Guided Retraining to Enhance the Detection of Difficult Android Malware

Nadia Daoudi
nadia.daoudi@uni.lu
University of Luxembourg
Luxembourg

Kevin Allix
kevin.allix@centralesupelec.fr
CentraleSupelec
France

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu
University of Luxembourg
Luxembourg

Jacques Klein
jacques.klein@uni.lu
University of Luxembourg
Luxembourg

ABSTRACT

The popularity of Android OS has made it an appealing target for malware developers. To evade detection, including by ML-based techniques, attackers invest in creating malware that closely resemble legitimate apps, challenging the state of the art with *difficult-to-detect* samples. In this paper, we propose Guided Retraining, a supervised representation learning-based method for boosting the performance of malware detectors. To that end, we first split the experimental dataset into subsets of "easy" and "difficult" samples, where difficulty is associated to the prediction probabilities yielded by a malware detector. For the subset of "easy" samples, the base malware detector is used to make the final predictions since the error rate on that subset is low by construction. Our work targets the second subset containing "difficult" samples, for which the probabilities are such that the classifier is not confident on the predictions, which have high error rates. We apply our Guided Retraining method on these difficult samples to improve their classification. Guided Retraining leverages the correct predictions and the errors made by the base malware detector to guide the retraining process. Guided Retraining learns new embeddings of the difficult samples using Supervised Contrastive Learning and trains an auxiliary classifier for the final predictions. We validate our method on four state-of-the-art Android malware detection approaches using over 265k malware and benign apps. Experimental results show that Guided Retraining can boost state-of-the-art detectors by eliminating up to 45.19% of the prediction errors that they make on difficult samples. We note furthermore that our method is generic and designed to enhance the performance of binary classifiers for other tasks beyond Android malware detection.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Machine learning**.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598123

KEYWORDS

Android, malware, retraining, difficult samples

ACM Reference Format:
Nadia Daoudi, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2023. Guided Retraining to Enhance the Detection of Difficult Android Malware. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598123

1 INTRODUCTION

Android malware plays hide and seek with mobile applications markets operators. Indeed, new emerging malware apps are increasingly sophisticated [11, 19, 23] and challenge state-of-the-art detection techniques, in particular literature ML-based approaches. These malware apps are designed to closely resemble benign apps in order to hide their malicious behaviour and evade detection. In typical ML-based malware detection schemes, Android apps are represented using feature vectors (i.e., apps are embedded), which are fed to an algorithm that learns to distinguish malware and benign samples. In such an embedding space, some malware (or benign) samples occupy a distinct region of the input space [54]. These samples share similar feature vectors that make them easily distinguishable and separable from the benign (respectively malware) apps in the embedding space. Nevertheless, there are other malware apps which have feature vectors that are similar to feature vectors of benign samples. Such apps are located in regions of the embedding space where malware and benign samples are not perfectly separable and distinguishable. In such regions, malware and benign apps overlap, which leads to misclassifications.

Deep representation learning aims to extract relevant patterns from the input data and discard the noise. Several techniques [20, 22, 36, 46, 47] have leveraged the class labels to generate powerful representations, which has led to state-of-the-art performance. Indeed, supervised representation learning methods are trained to automatically learn characteristic features of samples that share the same class labels. The resulting embeddings are passed to a classifier that maps the samples to their respective classes. Recently, Supervised Contrastive Learning [22] has been proposed to maximise the embedding similarity of samples from the same class and minimise the embedding similarity of samples belonging to different classes. This representation learning method transforms the input data into an embedding space in which samples with

the same labels are close to each other, so they can have similar representations. Furthermore, it increases the distance between samples from different classes so they can get distinct representations. Supervised Contrastive Learning seems to propose a solution for overlapping malware and benign samples since it transforms the input data into a new embedding space in which samples from the same class are grouped together and separated from the other class.

In binary classification, we can distinguish between two categories of samples based on their input labels: positives and negatives (e.g., malware and benign). It is also possible to classify samples into *easy* and *difficult* instances based on their feature vectors. Easy samples refer to positive and negative instances which a classifier can easily identify and correctly predict their classes. The difficult samples can also be positives or negatives, but they have similar input features that make it challenging for the classifier to correctly identify their classes. The notion of difficulty is related to the malware detector itself (i.e., its features set and ML algorithm). Specifically, depending on the features and the classification algorithm leveraged by a malware detector, a malware app might be difficult to detect by one approach but easy to detect by another. For a base classifier, identifying the class of the easy samples would be straightforward, which results in low prediction errors. For the difficult samples, they would need more advanced techniques to better discriminate the two classes.

In this paper, we investigate the feasibility of boosting existing malware detectors by focusing on difficult-to-detect samples. To that end, we explore the power of contrastive learning with the idea of further guiding the learning to build embeddings where samples that were previously close to samples of other classes are now clearly separated in the embedding space. We propose to address the problem of malware classification in two steps: The first step of the classification involves the samples that are easy to predict by a base classifier. To decide whether a sample is easy or difficult to predict, we rely on the prediction probabilities yielded by the base classifier. Thus, all samples that are identified as easy (i.e., with high prediction probabilities from the base classifier), are simply left to be predicted by the base classifier. If a sample is identified as difficult (i.e., with low prediction probabilities from the base classifier), then it is passed to the second step where an auxiliary classifier trained via our Guided Retraining method is meant to address its final prediction. Note that we use the term *"Retraining"* to refer to the task which consists in training a new classifier on a given dataset. As its name suggests, our technique is designed to guide the retraining on the difficult samples to reduce the prediction errors. We rely on the predictions generated by the base classifier on the difficult samples to learn distinctive representations for each class. Specifically, we leverage Supervised Contrastive Learning to generate embeddings for the difficult samples in five guided steps that teach the model to learn from the correct predictions and errors made by the base classifier. Then, we train an auxiliary classifier on the generated embeddings so it can make the final classification decision on the difficult samples.

To validate the effectiveness of our method, we evaluate it on four state-of-the-art Android malware detectors (i.e., with their variants) that were successfully replicated in the literature [7]: DREBIN [4], RevealDroid [15], MaMaDroid [31], and MalScan [50]. These detectors consider various features to discriminate between malware and benign apps, and they have been reported to be highly effective. Our experiments demonstrate that the prediction errors made by state-of-the-art Android malware detectors can be reduced via our Guided Retraining method. Specifically, we show that our technique boosts the detection performance and reduces up to 45.19% prediction errors made by the classifiers.

We have also assessed the effectiveness of our method in boosting the detection performance on new Android apps. For instance, we have trained the state-of-the-art DREBIN on samples from 2019 and tested its performance on apps from 2020. Our results showed that DREBIN achieves an F1 score of 85.31%. After using our Guided Retraining approach on DREBIN, it was able to detect **769** malicious samples that escaped its detection in the first place. 70% of these malicious samples (i.e., 535 apps) were originally collected from the Google Play Store and belong to different malware families such as: "jiagu", "blacklister" and "dnotua". Overall, our results show that Guided Retraining is an effective method to reduce the misclassifications of state-of-the-art Android malware detectors.

Our contributions can be summarised as follows:

- We propose to address the malware detection problem in two steps: the first step deals with the detection of the easy samples, and the second step is intended for the difficult-to-detect apps;
- We design a new technique, Guided Retraining, that improves the classification of the difficult-to-detect apps by yielding contrasted representations;
- We validate the effectiveness of our method on four state-of-the-art Android malware detectors;
- We make our code and dataset publicly available at: https://github.com/Trustworthy-Software/GuidedRetraining

2 APPROACH

2.1 Overview

Our method aims to leverage deep learning techniques in order to boost the performance of a binary base classifier. We present in Figure 1 an overview of our method. The first step consists of training a base classifier on the whole training dataset. Then, we leverage the prediction probabilities of the base classifier to split the dataset into two subsets: easy and difficult samples. The difficult samples are used to train an auxiliary classifier via our Guided Retraining method. The motivation behind the auxiliary classifier is to obtain a "specialised classifier" that will improve the performance on difficult samples.

Given a new sample, if it is identified as an easy sample, it will be predicted by the base classifier. Otherwise, the prediction decision will be made by the auxiliary classifier that is trained on the difficult samples via our Guided Retraining method.

