINA[1], a Deep Learning approach that detects malware based on image representation of binaries. This approach is built on deep transfer learning from computer vision, and it was shown to be highly effective in detecting malware. In the Android research community, image-based malware detection seems to be attractive. Some approaches have investigated the image representation of the APK's code along with deep learning techniques. However, they have directly jumped to non-trivial representations (e.g., colour) and/or leveraged complex learning architectures (e.g., Inception-v3 [16]). Furthermore, some supplementary processing steps are applied which may have some effects on the performance yielded. Besides, they create square or rectangular images, which might distort the bytecode sequences from the DEX files, therefore at the same time losing existing *locality* and creating artificial *locality*. Indeed, given that the succession of pixels in the image depends on the series of bytes in the DEX files, converting the bytecode to a "rectangular" image can result in having patterns that start at the end of a line (i.e., row of the image) and finish at the beginning of the next line in the image. Moreover, related approaches leverage 2-dimensional convolutional neural networks, which perform convolution operations that take into consideration the pixels in the 2-d neighbourhood using 2-d kernels. Since there is no relationship between pixels belonging to the same column (i.e., pixels that are above/below each others) of a "rectangular" image

[1] https://www.microsoft.com/security/blog/2020/05/08/microsoft-researchers-work-with-intel-labs-to-explore-new-deep-learning-approaches-for-malware-classification/

representation of code, the use of 2-d convolutional neural networks seems to be inappropriate.

Image-based representation is, still, a sweet spot for leveraging advances in deep learning as it has been demonstrated with the impressive results reported in the computer vision field. To ensure the development of the research direction within the community, we propose to investigate a straightforward "vector" image-based malware detection approach leveraging both a simple image representation and a basic neural network architecture. Our aim is to deliver a foundational framework towards enabling the community to build novel state-of-the-art approaches in malware detection. This paper presents the initial insights into the performance of a baseline that is made publicly available to the community.

Our paper makes the following contributions:

- We propose DexRay, a foundational image-based Android malware detection approach that converts the sequence of raw byte code contained in the DEX file(s) into a simple grey-scale "vector" image of size (1, 128*128). Features extraction and classification are assigned to a 1-dimensional-based convolutional neural network model.
- We evaluate DexRay on a large dataset of more than 158k malware and goodware apps.
- We discuss the performance achievement of DexRay in comparison with prior work, notably the state-of-the-art Drebin approach as well as related work leveraging APK to image representations.
- We evaluate the performance of DexRay on detecting new malware apps.
- We investigate the impact of image-resizing on the performance of DexRay.
- We assess the resilience of DexRay and Drebin to apps' obfuscation.
- We make the source code of DexRay publicly available along with the dataset of apps and images

2 Approach

In this section, we present the main basic blocks of DexRay that are illustrated in Fig. 1. We present in Section 2.1 our image representation process which covers step (1.1) and (1.2) in Fig. 1. As for step (2), it is detailed in Section 2.2

2.1 Image representation of Android apps

Android apps are delivered in the form of packages called APK (Android Package) whose size can easily reach several MB [17]. The APK includes the bytecode (DEX files), some resource files, native libraries, and other assets.

(1.1) Bytecode extraction: Our approach focuses on code, notably the applications' bytecode, i.e., DEX files, where the app behaviour is supposed to be implemented.

(1.2) Conversion into image: Our straightforward process for converting DEX files into images is presented in Fig. 2. We concatenate all the DEX files

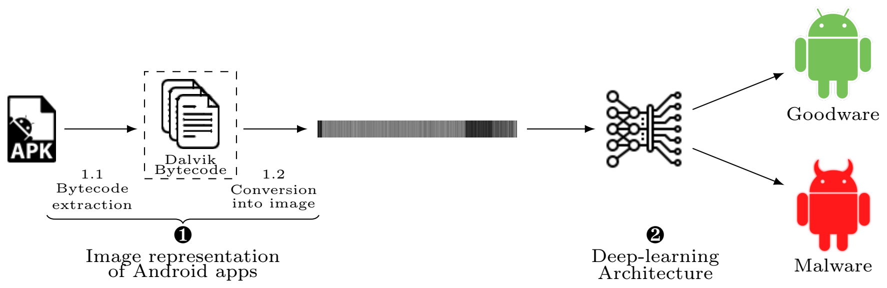
APK
Dalvik Bytecode
1.1 Bytecode extraction
1.2 Conversion into image
❶
Image representation of Android apps
❷
Deep-learning Architecture
Goodware
Malware

Fig. 1. DexRay basic building blocks.

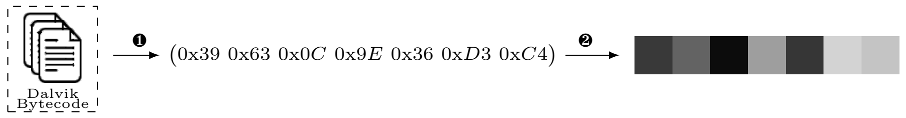

Fig. 2. Process of image generation from dalvik bytecode. ❶: bytecode bytes' vectorisation; ❷: Mapping bytes to pixels.

present in the APK as a single byte stream vector (step ❶ in Fig. 2). This vector is then converted to a grey-scale "vector" image by considering each byte as an 8-bit pixel (step ❷ in Fig. 2). Given that apps can widely differ in their code size, the size of the resulting "vector" images is also different. Note that the size of our "vector" image representation refers to the width of the image, since its height is 1. To comply with the constraints [18] of off-the-shelf deep learning architectures for images, we leverage a standard image resizing algorithm[2] to resize our "vector" images to a fixed-size. For our experiments, we have selected a size of (1, 128x128). We also investigate the impact of resizing on the performance of DexRay in Section 4.3.

We remind that related image-based Android malware detectors leverage a "rectangular" image representation, which might destroy the succession of bytes in the DEX files (i.e., the succession of pixels in the rectangular image). Moreover, this representation usually requires padding to have a rectangular form of the image. The complete procedure that shows the difference between our "vector" image and related approaches "rectangular" image generation is detailed in Algorithm 1 and Algorithm 2 respectively. While there is more advanced "rectangular" image representations, Algorithm 2 shows an illustration with a basic "square" grey-scale image of size (128, 128).

Comparing the two Algorithms, we can notice that our image representation is very basic since it uses the raw bytecode "as it is", without any segmentation or padding needed to achieve a "square" or "rectangular" form of the image. In the remainder of this paper, we use "image" to refer to our "vector" image representation of code.

[2] https://www.tensorflow.org/api_docs/python/tf/image/resize

Algorithm 1: Algorithm describing 8-bit grey-scale "vector" image generation from an APK

```
Input: APK file
Output: 8-bit grey-scale "vector" image of size (1, 128x128)
bytestream ← ∅
for dexFile in APK do
    bytestream ← bytestream + dexFile.toByteStream()
end
l ← bytestream.length()
img ← generate8bitGreyScaleVectorImage(bytestream, l)
img.resize_to_size(height=1, width=128x128)
img.save()
```

Algorithm 2: Algorithm describing 8-bit grey-scale "square" image generation from an APK

```
Input: APK file
Output: 8-bit grey-scale "square" image of size (128, 128)
bytestream ← ∅
for dexFile in APK do
    bytestream ← bytestream + dexFile.toByteStream()
end
l ← bytestream.length()
sqrt ← $\lceil\sqrt{l}\rceil$
sq ← $\text{sqrt}^2$
while bytestream.length() $\neq$ sq do
    bytestream ← bytestream + "\x00" // padding with zeros
end
// At this point, bytestream is divided in sqrt part of length sqrt
// In other words, it is represented as a sqrt × sqrt matrix
img ← generate8bitGreyScaleSquareImage(bytestream, sqrt)
img.resize_to_size(height=128, width=128)
img.save()
```

2.2 Deep Learning Architecture

Convolutional Neural Networks (CNN) are specific architectures for deep learning. They have been particularly successful for learning tasks involving images as inputs. CNNs learn representations by including three types of layers: Convolution, Pooling, and Fully connected layers [19]. We describe these notions in the following:

– The convolution layer is the primary building block of convolutional neural networks. This layer extracts and learns relevant features from the input images. Specifically, a convolutional layer defines a number of filters (matrices) that detect patterns in the image. Each filter is convolved across the input image by calculating a dot product between the filter and the input. The filters' parameters are updated and learned during the network's training with the backpropagation algorithm [20, 21]. The convolution operation creates feature maps that pass first through an activation function, and then they are received as inputs by the next layer to perform further calculation.

– A pooling layer is generally used after a convolutional layer. The aim of this layer is to downsize the feature maps, so the number of parameters and the time of computation is reduced [22]. Max pooling and Average pooling are two commonly used methods to reduce the size of the feature maps received by a pooling layer in CNNs [23].

– In the Fully connected layer, each neuron is connected to all the neurons in the previous and the next layer. The feature maps from the previous convolution/pooling layer are flattened and passed to this layer in order to make the classification decision.

Among the variety of Deep-Learning architectures presented in the literature, CNNs constitute a strong basis for deep learning with images. We further propose to keep a minimal configuration of the presented architecture by implementing a convolutional neural network model that makes use of 1-dimensional convolutional layers. In this type of layers, the filter is convolved across one dimension only, which reduces the number of parameters and the time of the training. Also, 1-d convolutional layers are the best suited for image representation of code since the pixels represent the succession of bytecode bytes' from the apps. The use of 1-d convolution on our images can be thought of as sliding a convolution window over the sequences of bytes searching for patterns that can distinguish malware and benign apps.

We present in Fig. 3 our proposed architecture, which contains two 1-dimensional convolutional/pooling layers that represent the extraction units of our neural network. The feature maps generated by the second max-pooling layer are then flattened and passed to a dense layer that learns to discriminate malware from benign images. The second dense layer outputs the detection decision.

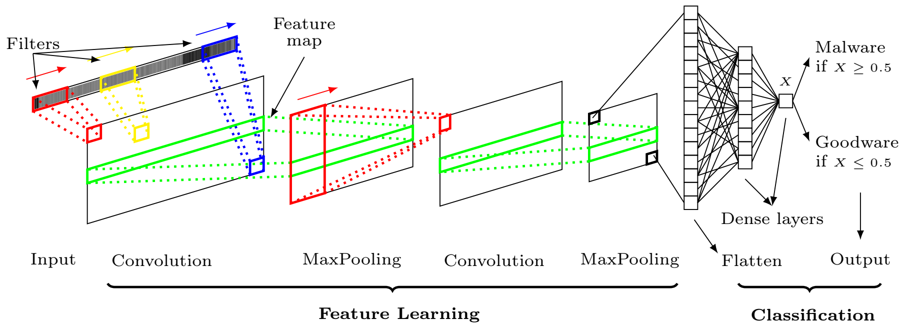

Fig. 3. Our Convolutional Neural Network architecture

3 Study Design

In this section, we first overview the research questions that we investigate. Then, we present the datasets and the experimental setup used to answer these research questions.

3.1 Research Questions

In this paper, we consider the following four main research questions:

- **RQ1:** How effective is DexRay in the detection of Android malware?
- **RQ2:** How effective is DexRay in detecting new Android malware?
- **RQ3:** What is the impact of image-resizing on the performance of DexRay?
- **RQ4:** How does app obfuscation affect the performance of DexRay?

3.2 Dataset

Initial Dataset. To conduct our experiments, we collect malware and benign apps from AndroZoo [17] which contains, at the time of writing, more than 16 million Android apps. AndroZoo crawls apps from several sources, including the official Android market Google Play[3]. We have collected AndroZoo apps that have their compilation dates between January 2019 and May 2020. Specifically, our dataset contains 134 134 benign, and 71 194 malware apps. Benign apps are defined as the apps that have not been detected by any antivirus from VirusTotal[4]. The malware collection contains the apps that have been detected by at least two antivirus engines, similarly to what is done in DREBIN [9].
Obfuscation process. In our experiments, we further explore the impact of app obfuscation. Therefore, we propose to generate obfuscated apps, by relying on the state-of-the-art Android app obfuscator Obfuscapk[5] [24]. This tool takes an Android app (APK) as input and outputs a new APK with its bytecode obfuscated. The obfuscation to apply can be configured by selecting a combination of more than 20 obfuscation processes. These obfuscation processes range from light obfuscation (e.g., inserting `nop` instructions, removing debug information, etc.) to heavyweight obfuscation (e.g., replacing method calls to reflection calls, strings encryption, etc.).

To evaluate our approach's resiliency to obfuscation, we decided to use seven different obfuscation processes: (1) renaming classes, (2) renaming fields, (3) renaming methods, (4) encrypting strings, (5) overloading methods, (6) adding call indirection, (7) transforming method calls with reflection[6]. Thus, the resulting APK is considered to be highly obfuscated. Since Obfuscapk is prone to crash, we were not able to get the obfuscated version for some apps.

[3] https://play.google.com/store
[4] https://www.virustotal.com/
[5] https://github.com/ClaudiuGeorgiu/Obfuscapk
[6] https://www.oracle.com/technical-resources/articles/java/javareflection.html

We generate the images for the non-obfuscated dataset (the apps downloaded from AndroZoo), and the obfuscated samples. The image generation process is detailed in Section 2.1.