More specifically, the overall process of our approach can be summarised as follows:

(1) Train a base classifier on the training set (step 1 in Figure 1)
(2) Use this classifier to collect the "difficult" samples (i.e., those samples from the training set that are close to the decision boundary of the trained classifier). This is illustrated in step 2 in Figure 1; Concretely, to do this we apply the classifier on

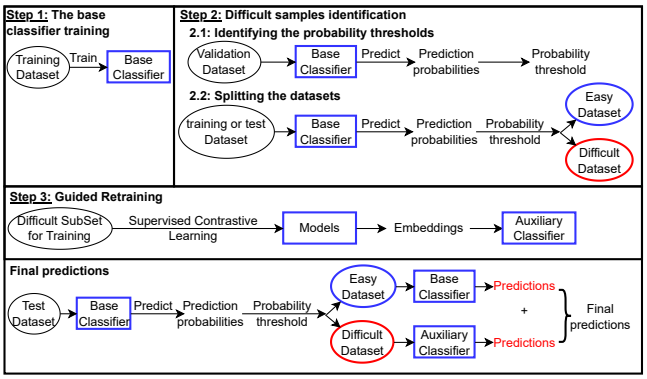

Figure 1: An overview of our approach

its own training set (which is uncommon, but useful here to identify the "difficult" samples).

(3) Devise a new embedding, specialized to contrast "difficult goodware" from "difficult malware" (step 3 in Figure 1).

(4) Train another classifier on the devised embeddings of the difficult samples in the training set. This classifier is the auxiliary classifier (step 3 in Figure 1).

To use our approach on a new sample: the sample would first be classified by the base classifier. If it turns out this is a difficult sample, then (and only then), we would instead use the auxiliary, specialized, classifier. In the following, we describe the main steps of our approach which are: The base classifier training, Difficult samples identification, and Guided Retraining.

2.2 The Base Classifier Training

Our approach is designed to boost the performance of an existing binary classifier that we denote as the base classifier. The type of this classifier is not important, but ideally it should be able to output the prediction probabilities, i.e., not only a binary classification (such as *malware* or *benign*) but a value, typically between 0 and 1, that indicates the likelihood that a given sample is a malware. If the classifier does not generate prediction probabilities, we propose other solutions in Section 3.5.

The first step consists in splitting the dataset into three subsets: training, validation, and test. We train the base classifier using all the samples in the training subset.

2.3 Difficult Samples Identification

The aim of this step is to identify the samples that are "difficult" to predict by the base classifier. The criteria we use to identify these samples is their probabilities of prediction.

In a binary classification experiment, if the model is confident about the label of a given sample, it assigns a high prediction probability to the class that is associated with that label (i.e., a probability of prediction that is close to 1). Otherwise, samples from any of the two classes get similar probabilities of prediction (i.e., the probabilities of prediction for the two classes are close to 0.5). The predicted labels are then decided based on the probabilities of predictions. Generally, when the probability of prediction for the positive class (or the negative class) is higher than 0.5, the classifier predicts the sample as positive (or negative). Since the probability of prediction for the negative class can be deduced from the probability of prediction of the positive class (i.e., the two probabilities sum up to 1), we consider only the probability of prediction of the positive class in the following, and we denote it p.

In our approach, we leverage the probabilities of prediction to split a dataset into easy and difficult subsets. After it is trained, the base classifier would assign either a very high or a very low probability of prediction p to the samples that it can predict their labels with a high confidence. Specifically, if p is very high, the base classifier is confident that the sample belongs to the positive class. Conversely, if p is very low, the classifier is confident that the sample belongs to the negative class. If a given sample is attributed a very high or a very low probability of prediction, we consider it as an easy sample. Otherwise, it is considered to belong to the difficult subset.

> We postulate that easy and difficult subsets have the following properties:
> **Easy subset:** Applying a base classifier on the samples of this subset will yield only a few prediction errors.
> **Difficult subset:** Applying a base classifier on the samples of this subset will mostly yield prediction errors.

2.3.1 On Applying the Base Classifier on Its Own Training Set. In practice, to identify difficult and easy samples, we apply the base classifier on its own training set, i.e., we collect the prediction probabilities to decide whether the samples are easy or difficult. We acknowledge that using a trained model (i.e., the base classifier) on its own training set is uncommon, and would be absurd in most cases. Here, however, we only do it to identify which samples are close to the decision boundary of the classifier (i.e., when the classifier "is unsure"), and later to improve the training on difficult samples by learning new embeddings (Cf. Section 2.4). It is important to note that this design conforms to traditional ML processes where the training set is clearly separated from the test set (i.e., there is no data leakage).

2.3.2 Identifying the Probability Thresholds. From the previous step, our base classifier has attributed a probability of prediction to each sample in the training and validation datasets. The next step consists of tagging each sample in the dataset as easy or difficult based on its probability of prediction. To this end, we need to identify two thresholds for considering a sample as easy or difficult. Specifically, we rely on one probability threshold to decide whether the prediction probability p of a given sample is high enough to consider that sample as easy (i.e., in this case the sample is an *easy positive* since p is high). Similarly, when the prediction probability p of a given sample is small, we need another probability threshold to decide whether p is small enough to tag the sample as easy (i.e., in this case the sample is an *easy negative*).

We rely on the validation dataset to determine the values of the two probability thresholds. Specifically, since the validation samples are classified into TNs (i.e., True Negatives), FPs (i.e., False Positives), FNs (i.e., False Negatives), and TPs (i.e., True Positives), we determine the probability thresholds that satisfy the following constraints:

- The probability threshold for considering a sample as an easy positive must ensure that the number of false positives in the easy validation dataset is equal to $X\%$ of the total number of FPs (i.e., FPs predicted by the base classifier on the whole validation dataset). We denote this threshold th_p.
- The probability threshold for classifying a sample as an easy negative must guarantee that the number of false negatives in the easy validation dataset is equal to $Y\%$ of the total number of FNs (i.e., FNs predicted by the base classifier based on the whole validation dataset). We denote this threshold th_n.

To identify the values of the two probability thresholds, we need to compute the number of FPs and FNs that we tolerate in the easy validation dataset. We note these variables $toleratedFPs$ and $toleratedFNs$ and we calculate their values as follows:

$$toleratedFPs = \frac{X \times FP_v}{100}; toleratedFNs = \frac{Y \times FN_v}{100}$$

where FP_v and FN_v represent the number of FPs and FNs returned by the base classifier on the whole validation dataset, respectively.

The process of identifying the two probability of prediction thresholds is adequately detailed in Algorithm 1.

Algorithm 1: Thresholds selection

```
Input: vDataset, yProbabilities, toleratedFPs, toleratedFNs, indicesOfFPs,
  indicesOfFNs
Output: thresholdFPs, thresholdFNs
counterFPs ← 0
counterFNs ← 0
lenData ← vDataset.length()
probasIndicesPos ← ∅
probasIndicesNeg ← ∅
for i ← 1, lenData do
    if yProbabilities(i) ≥ 0.5 then
        probasIndicesPos ← probasIndicesPos + (yProbabilities(i), i)
    // We keep track of the index of the sample to verify that it is not among
      the FPs and FNs. We later search that index in indicesOfFPs and
      indicesOfFNs lists
    else
        probasIndicesNeg ← probasIndicesNeg + (yProbabilities(i), i)
probasIndicesPos ← probasIndicesPos.inverselySortProbas()
// The prediction probabilities of the positive samples are sorted in
  descending order
probasIndicesNeg ← probasIndicesNeg.sortProbas()
// The prediction probabilities of the negative samples are sorted in ascending
  order
lenPos ← probasIndicesPos.length()
lenNeg ← probasIndicesNeg.length()
for i ← 1, lenPos do
    if counterFPs == toleratedFPs then
        thresholdFPs ← probasIndicesPos[i][0]
        break
    if probasIndicesPos[i][1] in indicesOfFPs then
        counterFPs ← counterFPs + 1
for i ← 1, lenNeg do
    if counterFNs == toleratedFNs then
        thresholdFNs ← probasIndicesNeg[i][0]
        break
    if probasIndicesNeg[i][1] in indicesOfFNs then
        counterFNs ← counterFNs + 1
```

The inputs to this algorithm are the validation dataset, the probabilities of prediction returned by the base classifier on the validation dataset, toleratedFPs, toleratedFNs, and the indices of the FP_v and FN_v in the validation dataset (i.e., we consider that each instance in the dataset has a unique index, and we denote the lists of the FPs and FNs indices as indicesOfFPs and indicesOfFNs respectively). To identify the threshold of the positives, we first select all the samples from the validation dataset that have their $p \geq 0.5$ and we sort their probabilities in descending order. We also keep track of the indices of these samples in the validation dataset to verify whether they are predicted as TPs or FPs by the base classifier (i.e., based on indicesOfFPs list). Then, we initialise a counter of the number of FPs in the easy dataset and we iterate over the sorted samples starting from the one with the highest probability of prediction. During each iteration, we first check whether the value of the FPs counter has reached the number of toleratedFPs, in which case we stop the iteration and set the th_p to the current probability of prediction. Otherwise, we increment the counter of FPs if the sample has been predicted as FP by the base classifier.