Since we compare our method with Drebin's approach, we have also extracted Drebin's features from the exact same apps we use to evaluate our approach. In our experiments, we only consider the apps from the non-obfuscated and the obfuscated datasets for which we have (a) successfully generated their images, and (b) successfully extracted their features for Drebin. Consequently, our final dataset contains 61 809 malware and 96 994 benign apps. We present in Table 1 a summary of our dataset.

Table 1. Dataset summary

	malicious apps	**benign apps**
Initial Set	71 194	134 134
Removed because of obfuscation failure	9023	34 629
Removed because of Image generation failure	4	2023
Removed because of DREBIN extraction failure	358	488
Final Set	61 809	96 994

3.3 Empirical Setup

Experimental validation. We evaluate the performance of DexRay using the Hold-out technique [25]. Specifically, in all our experiments, we shuffle our dataset and split it into 80% training, 10% validation, and 10% test. We train our model on the training dataset, and we use the apps in the validation set to tune the hyper-parameters of the network. After the model is trained, we evaluate its performance using the apps in the test set. This process is repeated ten times by shuffling the dataset each time and splitting it into training, validation, and test sets. We repeat the Hold-out technique in order to verify that our results do not depend on a specific split of the dataset.

We set to 200 the maximum number of epochs to train the network, and we stop the training process if the accuracy score on the validation dataset does not improve for 50 consecutive epochs. As for the models' parameters, we use `kernel_size=12`, and `activation=relu` for the two convolution layers, and we set their number of filters to 64 and 128 respectively. We also use `pool_size=12` for the two max-pooling layers, and `activation=sigmoid` for the two dense layers. The number of neurons in the first dense layer is set to 64. As for the output layer, it contains one neuron that predicts the class of the apps. We rely on four performance measures to assess the effectiveness of DexRay: Accuracy, Precision, Recall, and F1-score. Our experiments are conducted using the TensorFlow library[7].

State-of-the-art approaches to compare with.

Drebin *[9]:* We compare DexRay against Drebin, the state-of-the-art approach in machine learning based malware detection. Drebin extracts features

[7] https://www.tensorflow.org

that belong to eight categories: Hardware components, Requested permissions, App components, Filtered intents, Restricted API calls, Used permissions, Suspicious API calls, and Network addresses. These features are extracted from the disassembled bytecode and the Manifest file. The combination of extracted features is used to create an n-dimensional vector space where n is the total number of extracted features. For each app in the dataset, an n-dimensional binary vector space is created. A value of 1 in the vector space indicates that the feature is present in the app. If a feature does not exist in the app, its value is set to 0. DREBIN feeds the vectors space to a Linear SVM classifier, and it uses the trained model to predict if an app is malware or goodware. In this study, we use a replicated version [26] of DREBIN that we run on our dataset.

R2-D2 *[27]* is an Android malware detector that converts the bytecode of an APK file (i.e., DEX files) into RGB colour images. Specifically, the hexadecimal from the bytecode is translated to RGB colour. R2-D2 is a CNN-based approach that trains Inception-v3 [16] with the coloured images to predict malware. In their paper, the authors state that their approach is trained with more than 1.5 million samples. However, they do not clearly explicit how many apps are used in the test nor the size fixed for the coloured images. The authors provide a link to the materials of their experiment[8], but we could not find the image generator nor the original apps used to evaluate their approach in Section IV-C of their paper. Only the generated images are available. As a result, we were unable to reproduce their experiment to compare with our model. Instead, we compare directly with the results reported in their paper.

Ding et al. [28] proposes to convert the DEX files into (512, 512) grey-scale images in order to feed them to a deep learning model. The authors experiment with two CNN-based models: The first one, we note `Model1`, contains four convolutional layers, four pooling layers, a fully-connected hidden layer, and a fully-connected output layer. `Model2`, which is the second model, has the same architecture as `Model1` but with an additional high-order feature layer that is added just after the first pooling layer. Basically, this layer contains the multiplication of each two adjacent feature maps from the previous layer, as well as the feature maps themselves. Since neither the dataset nor the implementation of Ding et al.'s models is publicly available, we also rely on the results reported in Ding et al.'s manuscript.

4 Study Results

4.1 RQ1: How effective is DEXRAY in the detection of Android malware?

In this section, we assess the performance of DEXRAY on Android malware detection. We consider the performance against a ground truth dataset (the non-obfuscated apps introduced in Section 3.2) as well as a comparison against prior

[8] http://R2D2.TWMAN.ORG

approaches. We rely on the experimental setup presented in Section 3.3 and we test the performance of DexRay on 15 880 apps (10% of the non-obfuscated apps).

We report in Table 2 the scores as the average of Accuracy, Precision, Recall, and F1-score.

Table 2. Performance of DexRay against Drebin on our experimental dataset

	Accuracy	**Precision**	**Recall**	**F1-score**
DexRay	0.97	0.97	0.95	0.96
Drebin	0.97	0.97	0.94	0.96

Overall, as shown in Table 2, DexRay reaches an average score of 0.97, 0.97, 0.95, and 0.96 for Accuracy, Precision, Recall, and F1-score respectively. The reported results show the high effectiveness of DexRay in detecting Android malware. We further compare the performance of DexRay against three Android malware detectors in the following.

Comparison with Drebin. To assess the effectiveness of DexRay, we compare our results against a state-of-the-art Android malware detector that relies on static analysis: Drebin. Specifically, we evaluate Drebin using the same exact non-obfuscated apps we use to evaluate DexRay, and the same experimental setup described in Section 3.3. Moreover, we use the same split of the dataset for the Hold-out technique to evaluate the two approaches. We report the average of Accuracy, Precision, Recall, and F1-score of Drebin's evaluation in Table 2.

We notice that Drebin and DexRay achieve the same Accuracy, Precision, and the F1-score. As for the Recall DexRay slightly outperforms Drebin with a difference of 0.01.

Table 3. Authors-reported performance of R2-D2 and Ding et al. on different and less-significant datasets

	Accuracy	**Precision**	**Recall**	**F1-score**
DexRay	0.97	0.97	0.95	0.96
R2-D2	0.97	0.96	0.97	0.97
Ding et al.-Model 1	0.94	-	0.93	-
Ding et al.-Model 2	0.95	-	0.94	-

Comparison against other Image-based malware detection Approaches. We present in Table 3 the detection performance of R2-D2 and Ding et al.'s approaches as reported in their original publications. We note that DexRay and these two approaches are not evaluated using the same experimental setup and dataset. Specifically, R2-D2 is trained on a huge collection of 1.5 million apps, but it is evaluated on a small collection of 5482 images. In our experiments, we have conducted an evaluation on 15 880 test apps. We note that we have inferred the size of the test set based on R2-D2 publicly available images. Also, the scores we report for DexRay are the average of the scores achieved by ten different classifiers, each of which is evaluated on different test samples. R2-D2 scores are the results of a single train/test experiment which makes it difficult to properly compare the two approaches.