We apply the same technique to identify the value of negatives threshold th_n. We select the samples that have their $p \leq 0.5$ and we sort their probabilities in ascending order since the classifier is confident about the samples with low probabilities of prediction. Similarly, we keep a counter for the number of FNs that are tolerated in the easy dataset and we iterate over the sorted samples starting from the one with the lowest probability of prediction. When the value of the FNs counter is equal to the value of toleratedFNs, we stop the iteration. We then set the value of th_n to the probability of prediction of the last sample in which the iteration stopped.

2.3.3 Splitting the Datasets. After identifying the values of th_p and th_n, we split our datasets into easy and difficult subsets. The easy dataset contains all the samples whose probabilities of prediction satisfy:

$$easyDataset = \{x_i \in dataset \mid th_n \geq p_{\mathrm{i}} \text{ or } th_p \leq p_{\mathrm{i}}\}$$

where p_{i} is the probability of prediction of sample x_i.

The easy dataset includes all the positive samples whose prediction probabilities are greater than the threshold th_p (i.e., they are predicted as positives with high confidence by the base classifier). It also includes the negative samples whose prediction probabilities are smaller than the threshold th_n (i.e., they are predicted as negatives with high confidence by the base classifier).

As for the difficult dataset, it contains all the samples that do not satisfy the constraints of the easy dataset. Specifically, it includes the samples whose prediction probabilities are at the same time below the threshold th_p and above the threshold th_n (i.e., the base classifier is not confident that these samples are positives or negatives). The samples in the difficult dataset satisfy:

$$difficultDataset = \{x_i \in dataset \mid th_n < p_{\mathrm{i}} < th_p\}$$

At the end of this step, we have the training, validation, and test datasets split into easy and difficult subsets.

2.4 Guided Retraining

In our approach we make use of Supervised Contrastive Learning [22] to generate the embeddings of the difficult samples. Contrastive Learning is a technique that generates new embeddings of the dataset in such a way that samples belonging to the same class are close to each other in the embedding space. Similarly, samples belonging to different classes are far from each other in the embedding space. While Contrastive Learning [22] has been proposed for the general case of multi-class classification, we have adapted it to

the special case of having only two classes: malware and benign (i.e., positive and negative). Supervised Contrastive Learning works in two stages: First, it generates the embeddings using an Encoder followed by a Projection Network (we refer to both of them as the `Model`). After the training is done, the Projection Network is discarded and a classifier is trained on the embeddings from the last layer of the Encoder. This classifier is referred to as the auxiliary classifier. At the end of the second stage, the samples are classified into their respective classes. Using Supervised Contrastive Learning, we aim to create contrasted representations for the samples in the difficult subsets which would help to better classify them into their respective classes.

From the previous step (i.e., Section 2.3), we have created two validation subsets: easy and difficult. By construction, the difficult validation subset contains most of the misclassified samples yielded by the base classifier. Specifically, it contains (100 - X)% of the total number of FPs contained in the whole validation dataset. Likewise, the number of FNs reaches (100 - Y)% of the total number of FNs in the validation dataset. The difficult subset for training is also expected to include similar proportions of FPs and FNs (i.e., it includes most of the prediction errors from the whole training dataset). We remind that the difficult subsets also contain correctly predicted samples yielded by the base classifier. In the following, we use TN'_{tr}, FP'_{tr}, FN'_{tr}, and TP'_{tr} to refer to TNs, FPs, FNs, and TPs of the base classifier on the difficult subset for training.

As the title suggests, we propose a method that would guide the retraining on the difficult samples. Specifically, we aim to help the `Model` distinguish between four categories of samples in the difficult training subset. These categories are: TN'_{tr}, FP'_{tr}, FN'_{tr}, and TP'_{tr}. We present in Figure 2 an overview of our Guided Retraining approach.

Since training a binary classifier requires a dataset that contains samples from two classes (i.e., positives and negatives), we make use of the different combinations of subsets in the difficult training subset to help the `Model` generate more contrasted embeddings. Specifically, we first train a `Model` using TP'_{tr} (i.e., they have positive real labels) and FP'_{tr} (i.e., they have negative real labels), and we denote it $\texttt{Model}_1$. Basically, we guide $\texttt{Model}_1$ to distinguish between the positive samples that are correctly predicted by the base classifier and the negative samples that are all misclassified by the same classifier. Consequently, $\texttt{Model}_1$ focuses on learning a contrasted representation for the true positives and the false positives in the difficult subset for training. Then, we train another `Model` using TN'_{tr} (i.e., they have negative real labels) and FN'_{tr} (i.e., they have positive real labels) and we denoted it $\texttt{Model}_2$. This `Model` would learn to distinguish between the true negatives and the false negatives predicted by the base classifier on the difficult subset for training. Similarly, we train $\texttt{Model}_3$ on TP'_{tr} (i.e., they have positive real labels) and TN'_{tr} (i.e., they have negative real labels), and $\texttt{Model}_4$ on FP'_{tr} (i.e., they have negative real labels) and FN'_{tr} (i.e., they have positive real labels).

In summary, the four `Models` are trained on two difficult training subsets that the base classifier has: (1) either correctly or incorrectly classified both of them, (2) correctly predicted one subset and misclassified the other subset.

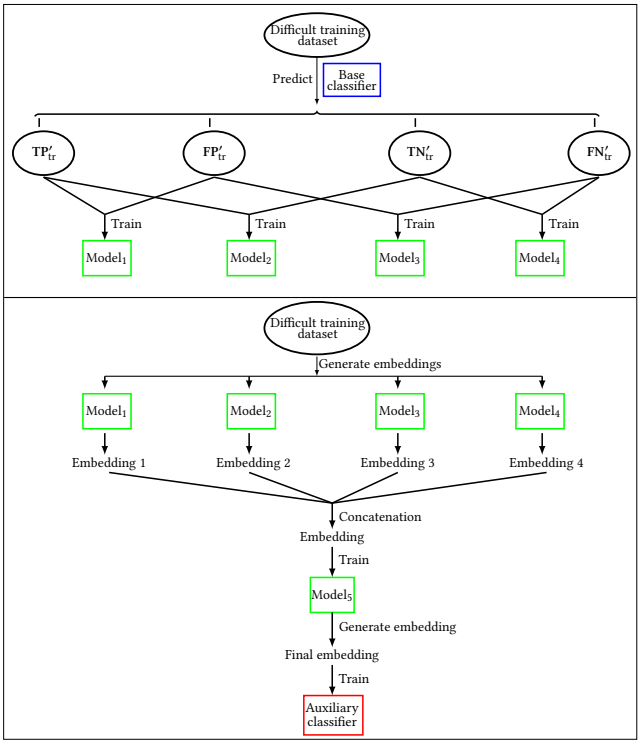

Figure 2: An illustration of our Guided Retraining method

After the four `Models` are trained, they are used to generate embeddings for the difficult training subset. Specifically, four embeddings are generated for each sample in the difficult training subset. Then, we concatenate the four feature representations of each sample into one vector in order to have one embedding per sample.

To create more contrasted representations for the difficult samples, we train another `Model` on the concatenated embeddings and we denote it $\texttt{Model}_5$. Basically, $\texttt{Model}_5$ is trained on all the samples from the difficult training subset, which would create fine-grained contrasted representations based on the embeddings generated by the four previous `Models`. Indeed, $\texttt{Model}_5$ would learn from the concatenated embeddings of each sample in the difficult subset (i.e., regardless if the base classifier has correctly or incorrectly predicted it) to generate the final feature representations.

The last step in our approach is to train the auxiliary classifier on the difficult training embeddings that are generated by $\texttt{Model}_5$. This classifier is trained on all the difficult samples in the training subset. The final classification decision of the difficult samples is given by the auxiliary classifier. We remind that for the easy dataset, it is the base classifier that is in charge of predicting their class labels, as illustrated in Figure 1.

3 EVALUATION SETUP

In this section, we first present the research questions we investigate in our study and the evaluation subjects we use to assess the effectiveness of our approach. Then, we describe the dataset, the architecture of both the `Models` and the auxiliary classifier, and we

overview our experimental setup (i.e., models' hyperparameters and implementation details).

3.1 Research Questions

In our study, we investigate the possibility of selecting and separating the samples that are most challenging to classify. Specifically, we aim to identify the difficult subset in a dataset that would contain samples that yield most of the prediction errors.

- **RQ1:** *To what extent is it feasible to split a dataset into two subsets, one with fewer prediction errors and one with most errors?*

After identifying the difficult subset in a dataset, we assess the impact of Guided Retraining on the detection performance and we compare it to other classic retraining methods.

- **RQ2:** *How effective is Guided Retraining in improving the classification results of state-of-the-art malware detectors?*

Additionally, we evaluate the effectiveness of Guided Retraining in detecting new Android malware.

- **RQ3:** *How effective is Guided Retraining in improving the classification performance on new Android apps?*

Finally, we investigate the impact of the errors thresholds used to construct the difficult and the easy subsets.

- **RQ4:** *What is the impact of the errors thresholds on the detection performance of Guided Retraining?*

We note that we have also conducted an ablation study to assess the importance of Guided Retraining's components. Due to space limitation, we provide the results of the study in our repository: https://github.com/Trustworthy-Software/GuidedRetraining

3.2 Evaluation Subjects

To evaluate the effectiveness of our approach in boosting the performance of base classifiers, we conduct our experiments on classifiers trained to detect Android malware. Specifically, we apply our method on four state-of-the-art Android malware detectors from the literature: DREBIN [4], RevealDroid [15], MaMaDroid [31] (using two variants: MaMaDroid Family and MaMaDroid Package), and MalScan [50] (i.e., two variants: MalScan Average and MalScan Concatenate). These detectors have been successfully replicated [7] in a study that has considered malware detectors from leading venues in Security, Software Engineering, and Machine Learning. We present a brief description of the features set and the ML algorithms used by these approaches in Table 1 and we refer the reader to the replication study [7] for further details.

Table 1: Evaluation subjects

	ML algorithm	Features set
DREBIN	LinearSVC	App Components, Filtered Intents, Hardware Components, Network Addresses, Restricted API Calls, Requested Permissions, Suspicious API Calls and Used Permissions
RevealDroid	LinearSVC	Android API usage, Native Call and Reflective Features
MaMaDroid	Random Forest	The representation of the abstracted API calls as Markov Chain
MalScan	KNN	Call graphs are represented as social networks to conduct centrality analysis

3.3 Dataset

We conduct our experiments on a public dataset of Android malware and benign apps from the literature [8]. It has been collected from AndroZoo [2], which is a growing collection that contains more than 22 million apps crawled from different markets, including Google Play. In this dataset, benign apps are defined as apps that have not been flagged by any antivirus engine from VirusTotal [45]. A sample is labelled as malware if it is flagged by at least two antivirus engines. The apps in this dataset are created between 2019 and 2020 (i.e., according to their compilation date). In total, the dataset contains 78 002 malware and 187 797 benign apps.

3.4 `Model` and Auxiliary Classifier Architectures

In this section, we present the neural network architecture we adopt for the `Models` and the auxiliary classifier, which are both based on the multi-layer perception (MLP).

3.4.1 The `Model`. As stated in Section 2, we use `Model` to refer to the Encoder and the Projection Network, that we train to generate contrasted embeddings of the difficult samples. For the Encoder, our MLP contains five fully connected layers[1] that have 2048, 1024, 512, 256, and 128 neurons, respectively. The outputs from each layer are normalised and passed through a RELU activation function. The size of the input in the Encoder is not fixed since it depends on the size of the feature vectors of each approach.

For the Projection Network, we use a two layers MLP[2] that receives normalised inputs from the Encoder. The first layer has 64 neurons with a RELU activation function and the output layer contains 32 neurons. After it is trained, only the embeddings at the last layer of the Encoder are considered [22]. Consequently, the size of the feature vectors generated by the `Models` is 128.

3.4.2 The Auxiliary Classifier. This neural network is used to classify the samples using the embeddings generated by $\texttt{Model}_5$. It contains five layers with 64, 32, 16, 8, and 2 neurons, respectively. The RELU activation function is applied to the normalised output of the first four layers. Since we conduct our experiments on binary classifiers, the last layer contains two neurons with a Sigmoid activation function (i.e., to output prediction probabilities for the two classes).

3.5 Experimental Setup

We conduct our experiments using PyTorch [35] and scikit-learn [37] libraries. For the base classifiers training step (i.e., Section 2.2), we split the dataset into training (80%), validation (10%), and test (10%), and we rely on the implementation of the evaluation subjects from the replication study [7]. In our experiments, we set the percentage of FPs and FNs tolerated in the easy dataset (i.e., the values of the parameters X and Y described in Section 2.3) to 5%. We also study the impact of these two parameters in Section 4.4.

For training the `Models` and the auxiliary classifiers, we leverage a publicly available implementation [43] of Supervised Contrastive Learning. We set 2000 as the maximum number of epochs, and we

[1] We were inspired by the Contrastive Learning implementation [43] that relies on ResNet-50. We have replaced the embedding layer and each of the four basic blocks with fully connected layers.
[2] We considered the same number of layers used in the CL implementation [43]

Table 2: Size of input vectors, the number of samples and the number of FPs and FNs in the test subsets

	Size of input vectors in the difficult datasets	Number of samples in the test dataset (i.e., benign:18 739 and malware: 7841)				Number of FPs and FNs in the test dataset					
		Easy dataset		Difficult dataset		Whole dataset		Easy dataset		Difficult dataset	
		benign	malware	benign	malware	FPs	FNs	FPs	FNs	FPs	FNs
DREBIN	1 184 063	7824	4146	10 915	3695	154	419	7	26	147	393
RevealDroid	7 882 350	6120	5308	12 619	2533	90	625	2	37	88	588
MaMaDroid Family	65	7770	743	10 969	7098	52	734	3	33	49	701
MaMaDroid Package	198 916	9642	178	9097	7663	136	322	4	10	132	312
MalScan Average	21 986	13 856	5621	4883	2220	243	420	25	40	218	380
MalScan Concatenate	87 944	13 760	5619	4979	2222	187	414	30	33	157	381

stop the training if the optimised metric (i.e., the loss for the `Models` and the accuracy for the auxiliary classifier) does not improve after 100 epochs [3]. We also set the batch size to the size of the training dataset divided by 10. Due to the huge size of the input vectors of some evaluated approaches, we had to divide their training size by 20, so the dataset could fit into memory. For the learning-rate hyper-parameter, we set its value to 0.001[4]. We note that we did not conduct any fine-tuning of the `Models` and auxiliary classifiers hyper-parameters.

Since the evaluated subjects have different feature vector sizes and leverage different base classifier algorithms, we had to resolve some issues faced during our experiments which are related to:

3.5.1 The Size of the Input Vectors. We present in the first column of Table 2 the size of the feature vectors in the difficult datasets of our evaluation subjects. As we can see, DREBIN and RevealDroid leverage huge input vectors that would need massive memory resources to conduct the training. To solve this issue, we rely on feature selection methods to select the top best 200 000 features for both DREBIN and RevealDroid. Although the performance might decrease when discarding the other features, this method can guarantee that the training is feasible.