Similarly, Ding et al.'s experiments are conducted using the cross-validation technique, and both `Model1` and `Model2` are trained and evaluated using a small dataset of 4962 samples. Overall, we note that in terms of Recall, Precision and Accuracy, DexRay achieves performance metrics that are on par with prior work.

RQ1 answer: DexRay is a straightforward approach to malware detection which yields performance metrics that are comparable to the state of the art. Furthermore, it demonstrates that its simplicity in image generation and network architecture has not hindered its performance when compared to similar works presenting sophisticated configurations

4.2 RQ2: How effective is DexRay in detecting new Android malware?

Time decay [29] or model ageing [30] refers to the situation when the performance of ML classifiers drops after they are tested on new samples [31]. In this section, we aim to assess how does model ageing affect the performance of DexRay. Specifically, we investigate if DexRay can detect new malware when all the samples in its training set are older than the samples in its test set—a setting that Tesseract authors called *Temporally consistent* [29]. To this end, we split our dataset into two parts based on the date specified in the DEX file of the apps. The apps from 2019 are used to train and tune the hyper-parameters of the model, and the apps from 2020 are used for the test. The training and validation datasets consist of 113 951 and 28 488 apps respectively (i.e., 80% and 20% of 2019 dataset). As for the test dataset, it contains 16 364 malware and benign apps from 2020. We report our results on Table 4.

Table 4. Impact of model ageing on the performance of DexRay

	Accuracy	**Precision**	**Recall**	**F1-score**
DexRay results from RQ1	0.97	0.97	0.95	0.96
DexRay (Temporally Consistent)	0.97	0.97	0.98	0.98

We notice that DexRay detects new malware apps with a high detection rate. Specifically, it achieves detection scores of 0.97, 0.97, 0.98, 0.98 for Accuracy, Precision, Recall, and F1-score respectively. Compared to its effectiveness reported in RQ1 in Section 4.1, we notice that DexRay has reported higher Recall in this Temporally consistent experiment. This result could be explained by the composition of the malware in the training and test set. Indeed, malicious patterns in the test apps have been learned during the training that contains a representative set of Android malware from January to December 2019. The high performance of DexRay on new Android apps demonstrates its ability to generalise and its robustness against model ageing.

In this experiment, it is not possible to use the 10-times Hold-out technique because we only have one model trained on apps from 2019 and tested on new apps from 2020. To conduct an in-depth evaluation of the detection capabilities of this model, we also examine the receiver operating characteristic (ROC) curve. The ROC curve shows the impact of varying the threshold of the positive class on the performance of the classifier. It can be generated by plotting the true positive rate (TPR) against the false positive rate (FPR). We present in Figure 4 the ROC curve of our model.

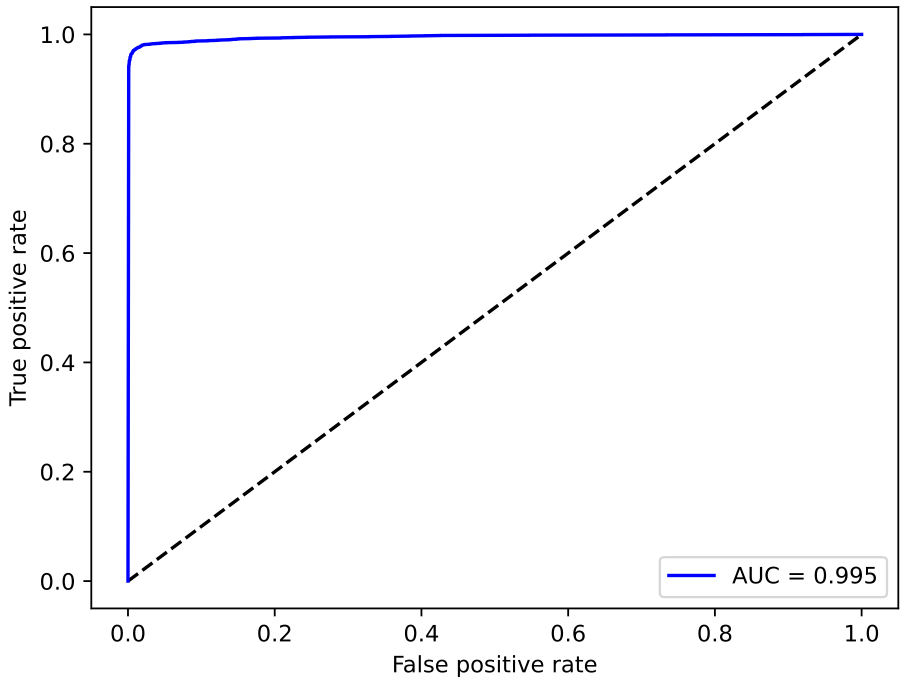

Fig. 4. ROC curve of DexRay evaluated against time decay

The ROC curve of DexRay further confirms its high effectiveness. As the area under the curve (AUC) can have a maximum value of 1, our approach has reached a very high AUC of 0.995.

It is noteworthy to remind that Tesseract [29] reported that Drebin performance was very significantly (negatively) impacted when tested in a *Temporally Consistent* setting.

RQ2 answer: DexRay's high performance is maintained in a *Temporally Consistent* evaluation. When tested on new Android apps, our approach has achieved very high detection performance, with an AUC of 0.995.

4.3 RQ3: What is the impact of image-resizing on the performance of DexRay?

In this section, we study the impact of image-resizing on the effectiveness of our approach. As we have presented in Section 2.1, we resort to resizing after mapping the bytecode bytes' to pixels in the image. Since the DEX files can have different sizes, resizing all images to the same size is necessary to feed our neural network. Resizing implies a loss of information. To better assess the impact of image-resizing, we evaluate the performance of DexRay using different image size values. Specifically, we repeat the experiment described in Section 4.1 using five sizes for our images: (1, 256*256), (1, 128*128), (1, 64*64), (1, 32*32), (1, 16*16). This experiment allows to assess the performance of DexRay over a large range of image sizes, i.e., from $2^8 = 256$ pixels to $2^{16} = 65536$ pixels. We present our results in Figure 5.