3.5.2 The Probabilities of Prediction. As we have mentioned in Section 2.2, our method requires a base classifier that outputs prediction probabilities. This requirement is satisfied for MaMaDroid since the base classifier is Random Forest.

For DREBIN and RevealDroid, they train an SVM algorithm that outputs a decision function (i.e., its absolute value indicates the distance of the sample to the hyper-plan that separates the two classes). This function that we denote f can take negative and positive values, and it is unbounded (i.e., it can take any value). In our experiments, we apply a transformation on the decision function to obtain prediction probabilities:

$$p_i = \frac{f_i - f_{\min}}{f_{\max} - f_{\min}} \times (p_{\max} - p_{\min}) + p_{\min}$$

where f_i, $f_{\min}$, $f_{\max}$, $p_{\min}$, and $p_{\max}$ refer to the decision function value of sample i, the minimum and maximum values of f and the minimum and maximum values of p respectively. This transformation converts the positive values of the decision function into probabilities that are equal or greater than 0.5 and the negative values to probabilities smaller than 0.5.

As for MalScan variants, they rely on the 1-Nearest Neighbour classifier that outputs either 0 or 1, which does not enable to exploit probability distributions for identifying thresholds to separate easy and difficult samples. Thus, we propose a work-around, where we train a Random Forest model based on MalScan features, for predicting MalScan outputs. Such a model yields probability distributions for the test set. We leverage this output to identifying the sought threshold for splitting the dataset.

4 EVALUATION RESULTS

4.1 RQ1: To What Extent Is It Feasible to Split a Dataset into Two Subsets, One with Fewer Prediction Errors and One with Most Errors?

In this section, we investigate the possibility of identifying the difficult and easy subsets within a dataset. As introduced in Section 2.3, we hypothesise that (1) most of the samples in the easy subset would be correctly classified by the base classifier (i.e., the easy dataset would yield only a few prediction errors). In contrast, (2) the difficult subset would be associated with most of the prediction errors that are yielded when applying the base classifier to the entire dataset.

We conduct our experiments on the evaluation subjects introduced in Section 3.2. For MaMaDroid variants, we directly apply our method described in Section 2.3 since the base classifiers output prediction probabilities. For DREBIN and RevealDroid, we use the technique described in Section 3.5.2 to map the decision function values returned by the base classifiers (i.e., linear SVM) to prediction probabilities. As for MalScan variants, the 1-NN base classifier does not output usable prediction probabilities (i.e., the probabilities are either 0 or 1). We thus rely on the method described in Section 3.5.2 for splitting the datasets. We note that most classifiers described in scikit-learn documentation [38] generate prediction probabilities or decision function values. Consequently, when the base classifier does not directly output prediction probabilities, our approach is still feasible using the techniques described in Section 3.5.2.

We report in Table 2 the size of the easy and difficult subsets as well as the prediction errors made by the base classifier in each subset. Overall, we are able to split the test dataset into easy and difficult subsets for all the evaluation subjects. Indeed, the easy

[3] We apply the early stopping constraint after the models start to converge
[4] In our preliminary experiments, we tested with three values: 0.05, 0.01 and 0.001. The best results were reported using the value of 0.001

subsets contain few FPs and FNs made by the base classifiers. As for the difficult subsets, they include most of the misclassified samples yielded by the base classifiers on the whole test dataset.

We also present in Figure 3 the evolution of the accumulated FPs and FNs against the prediction probability thresholds on the test dataset. The graphs that are defined for prediction probabilities smaller than 0.5 represent the accumulated FNs. Similarly, the accumulated FPs are represented by the graphs that are defined for prediction probabilities greater than 0.5.

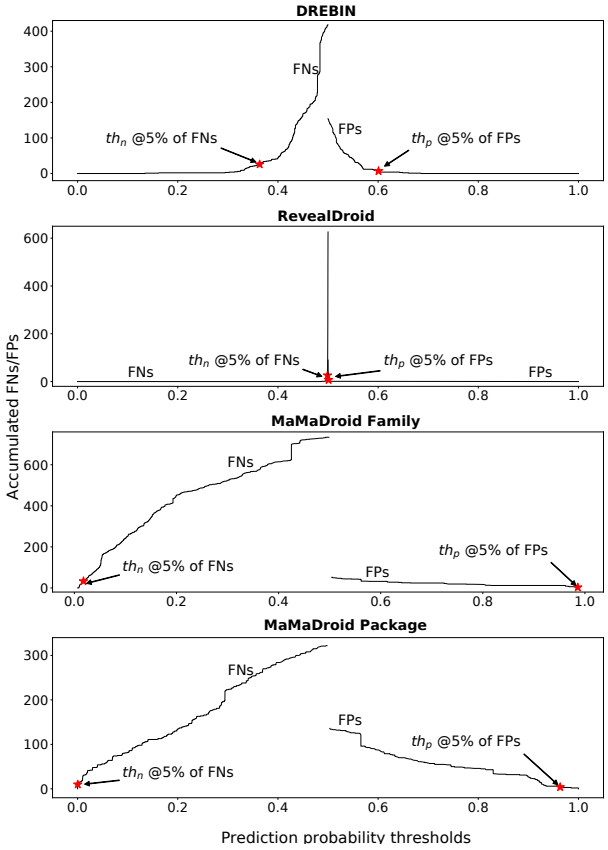

Figure 3: The accumulated number of FPs and FNs as a function of the prediction probability thresholds

From Figure 3, we observe that the accumulated FNs are positively correlated with the prediction probabilities. As for the accumulated FPs, they are negatively correlated with the prediction probabilities. These two observations support our splitting method since we select the easy samples from the two ends of the graphs, where the FNs and FPs are low.

RQ1 answer: Splitting the dataset based on the prediction probabilities indeed leads to two subsets that can be qualified as easy and difficult: the former subset indeed contains samples that a base classifier is effective in predicting (i.e., fewer errors), while the latter subset contains the samples associated to most misclassifications.

4.2 RQ2: How Effective Is Guided Retraining in Improving the Classification Results of State-of-the-Art Malware Detectors?

As we have seen in the previous section, we have created easy and difficult subsets based on the prediction probabilities of the base classifiers. We can directly predict the class of the easy samples using the base classifiers since they make few classification mistakes on these samples. For the difficult subsets, the prediction errors are important.

In this section, we investigate the impact of Guided Retraining on the detection performance of the base classifiers on the difficult samples. To that end, we compare the following classifiers:

- BC, which refers to the original base classifiers that are trained on the whole training dataset;
- RBC: it refers to the original algorithms of the approaches retrained only on the difficult subset for training;
- RClassic, which refers to the use of Contrastive learning (without guiding). We retrain only one `Model` on a training dataset (i.e., either the difficult subset or the whole training samples) to generate the embeddings. Then, we directly train the auxiliary classifier. This method consists of a trivial retraining that does not involve any guidance to generate the embeddings. We present an illustration of this method in Figure 4;
- Guided Retraining, which refers to our approach (Contrastive learning + guiding as described in Section 2). We train the classifiers on the difficult subset for training and evaluate it on the difficult test samples.

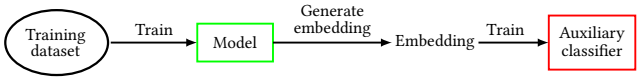

Figure 4: The classic retraining method: RClassic

We define $\Delta Errors$ as the difference between the number of prediction errors made by the original base classifier and the number of prediction errors from the evaluated classifier. Its value can be positive or negative. If it is positive, $\Delta Errors$ means that the evaluated classifier has made more prediction errors than the base classifier. If $\Delta Errors$ is negative, the evaluated classifier has improved the detection performance by decreasing the number of misclassifications reported by the base classifier. Its formula is as follows:

$$\Delta Errors = (FP_{\text{bc}} + FN_{\text{bc}}) - (FP_{\text{ec}} + FN_{\text{ec}})$$

where FP_{bc}, FN_{bc}, FP_{ec}, and FN_{ec} refer to the FPs and FNs of the base classifier (i.e., bc) and the evaluated classifier (i.e., ec) respectively.