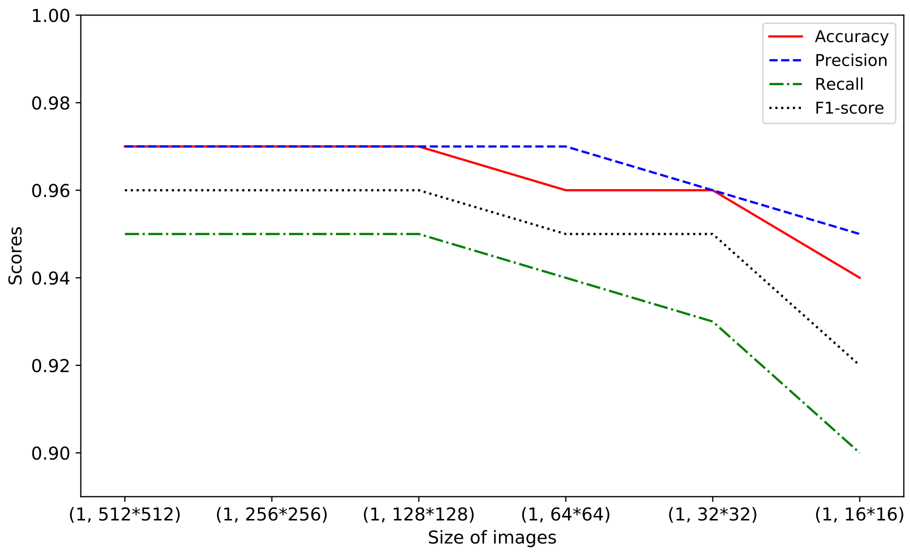

Fig. 5. The impact of image-resizing on the performance of DexRay

Overall, the size of images is a significant factor in the performance of DexRay. Unsurprisingly, the general trend is that the performance decreases as the image size is reduced: We notice that the values of the four metrics are lower for the three sizes that are smaller than our baseline (1, 128*128), with the worst performance being obtained with the smallest images. However, increasing the size from (1, 128*128) to either (1, 256*256) or (1, 512*512) does not improve the performance.

RQ3 answer: Image-resizing has a significant impact on the effectiveness of our approach. While downsizing decreases the evaluation scores values by up to 5%, increasing the size of our images does not bring significant performance benefits. Hence (1, 128*128) seems to be the sweet spot of DexRay.

4.4 RQ4: How does app obfuscation affect the performance of DexRay?

Malware detectors in the literature are often challenged when presented with obfuscated apps. We propose to investigate to what extent DexRay is affected by obfuscation. We consider two scenarios where: (1) the test set includes obfuscated apps; (2) the training set is augmented with obfuscated samples.

Performance on obfuscated apps when DexRay is trained on a dataset of non-obfuscated apps While our non-obfuscated dataset may contain obfuscated samples, we consider it as a normal dataset for training and we assess the performance of DexRay when the detection targets obfuscated apps. Specifically, we conduct two variant experiments to assess whether DexRay can detect obfuscated malware when: (1) It is tested on obfuscated apps that it has seen their non-obfuscated version in the training dataset, and (2) It is tested on obfuscated apps that it has NOT seen their non-obfuscated version in the training dataset. We consider both the non-obfuscated and the obfuscated samples, and we perform our experiments using the 10-times Hold-out technique described in Section 3.3.

For the first experiment, the non-obfuscated dataset is split into 90% training and 10% validation. The test set, which we note `Test1`, includes all the obfuscated apps. As for the second experiment, we design it as follows: We split the non-obfuscated dataset into 80% training, 10% validation, and 10% test. The training and the validation apps are used to train and tune DexRay hyper-parameters. We do not rely on the test images themselves, but we consider their obfuscated versions, which we note `Test2`. The average scores of the two experiments are presented in Table 5.

We have also evaluated Drebin using the same experimental setup in order to compare with DexRay. Its results are also presented in Table 5.

Table 5. Performance of DexRay & Drebin on the obfuscated apps

	Accuracy	**Precision**	**Recall**	**F1-score**
DexRay evaluated on `Test1`	0.64	0.64	0.17	0.26
DexRay evaluated on `Test2`	0.64	0.65	0.13	0.22
Drebin evaluated on `Test1`	0.94	0.94	0.91	0.93
Drebin evaluated on `Test2`	0.94	0.93	0.90	0.92

We notice that in the two experiments, DexRay does not perform well on the obfuscated apps detection. Its scores reported previously in Table 2 for malware detection have dropped remarkably, especially for the Recall that is decreased by at least 0.78. The comparison of Drebin's detection scores in Table 5 and

Table 2 suggests that its effectiveness is slightly decreased on the obfuscated apps detection. The scores reported in Table 5 are all above 0.9, which demonstrates that Drebin's overall performance on the obfuscated apps is still good. Drebin's results can be explained by the fact that it relies on some features that are not affected by the obfuscation process (e.g., requested permissions).

In the rest of this section, we investigate whether augmenting the training dataset with obfuscated samples can help DexRay discriminate the obfuscated malware.

Can augmenting the training dataset with obfuscated samples help to discriminate malware?

With this RQ, we aim to investigate if we can boost DexRay's detection via augmenting the training dataset with obfuscated apps. Specifically, we examine if this data augmentation can improve: (1) Obfuscated malware detection, and (2) Malware detection.

We conduct our experiments as follows: We split the non-obfuscated dataset into three subsets: 80% for the training, 10% for the validation, and 10% for the test. We augment the training and the validation subsets with `X%` of their obfuscated versions from the obfuscated dataset, with `X = {25, 50, 75, 100}`. As for the test dataset, we evaluate DexRay on both the non-obfuscated images and their obfuscated versions separately. Specifically, we assess whether augmenting the dataset with obfuscated apps can further enhance: (1) DexRay's performance on the detection of obfuscated malware reported in Table 5 (the test set is the obfuscated samples), and (2) DexRay's effectiveness reported in Table 2 (the test set contains non-obfuscated apps).

Similarly, we evaluate Drebin on the same experimental setup, and we report the average prediction scores of both DexRay and Drebin in Table 6.