In the following, we compare the detection performance of Guided Retraining to BC, RBC, and RClassic on the difficult test subset in Section 4.2.1. We also assess the overall gain in performance by comparing Guided Retraining to BC, and RClassic on the whole test dataset in Section 4.2.2. We note that BC and RBC refer to the same classifier when they are evaluated on the whole test dataset.

Table 3: Comparison of the detection performance of BC, RBC, RCLASSIC, and GUIDED RETRAINING, on the difficult test dataset and the whole test dataset

	Difficult test dataset											Whole test dataset							
	BC		RBC			RCLASSIC trained on the difficult training subset			GUIDED RETRAINING			BC		RCLASSIC trained on the whole training dataset			GUIDED RETRAINING		
	F1 (%)	# Errors (FPs + FNs)	F1 (%)	ΔErr	Errors reduction	F1 (%)	ΔErr	Errors reduction	F1 (%)	ΔErr	Errors reduction	F1 (%)	# Errors (FPs + FNs)	F1 (%)	ΔErr	Errors reduction	F1 (%)	ΔErr	Errors reduction
DREBIN	92.44	540	92.45	-1	0.18%	93.33	-63	11.67%	**93.44**	-69	12.78%	96.28	573	96.52	-36	6.28%	**96.74**	-69	12.04%
Reveal	85.19	676	85.03	+9	-1.33%	87.18	-72	10.65%	**91.44**	-248	36.69%	95.28	715	95.89	-87	12.17%	**97.00**	-248	34.69%
MaMaF	94.46	750	94.32	+18	-2.4%	96.31	-232	30.93%	**96.55**	-264	35.2%	94.76	786	96.36	-221	28.12%	**96.64**	-264	33.59%
MaMaP	97.07	444	94.84	+356	-80.18%	96.9	+25	-5.63%	**97.13**	-9	2.03%	97.04	458	96.98	+12	-2.62%	**97.11**	-9	1.97%
MalscanA	86.02	598	85.38	+32	-5.35%	90.81	-194	32.44%	**91.83**	-243	40.64%	95.72	663	97.28	-241	36.35%	**97.30**	-243	36.65%
MalscanCO	87.25	538	86.51	-37	6.88%	88.43	-58	10.78%	**91.66**	-177	32.90%	96.11	601	95.45	+93	-15.47%	**97.28**	-177	29.45%

4.2.1 ***RQ2-A****: How Effective Is* GUIDED RETRAINING *in Improving the Classification on the Difficult Test Subset?* We calculate the F1-score, $\Delta Errors$, and the percentage of errors reduction for BC, RBC, RCLASSIC and GUIDED RETRAINING on the difficult test samples and we present them in the left part of Table 3 (i.e., Difficult test dataset column).

We observe that RBC has degraded the detection performance for four out of six approaches. For RCLASSIC, it has improved the detection scores for five approaches. However, it has increased the prediction errors of MaMaP by 5.63%. As for GUIDED RETRAINING, it has improved the detection performance of all the approaches. Compared to RCLASSIC, the error reduction of GUIDED RETRAINING is more important and reaches 40.64% for MALSCAN AVERAGE.

We have also repeated our experiments 5-times to verify the generalisability of our results. Before each run of the experiments, we randomly shuffle and split our dataset into training, validation, and test. We present in the left part of Table 4 the average of F1 scores and errors reduction over the five runs of the experiments.

Table 4: Comparison of the detection performance of BC, RBC, RCLASSIC, and GUIDED RETRAINING using 5-times hold-out evaluation

	Difficult test datasets							Whole test datasets				
	BC	RBC		RCLASSIC trained on the difficult subset		GUIDED RETRAINING		BC	RCLASSIC trained on the whole subset		GUIDED RETRAINING	
	F1 (%)	F1 (%)	Errors reduction	F1 (%)	Errors reduction	F1 (%)	Errors reduction	F1 (%)	F1 (%)	Errors reduction	F1 (%)	Errors reduction
DRE	93.36	93.35	-0.08%	94.11	11.21%	**94.18**	12.77%	96.42	96.70	8.06%	**96.85**	12.10%
Rev	86.38	86.34	-0.67%	88.37	9.45%	**91.31**	28.74%	95.36	95.87	10.47%	**96.69**	27.10%
MaMF	94.64	94.54	-1.62%	96.33	28.99%	**96.50**	32.06%	94.88	96.34	25.95%	**96.56**	30.33%
MaMP	86.77	82.49	-18.2%	89.13	14.37%	**89.84**	15.58%	96.29	96.94	13.34%	**96.99**	14.91%
MalA	86.69	86.24	-3.73%	92.48	42.82%	**92.81**	45.19%	95.47	97.28	40.03%	**97.34**	41.06%
MalC	87.81	86.54	-13.02%	89.36	12.94%	**92.50**	36.71%	95.84	95.55	-4.86%	**97.25**	33.09%

Overall, our previous observations are confirmed by the results reported in Table 4. GUIDED RETRAINING outperforms BC, RBC and RCLASSIC classifiers and decreases up to 45.19% of the prediction errors made by the state-of-the art malware detectors on the difficult test subsets.

4.2.2 ***RQ2-B****: How Effective Is* GUIDED RETRAINING *in Improving the Classification on the Whole Test Subset?* To assess the detection performance on the whole test dataset, we leverage GUIDED RETRAINING and the original base classifiers to classify the difficult and easy test subsets respectively. For the experimental comparison, we rely on the base classifiers (i.e., BC) evaluated on the whole test dataset, and RCLASSIC that is trained on the whole training dataset. We remind that training RBC on the whole training dataset is equivalent to BC training. Consequently, we compare GUIDED RETRAINING only to BC and RCLASSIC on the whole dataset and we report our results in the right part of Table 3.

We observe that GUIDED RETRAINING has outperformed RCLASSIC and reduced up to 36.65% of the prediction errors made by the base classifiers.

We also present the results of the five runs of the experiments in the right part of Table 4. The detection scores reported in Table 4 show that GUIDED RETRAINING decreases up to 41.06% of the prediction errors and outperforms both BC and RCLASSIC on the whole test dataset.

> **RQ2 answer:** GUIDED RETRAINING boosts the detection performance of the base classifiers. Indeed, it has reduced the prediction errors made by the base classifiers by up to 45.19% on the difficult test dataset. Furthermore, GUIDED RETRAINING results in higher detection performance than RBC and RCLASSIC classifiers.

4.3 RQ3: How Effective Is GUIDED RETRAINING in Improving the Classification Performance on New Android Apps?

In this section, we evaluate the detection performance of GUIDED RETRAINING in a temporally-consistent scenario. Specifically, we train the base classifiers on apps that are temporally anterior to the apps in the test set. Since this setting has been reported to be challenging for Android malware detectors [3, 34], we assess the added value of GUIDED RETRAINING in enhancing their detection performance. We rely on the experimental setup presented in section 3.5[5], and we evaluate GUIDED RETRAINING against the base classifiers on the whole test dataset. We remind that the easy samples are predicted by the base classifiers, and GUIDED RETRAINING is only used on the difficult subset. We report our results in columns 1 and 2 of Table 5.

We observe that GUIDED RETRAINING with the hyperparameters introduced in Section 3.5 (i.e., batch size = (size_data/10) and early stopping = True) improves the detection performance only for three approaches. During the training that generates the embeddings, we observed that many `Models` do not train for enough epochs due to the early stopping constraint. Consequently, we investigated

[5]We note that we increased the number of epochs for MaMaP auxiliary classifier to 3000 because the difficult dataset was imbalanced

Table 5: Evaluation of the performance of Guided Retraining in enhancing the detection of new Android malware

	Base classifiers		Guided Retraining batch size = (size_data/10) early stopping = True			Guided Retraining batch size = (size_data/10) early stopping = False			Guided Retraining batch size = (size_data/20) early stopping = False		
	F1 (%)	# Errors (FPs + FNs)	F1 (%)	ΔErr	Errors reduction	F1 (%)	ΔErr	Errors reduction	F1 (%)	ΔErr	Errors reduction
DRE	85.31	2032	89.03	-479	23.57%	88.83	-450	22.15%	**89.58**	-547	26.92%
Rev	89.37	1519	88.80	70	-4.61%	88.64	103	-6.78%	**91.42**	-230	15.14%
MaMF	**92.72**	1061	91.84	153	-14.42%	91.94	132	-12.44%	92.19	99	-9.33%
MaMP	91.94	1213	93.30	-194	18.22%	93.33	-224	18.47%	**93.39**	-233	19.21%
MalA	92.77	1090	93.59	-134	12.29%	93.38	-103	9.45%	**93.74**	-158	14.50%
MalCO	92.67	1104	92.40	49	-4.44%	92.37	53	-4.80%	**93.89**	-189	17.12%

two additional settings to help Guided Retraining learn better representations: (1) We removed the early stopping constraint, (2) We removed the early stopping constraint and decreased the batch size to the size of the dataset divided by 20. We present the results of these two settings in columns 3 and 4 of Table 5.

We observe that only removing the early stopping constraint does not improve the detection performance. However, decreasing the batch size and removing the early stopping constraint (i.e., column 4 of Table 5) results in better classifiers. Guided Retraining has reduced the detection errors made by state-of-the-art approaches by up to 26.92%. For MaMaF, the F1 score is still not improved. We further investigated the case of MaMaF by increasing the number of epochs to 4000. This setting has helped the approach to learn better from the dataset and has increased the F1 score to 93.01%. We remind that in this work, we did not fine-tune the hyperparameters of Guided Retraining. We expect the fine-tuning to further reduce the prediction errors of state-of-the-art classifiers.

RQ3 answer: Guided Retraining improves the detection performance of state-of-the-art approaches on **new Android malware** and reduces their prediction errors by up to 26.92%.

4.4 RQ4: What Is the Impact of the Errors Thresholds on the Detection Performance of Guided Retraining?

We now investigate the impact of the parameters X and Y described in Section 2.3 on the detection performance of Guided Retraining. We remind that X and Y represent the percentage of FPs and FNs tolerated in the easy subset, respectively. In the previous experiments, we set their values to 5%, which means we split the datasets to have only 5% of the FPs and FNs in the easy subsets. To assess the impact of these two parameters on the detection performance, we split our datasets into easy and difficult subsets using the following thresholds: 1%, 2%, 5%, 10%, 15%, and 20%. For each of these thresholds, we report Guided Retraining errors reduction on the difficult test datasets in Table 6. We also report the average error reduction of the four classifiers in Table 6. For fair comparison, we do not consider the specific cases of MalScan classifiers since their probability thresholds are inferred from a method that is not fully aligned with the initial publication.

We observe that the impact of X and Y on Guided Retraining detection performance varies depending on the base classifiers. Among the evaluated thresholds, only 2% and 5% have resulted in improving the detection performance of the four approaches. Additionally, the value of 5% has enabled the highest errors reduction and seems to be the best threshold for splitting a dataset into easy and difficult subsets. Moreover, the average error reduction shows that the highest detection performance is achieved when using the threshold of 5%.

Table 6: The impact of FPs and FNs percentage tolerated in the easy dataset on the error reduction* of Guided Retraining

	1%	2%	5%	10%	15%	20%
DREBIN	9.11%	10.09%	12.78%	**16.76%**	-12.06%	-39.30%
Reveal	34.46%	34.14%	36.69%	37.23%	**44.46%**	41.08%
MaMaF	33.46%	32.77%	35.20%	33.24%	**38.02%**	35.78%
MaMaP	-1.55%	0.67%	**2.03%**	-10.31%	-16.29%	-18.25%
Average	18.87%	19.42%	21.67%	19.23%	13.53%	4.83%

* We remind that the highest the error reduction, the best is the detection performance

RQ4 answer: The error thresholds may significantly impact the detection effectiveness of Guided Retraining. Thus, it is important to carefully choose the value of these thresholds to achieve the highest detection performance.

5 RELATED WORK

5.1 The Concept of Difficult Samples

The notion of difficult or hard samples has been discussed in several previous works. Researchers have attributed different definitions to this concept depending on its use case. A study [42] has defined the difficult samples in the context of data imbalance as the samples that belong to the minority class and overlap with the majority class in the embedding space. Its authors have proposed a framework MISO that creates non-overlapping embeddings for the difficult samples based on anchor instances. ADASYN [18] is an algorithm that helps learning from imbalanced datasets by focusing more on the difficult samples during synthetic data generation. Specifically, ADASYN relies on a weighted distribution of the minority classes to generate the synthetic samples. Adaboost [14] is an ensemble learning technique that combines the predictions of a series of base learners. The basic idea of this technique is that each algorithm in the series increases the weights associated with the hard samples (i.e., samples that are incorrectly predicted) reported by the previous learner. Focal Loss [26] and Dice Loss [24] have been proposed to modify the weights associated with the hard/difficult samples. The notion of difficult samples has also been implicitly used in GANs [17] where the generator is trained to produce adversary samples that are difficult to classify by the discriminator.

Our work differs from these related works by considering difficult samples as the instances on which a base classifier is not confident about their predictions.

5.2 Retraining ML Models

Retraining is a technique that generally aims to improve the detection performance of the model. It has been defined and adopted in various ways in the literature. DeltaGrad [48] is proposed to

retrain a model by updating its parameters after adding or deleting a set of training instances. A Neural Network Tree algorithm [53] has been proposed, which relies on a retraining technique that updates the weights of the neural networks to minimise the prediction errors. Similarly, retraining using predicted prior time series data has been proposed to improve the prediction of Anaerobic digestion [33]. SURE [13] is a partial label learning technique that is based on self-training. It introduces the maximum infinity norm regularisation to generate pseudo-labels for the training samples. Weighted Retraining [44] is a method that updates the latent space with new instances and periodically retrains generative models (e.g., GANs [17]) to improve the optimisation. Model retraining techniques have also been proposed for medical research [5, 6] and IoT systems [40].

To tackle the problem of dataset imbalance, a resampling approach has been proposed to obtain a smaller representative subset of the negative samples [25]. This method is inspired by hard-negative mining [12, 16] to select the negative instances that will be considered during the training. Specifically, the resampling technique learns adversarial weights for the negative samples that are then leveraged to determine the size of the negative subset. Finally, a classifier is trained using both the positive samples and the selected negative samples.

Since the easy samples are effectively predicted by the base classifier, our Guided Retraining method targets the difficult samples in order to improve their classification and is guided using the predictions of a base classifier.

5.3 Android Malware Detection

The literature on Android malware lavishes with diverse approaches that aim to detect malicious applications. In addition to the state-of-the-art approaches that we have presented in Section 3.2, many ML-based malware detectors [1, 23, 27, 28, 30, 32, 39, 52] that rely on hand crafted features have been proposed. Recently, image-based Android malware detection has also become popular due to its automatic features extraction [9, 10, 21, 41]. With our Guided Retraining method, we aim to enhance the detection performance of Android malware detectors and reduce their misclassifications.

5.4 Supervised Contrastive Learning for Malware Detection

Recently, a few studies for malware detection have leveraged Supervised Contrastive Learning due to its promising results. IFDroid [49] is an Android malware family classification approach that relies on Supervised Contrastive Learning by considering the instances that belong to the same family as positive samples. Malfustection [29] is a malware classifier and Obfuscation detector that is based on semi-supervised contrastive learning. CADE [51] is a concept drift detection method that relies on Supervised Contrastive Learning to map input samples into a low-dimensional space. In our work, we leverage Contrastive Learning to generate the embeddings of the difficult samples. This process is guided using the predictions of the base classifier.

6 CONCLUSION

To evade detection, attackers devote time and effort to develop malicious software that resemble legitimate programs. Consequently, many malware are difficult to distinguish from genuine programs, and thus manage to make their way into application markets. Real-world software datasets are not perfectly separable into benign and malware samples due to the presence of malicious programs that are very similar to legitimate software and vice versa. These samples are challenging to malware detectors and require sophisticated techniques to achieve a high detection effectiveness. In this paper, we proposed to split a binary dataset into subsets containing either easy or difficult samples. The easy samples are efficiently predicted by a base classifier. For the difficult samples, we propose a more advanced technique to better differentiate the two classes (malicious vs benign). Specifically, we leverage Supervised Contrastive Learning to generate enhanced embeddings for the difficult samples. We rely on the predictions of the base classifier on the difficult samples to guide the retraining that generates the new representations. Then, we train an auxiliary classifier on the new embeddings of the difficult samples. We evaluate our method on four state-of-the-art Android malware detectors, and we show that Guided Retraining boosts the detection performance and reduces the prediction errors by up to 45.19%. We note that our method is not limited to Android malware detection and can be applied to other binary classification tasks.