Table 6. Performance of DexRay & Drebin after dataset augmentation

		Accuracy	**Precision**	**Recall**	**F1-score**
DexRay tested on Obf-apps	**25%**	0.95	0.96	0.92	0.94
	50%	0.96	0.97	0.92	0.95
	75%	0.96	0.97	0.93	0.95
	100%	0.96	0.97	0.94	0.95
Drebin tested on Obf-apps	**25%**	0.96	0.97	0.93	0.95
	50%	0.96	0.97	0.94	0.95
	75%	0.97	0.97	0.94	0.96
	100%	0.97	0.97	0.94	0.96
DexRay tested on non-Obf-apps	**25%**	0.97	0.97	0.94	0.96
	50%	0.97	0.97	0.94	0.96
	75%	0.97	0.97	0.94	0.96
	100%	0.97	0.97	0.94	0.95
Drebin test on non-Obf-apps	**25%**	0.97	0.97	0.94	0.96
	50%	0.97	0.97	0.94	0.96
	75%	0.97	0.97	0.94	0.96
	100%	0.97	0.97	0.94	0.96

We can see that DexRay detection scores on obfuscated samples increase remarkably when adding obfuscated apps to the training dataset. With 100% data augmentation, DexRay achieves a detection performance that is comparable to its performance on malware detection reported in Table 2. As for the impact of data augmentation on malware detection, we notice that the detection scores are stable. These results suggest that data augmentation boosts DexRay detection on obfuscated malware, but it does not affect its performance on malware detection (non-obfuscated apps).

As for Drebin, we notice that its performance on the obfuscated apps is also improved, and it is comparable to its performance on malware detection reported in Table 2. Similarly, its effectiveness on the non-Obfuscated apps after data augmentation is not enhanced, but it is stable.

RQ4 answer: Obfuscation can have a significant impact on the performance of DexRay. When the training set does not include obfuscated apps, the performance on obfuscated samples is significantly reduced. However, when the training set is (even slightly) augmented with obfuscated samples, DexRay can maintain its high detection rate.

5 Discussion

5.1 Simple but Effective

Prior work that propose image-based approaches to Android malware detection build on advanced "rectangular" image representations and/or complex network architectures. While they achieve high detection rates, the various levels of sophistication in different steps may *hide* the intrinsic contributions of the basic underlying ideas. With DexRay, we demonstrate that a minimal approach (i.e., with straightforward image generation and a basic CNN architecture) can produce high detection rates in detecting Android malware.

Our experimental comparison against the state-of-the-art detector Drebin further reveals that DexRay is competitive against Drebin. In the absence of artefacts to reproduce R2-D2 and Ding et al.'s approaches, our comparison focused on the detection scores reported by the authors. Note that while DexRay yields similar scores, both approaches involve a certain level of complexity: they both rely on 2-dimensional convolution that needs more computational requirements than the simple form of convolution leveraged by DexRay. Moreover, Ding et al.'s best architecture includes a high-order feature layer that is added to four extraction units (convolution/pooling layers). As for R2-D2, in addition to the coloured images that require a three-channel representation, it already leverages a sophisticated convolutional neural network with 42 layers [16]. Besides, 2-d convolution may not be suitable for image representation of code since pixels from one column of the image are not related. Such sophisticated design choices might affect the real capabilities of image-based malware detectors.

5.2 The Next Frontier in Malware Detection?

Selecting the best features for malware detection is an open question. Generally, every new approach to malware detection comes with its set of features, which are a combination of some known features and some novel hand-crafted feature engineering process. Manual feature engineering is however challenging and the contributions of various features remain poorly studied in the literature.

Recent deep learning approaches promise to automate the generation of features for malware detection. Nevertheless, many existing approaches still perform some form of feature selection (e.g., APIs, AST, etc.). Image-based representations of code are therefore an appealing research direction in learning features without a priori selection.

DexRay's effectiveness suggests that deep learned feature sets could lead to detectors that outperform those created with hand-crafted features. With DexRay, we use only the information contained in the DEX file, but still we achieve a detection performance comparable to the state of the art in the literature. This research direction therefore presents a huge potential for further breakthroughs in Android malware detection. For instance, the detection capability of DexRay can be further boosted using the image representation of other files from the Android APKs (e.g., the Manifest file). We have also revealed that DexRay is not resilient to obfuscation, which calls for investigations into adapted image representations and neural network architectures. Nevertheless, we have demonstrated that the performance of DexRay is not affected by the time decay. Overall, emerging image-based deep learning approaches to malware detection are promising as the next frontier of research in the field: with the emergence of new variants of malware, automated deep feature learning can overcome prior challenges in the literature for anticipating the engineering of relevant features to curb the spread of malware.

5.3 Explainability and Location Concerns

Explainable AI is an increasingly important concern in the research community. In deep learning based malware detection, the lack of explainability may hamper adoption due to the lack of information to enable analysts to validate detectors' performance. Specifically, with DexRay, we wonder: how even a straightforward approach can detect malware with such effectiveness? Are malicious behaviours that easy to distinguish? What do image-based malware detectors actually learn? The results of our work call for further investigation of the explainability of such approaches. Since neural networks are black-box models, explanation methods have been proposed to interpret their predictions (e.g., LIME [32], LEMNA [33]). Leveraging these explanation methods for image-based malware detection can serve to interpret their prediction, and further advance this research direction.

Following up on the classical case of the wolf and husky detector that turned out to be a snow detector [32], we have concerns as to whether image-based malware detectors learn patterns that humans can interpret as signs of maliciousness. A more general question that is raised is whether such approaches can eventually help to identify the relevant code that implements the malicious behaviour in an app. Malware location is indeed important as it is essential for characterising variants and assessing the severity of maliciousness.

5.4 Threats to validity

Our study and conclusions bear some threats to validity that we attempted to mitigate.

Threats to **external validity** refer to the generalisability of our findings. Our malware dataset indeed may not be representative of the population of malware in the wild. We minimised this threat by considering a large random sampling from AndroZoo, and by further requiring the consensus of only 2 antivirus engines to decide on maliciousness. Similarly, our results may have been biased by the proportion of obfuscated samples in our dataset. We have mitigated this threat by performing a study on the impact of obfuscation.

Threats to **internal validity** relate to the implementation issues in our approach. First, DexRay does not consider code outside of the DEX file. While this is common in current detection approaches that represent apps as images, it remains an important threat to validity due to the possibility to implement malware behaviour outside the main DEX files. Future studies should investigate apk to image representations that account for all artefacts. Second, we have relied on third-party tools (e.g., Obfuscapk), which fail on some apps, leading us to discard apps that we were not able to obfuscate or for which we cannot generate images. This threat is however mitigated by our selection of large dataset for experiments. Finally, setting the parameters of our experiments may create some threats to validity. For example, since Android apps differ in size, the generated images also have different sizes, which requires to resize all images in order to feed the Neural Network. The impact of image-resizing on the performance of our approach has been investigated in Section 4.3.

6 Related Work

Since the emergence of the first Android malware apps more than ten years ago [34], several researchers have dedicated their attention to develop approaches to stop the spread of malware. Machine Learning and Deep Learning techniques have been extensively leveraged by malware detectors using features extracted either using static, dynamic or hybrid analysis. We present these approaches in Section 6.1 and Section 6.2. We also review related work that leverages the image representation of code for malware detection in Section 6.3.

6.1 Machine Learning-based Android malware detection

Recent studies have been proposed to review and summarise the research advancement in the field of machine learning-based Android malware detection [35, 35, 36]. In 2014, Drebin [9] has made a breakthrough in the field by proposing a custom feature set based on static analysis. DREBIN trains a Linear SVM classifier with features extracted from the DEX files and the Manifest file. Similarly, a large variety of static analysis-based features are engineered and fed to machine learning techniques to detect Android malware (e.g., MaMaDroid [11], RevealDroid [10], DroidMat [37], ANASTASIA [12]). Dynamic analysis has also been leveraged to design features for malware detection (e.g., DroidDolphin [14], Crowdroid [38], DroidCat [13]). While the above detectors rely either on static or dynamic analysis, some researchers have chosen to rely on features that combine the two techniques (e.g., Androtomist [39], BRIDEMAID [15], SAMADroid [40]).

All the referenced approaches require feature engineering and pre-processing steps that can significantly increase the complexity of the approach. Our method learns from raw data and extracts features automatically during the training process.

6.2 Deep Learning-based Android malware detection

Deep Learning techniques have been largely adopted in the development of Android malware detectors. They are anticipated to detect emerging malware that might escape the detection of the conventional classifiers [41]. A recent review about Android malware detectors that rely on deep learning networks has been proposed [42]. MalDozer [43], a multimodal deep learning-based detector [44], DroidDetector [45], DL-Droid [46], Deep4MalDroid [47], and a deep autoencoder-based approach [48] are examples of DL-based detectors that predict malware using a variety of hand-crafted features. For instance, MalDozer [43] is a malware detection approach that uses as features the sequences of API calls from the DEX file. Each API call is mapped to an identifier stored in a specific dictionary. MalDozer then trains an embedding word model word2vec [49] to generate the feature vectors. DroidDetector [45] is a deep learning-based Android malware detector that extracts features based on hybrid analysis. Required permissions, sensitive APIs, and dynamic behaviours are extracted and fed to the deep learning model that contains an unsupervised pre-training phase and a supervised back-propagation phases. Opcode sequences have been extracted and projected into an embedding space before they are fed to a CNN-based model [50]. Again, most of these approaches require a feature engineering step and/or huge computation requirement that is not needed by DEXRAY.

6.3 Image-based malware detection

Different than traditional methods, some approaches have been proposed to detect malware based on the classical "rectangular" image-representation of source code. In 2011, a malware detection approach [51] that converts malware binaries into grey images, and extracts texture features using GIST [52] has been proposed. The features are fed to a KNN algorithm for classification purposes. Another study [53] has considered the same features for Android malware detection. Recently, a study [54] has been published about the use of image-based malware analysis with deep learning techniques.

In the Android ecosystem, an malware detection approach [55] with three types of images extracted from (1) Manifest file, (2) DEX file, and (3) Manifest, DEX and Resource.arsc files has been proposed. From each type of image, three global feature vectors and four local feature vectors are created. The global feature vectors are concatenated in one feature vector, and the bag of visual words algorithm is used to generate one local feature vector. The two types of vectors for the three types of images have been used to conduct a set of malware classification experiments with six machine learning algorithms. Similarly, an Android (and Apple) malware detector [56] trained using features extracted from grey-scale binary images has been proposed. The method creates the histogram of images based on the intensity of pixels and converts the histogram to 256 features vectors. The authors have experimented with different deep learning architectures, and the best results are achieved using a model with ten layers. R2D2 [27] and Ding et al. [28] have proposed malware detectors that are also based on image-learning and they are discussed in detail in Section 3.3. None of the above approaches has considered an image representation that preserve the continuity of the bytecode in DEX files. Moreover, they perform further pre-processing on the images before automatically extracting the features or rely on sophisticated ML or DL models for features extraction and classification. Our method converts the raw bytecode to "vector" images and feeds them directly to a simple 1-dimensional-based CNN model for features extraction and classification.

7 Conclusion

We have conducted an investigation to assess the feasibility of leveraging a simple and straightforward approach to malware detection based on image representation of code. DexRay implements a 1-dimensional convolution with two extraction units (Convolution/Pooling layers) for the neural network architecture. The evaluation of DexRay on a large dataset of malware and benign android apps demonstrates that it achieves a high detection rate. We have also showed that our approach is robust against time decay, and studied the impact of image-resizing on its performance. Moreover, we have investigated the impact of obfuscation on the effectiveness of DexRay and demonstrated that its performance can be further enhanced by augmenting the training dataset with obfuscated apps.

We have also compared DexRay against prior work on Android malware detection. Our results demonstrate that DexRay performs similarly to the state of the art Drebin and two image-based detectors that consider more sophisticated network architectures. The high performance of DexRay suggests that image-based Android malware detectors are indeed promising. We expect our work to serve as a foundation baseline for further developing the research roadmap of image-based malware detectors. We release the dataset of apps and the extracted images to the community. We also make DexRay source code publicly available to enable the reproducibility of our results and enable other researchers to build on our work to further develop this research direction.

Next steps: We expect sophisticated image representations and model architectures to offer higher performance gains compared to the studied baseline. This requires systematic ablation studies that extensively explore the techniques involved in the various building blocks of the pipeline (i.e., app preprocessing, image representation, neural network architecture, etc.) as well as the datasets at hand (e.g., with various levels of obfuscation). As part of the larger research agenda, the community should dive into the problem of interpretability of malware classification (with image representations) in order to facilitate malicious code localisation.

Data Availability

All artefacts are available online at:

https://github.com/Trustworthy-Software/DexRay

References

1. H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 479174, 2015.
2. T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proceedings of the Seventh European Workshop on System Security*, ser. EuroSec '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592791.2592796
3. M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, pp. 163–171.
4. P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: Robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, ser. SIN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 152–159. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/2523514.2523539
5. McAfee, "Mcafee labs threats report," 2020, accessed February 22, 2021. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-nov-2020.pdf
6. Google, "Android security & privacy 2018 year in review," 2018, accessed February 22, 2021. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf
7. Malwarebytes Lab, "2020 state of malware report," 2020, accessed February 22, 2021. [Online]. Available: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report-1.pdf
8. Kaspersky Lab, "Kaspersky security network," 2017, accessed February 22, 2021. [Online]. Available: https://media.kaspersky.com/pdf/KESB_Whitepaper_KSN_ENG_final.pdf

9. D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2014.
10. J. Garcia, M. Hammad, and S. Malek, "[journal first] lightweight, obfuscation-resilient detection and family identification of android malware," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 497–497.
11. L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Trans. Priv. Secur.*, vol. 22, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3313391
12. H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "Anastasia: Android malware detection using static analysis of applications," in *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016, pp. 1–5.
13. H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019.
14. W.-C. Wu and S.-H. Hung, "Droiddolphin: A dynamic android malware detection framework using big data and machine learning," in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, ser. RACS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 247–252. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/2663761.2664223
15. F. Martinelli, F. Mercaldo, and A. Saracino, "Bridemaid: An hybrid tool for accurate detection of android malware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 899–901. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3052973.3055156
16. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.
17. K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508
18. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539
19. R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into imaging*, vol. 9, no. 4, pp. 611–629, 2018.
20. W. Zhiqiang and L. Jun, "A review of object detection based on convolutional neural network," in *2017 36th Chinese Control Conference (CCC)*, 2017, pp. 11 104–11 109.
21. N. Aloysius and M. Geetha, "A review on deep convolutional neural networks," in *2017 International Conference on Communication and Signal Processing (ICCSP)*, 2017, pp. 0588–0592.
22. Q. Ke, J. Liu, M. Bennamoun, S. An, F. Sohel, and F. Boussaid, "Computer vision for human–machine interaction," in *Computer Vision for Assistive Healthcare*. Elsevier, 2018, pp. 127–145.

23. D. Yu, H. Wang, P. Chen, and Z. Wei, "Mixed pooling for convolutional neural networks," in *Rough Sets and Knowledge Technology*, D. Miao, W. Pedrycz, D. Ślęzak, G. Peters, Q. Hu, and R. Wang, Eds., Cham, 2014, pp. 364–375.
24. S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711019302791
25. S. Raschka, "Model evaluation, model selection, and algorithm selection in machine learning," *arXiv preprint arXiv:1811.12808*, 2018.
26. N. Daoudi, K. Allix, T. Bissyandé, and J. Klein, "Lessons learnt on reproducibility in machine learning based android malware detection," *Empirical Software Engineering*, vol. 26, 07 2021.
27. T. H. Huang and H. Kao, "R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 2633–2642.
28. Y. Ding, X. Zhang, J. Hu, and W. Xu, "Android malware detection method based on bytecode image," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–10, 2020.
29. F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 729–746. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury
30. K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 47–62.
31. X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 757–770. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3372297.3417291
32. M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.2939778
33. W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 364–379. [Online]. Available: https://doi.org/10.1145/3243734.3243792
34. P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen, "A pragmatic android malware detection procedure," *Computers & Security*, vol. 70, pp. 689–701, 2017.
35. K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124 579–124 607, 2020.
36. T. Sharma and D. Rattan, "Malicious application detection in android — a systematic literature review," *Computer Science Review*, vol. 40, p.

100373, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013721000137

37. D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu, “Droidmat: Android malware detection through manifest and api calls tracing,” in *2012 Seventh Asia Joint Conference on Information Security*, 2012, pp. 62–69.
38. I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: Behavior-based malware detection system for android,” in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 15–26. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/2046614.2046619
39. V. Kouliaridis, G. Kambourakis, D. Geneiatakis, and N. Potha, “Two anatomists are better than one—dual-level android malware detection,” *Symmetry*, vol. 12, no. 7, p. 1128, 2020.
40. S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu, “Samadroid: A novel 3-level hybrid malware detection model for android operating system,” *IEEE Access*, vol. 6, pp. 4321–4339, 2018.
41. Z. Wang, J. Cai, S. Cheng, and W. Li, “Droiddeeplearner: Identifying android malware using deep learning,” in *2016 IEEE 37th Sarnoff Symposium*, 2016, pp. 160–165.
42. J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, “A survey of android malware detection with deep neural models,” *ACM Comput. Surv.*, vol. 53, no. 6, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3417978
43. E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, “Maldozer: Automatic framework for android malware detection using deep learning,” *Digital Investigation*, vol. 24, pp. S48–S59, 2018.
44. T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, “A multimodal deep learning method for android malware detection using various features,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.
45. Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.
46. M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, “Dl-droid: Deep learning based android malware detection using real devices,” *Computers & Security*, vol. 89, p. 101663, 2020.
47. S. Hou, A. Saas, L. Chen, and Y. Ye, “Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs,” in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, 2016, pp. 104–111.
48. W. Wang, M. Zhao, and J. Wang, “Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 8, pp. 3035–3043, 2019.
49. T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf
50. N. McLaughlin, J. Del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupe, and G.-J. Ahn, “Deep android malware detection,”

in *CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy. Association for Computing Machinery, Inc, Mar. 2017, pp. 301–308, funding Information: This work was partially supported by the grants from Global Research Laboratory Project through National Research Foundation (NRF-2014K1A1A2043029) and the Center for Cybersecurity and Digital Forensics at Arizona State University. This work was also partially supported by Engineering and Physical Sciences Research Council (EPSRC) grant EP/N508664/1.; 7th ACM Conference on Data and Application Security and Privacy, CODASPY 2017 ; Conference date: 22-03-2017 Through 24-03-2017.

51. L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, “Malware images: visualization and automatic classification,” in *Proceedings of the 8th international symposium on visualization for cyber security*, 2011, pp. 1–7.
52. A. Oliva and A. Torralba, “Modeling the shape of the scene: A holistic representation of the spatial envelope,” *International journal of computer vision*, vol. 42, no. 3, pp. 145–175, 2001.
53. F. M. Darus, N. A. A. Salleh, and A. F. Mohd Ariffin, “Android malware detection using machine learning on image patterns,” in *2018 Cyber Resilience Conference (CRC)*, 2018, pp. 1–2.
54. B. Yadav and S. Tokekar, *Deep Learning in Malware Identification and Classification*. Cham: Springer International Publishing, 2021, pp. 163–205. [Online]. Available: https://doi.org/10.1007/978-3-030-62582-5_6
55. H. M. Ünver and K. Bakour, “Android malware detection based on image-based features and machine learning techniques,” *SN Applied Sciences*, vol. 2, no. 7, Jun. 2020. [Online]. Available: https://doi.org/10.1007/s42452-020-3132-2
56. F. Mercaldo and A. Santone, “Deep learning for image-based mobile malware detection,” *Journal of Computer Virology and Hacking Techniques*, vol. 16, no. 2, pp. 157–171, Jun 2020. [Online]. Available: https://doi.org/10.1007/s11416-019-00346-7