Data Availability: We make our code and dataset publicly available at https://github.com/Trustworthy-Software/GuidedRetraining

ACKNOWLEDGMENTS

This work was partially supported (a) by the Fonds National de la Recherche (FNR), Luxembourg, under project Reprocess C21/IS/16344458, (b) by the University of Luxembourg under the HitDroid grant, and (c) by the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under project LuxWAyS.

REFERENCES

[1] Fahad Akbar, Mehdi Hussain, Rafia Mumtaz, Qaiser Riaz, Ainuddin Wahid Abdul Wahab, and Ki-Hyun Jung. 2022. Permissions-Based Detection of Android Malware Using Machine Learning. *Symmetry* 14, 4 (2022), 718. https://doi.org/10.3390/sym14040718

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves LeTraon. 2015. Are Your Training Datasets Yet Relevant?. In *Engineering Secure Software and Systems*, Frank Piessens, Juan Caballero, and Nataliia Bielova (Eds.). Springer International Publishing, Cham, 51–67. https://doi.org/10.1007/978-3-319-15618-7_5

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. 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*. https://doi.org/10.14722/ndss.2014.23247

[5] Mariam Barque, Simon Martin, Jérémie Etienne Norbert Vianin, Dominique Genoud, and David Wannier. 2018. Improving wind power prediction with retraining machine learning algorithms. In *2018 International Workshop on Big Data and Information Security (IWBIS)*. 43–48. https://doi.org/10.1109/IWBIS.2018.8471713

[6] Cheng-Yi Chiang, Nai-Fu Chang, Tung-Chien Chen, Hong-Hui Chen, and Liang-Gee Chen. 2011. Seizure prediction based on classification of EEG synchronization patterns with on-line retraining and post-processing scheme. In *2011 Annual*

International Conference of the IEEE Engineering in Medicine and Biology Society. 7564–7569. https://doi.org/10.1109/IEMBS.2011.6091865

[7] Nadia Daoudi, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2021. Lessons Learnt on Reproducibility in Machine Learning Based Android Malware Detection. *Empirical Software Engineering* 26, 4 (2021), 1–53. https://doi.org/10.1007/s10664-021-09955-7

[8] Nadia Daoudi, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2023. Assessing the opportunity of combining state-of-the-art Android malware detectors. *Empirical Software Engineering* 28, 2 (2023), 22. https://doi.org/10.1007/s10664-022-10249-9

[9] Nadia Daoudi, Jordan Samhi, Abdoul Kader Kabore, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2021. DexRay: A Simple, yet Effective Deep Learning Approach to Android Malware Detection Based on Image Representation of Bytecode. In *Deployable Machine Learning for Security Defense*, Gang Wang, Arridhana Ciptadi, and Ali Ahmadzadeh (Eds.). Springer International Publishing, Cham, 81–106. https://doi.org/10.1007/978-3-030-87839-9_4

[10] Yuxin Ding, Xiao Zhang, Jieke Hu, and Wenting Xu. 2020. Android malware detection method based on bytecode image. *Journal of Ambient Intelligence and Humanized Computing* (2020), 1–10. https://doi.org/10.1007/s12652-020-02196-4

[11] Yujie Fan, Mingxuan Ju, Shifu Hou, Yanfang Ye, Wenqiang Wan, Kui Wang, Yinming Mei, and Qi Xiong. 2021. Heterogeneous Temporal Graph Transformer: An Intelligent System for Evolving Android Malware Detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 2831–2839. https://doi.org/10.1145/3447548.3467168

[12] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. 2009. Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence* 32, 9 (2009), 1627–1645. https://doi.org/10.1109/TPAMI.2009.167

[13] Lei Feng and Bo An. 2019. Partial Label Learning with Self-Guided Retraining. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 3542–3549. https://doi.org/10.1609/aaai.v33i01.33013542

[14] Yoav Freund and Robert E Schapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. Comput. System Sci.* 55, 1 (1997), 119–139. https://doi.org/10.1006/jcss.1997.1504

[15] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Trans. Softw. Eng. Methodol.* 26, 3, Article 11 (Jan. 2018), 29 pages. https://doi.org/10.1145/3162625

[16] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587. https://doi.org/10.1109/CVPR.2014.81

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[18] Haibo He, Yang Bai, Edwardo A. Garcia, and Shutao Li. 2008. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. 1322–1328. https://doi.org/10.1109/IJCNN.2008.4633969

[19] Shifu Hou, Yujie Fan, Yiming Zhang, Yanfang Ye, Jingwei Lei, Wenqiang Wan, Jiabin Wang, Qi Xiong, and Fudong Shao. 2019. α Cyber: Enhancing Robustness of Android Malware Detection System against Adversarial Attacks on Heterogeneous Graph Based Model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 609–618. https://doi.org/10.1145/3357384.3357875

[20] Ming Huang, Fuzhen Zhuang, Xiao Zhang, Xiang Ao, Zhengyu Niu, Min-Ling Zhang, and Qing He. 2019. Supervised representation learning for multi-label classification. *Machine Learning* 108, 5 (2019), 747–763. https://doi.org/10.1007/s10994-019-05783-5

[21] T. H. Huang and H. Kao. 2018. R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections. In *2018 IEEE International Conference on Big Data (Big Data)*. 2633–2642. https://doi.org/10.1109/BigData.2018.8622324

[22] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised Contrastive Learning. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 18661–18673. https://proceedings.neurips.cc/paper/2020/file/d89a66c7c80a29b1bdbab0f2a1a94af8-Paper.pdf

[23] Vasileios Kouliaridis and Georgios Kambourakis. 2021. A Comprehensive Survey on Machine Learning Techniques for Android Malware Detection. *Information* 12, 5 (2021). https://doi.org/10.3390/info12050185

[24] Xiaoya Li, Xiaofei Sun, Yuxian Meng, Junjun Liang, Fei Wu, and Jiwei Li. 2020. Dice Loss for Data-imbalanced NLP Tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 465–476. https://doi.org/10.18653/v1/2020.acl-main.45

[25] Yi Li and Nuno Vasconcelos. 2020. Background data resampling for outlier-aware classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 13218–13227. https://doi.org/10.1109/CVPR42600.2020.01323

[26] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. 2017. Focal Loss for Dense Object Detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/ICCV.2017.324

[27] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu. 2020. A Review of Android Malware Detection Approaches Based on Machine Learning. *IEEE Access* 8 (2020), 124579–124607. https://doi.org/10.1109/ACCESS.2020.3006143

[28] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep Learning for Android Malware Defenses: A Systematic Literature Review. *ACM Comput. Surv.* 55, 8, Article 153 (dec 2022), 36 pages. https://doi.org/10.1145/3544968

[29] Mohammad Mahdi Maghouli, Mohamadreza Fereydooni, Monireh Abdoos, and Mojtaba Vahidi-Asl. 2021. Malfustection: Obfuscated Malware Detection and Malware Classification with Data Shortage by Combining Semi-Supervised and Contrastive Learning. *arXiv preprint arXiv:2111.09975* (2021). https://doi.org/10.48550/arXiv.2111.09975

[30] Arvind Mahindru and AL Sangal. 2021. MLDroid—framework for Android malware detection using machine learning techniques. *Neural Computing and Applications* 33, 10 (2021), 5183–5240. https://doi.org/10.1007/s00521-020-05309-4

[31] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *ISOC Network and Distributed Systems Security Symposiym (NDSS)*. San Diego, CA. https://doi.org/10.14722/ndss.2017.23353

[32] Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, and Paul Miller. 2021. Multi-view deep learning for zero-day Android malware detection. *Journal of Information Security and Applications* 58 (2021), 102718. https://doi.org/10.1016/j.jisa.2020.102718

[33] Jun-Gyu Park, Hang-Bae Jun, and Tae-Young Heo. 2021. Retraining prior state performances of anaerobic digestion improves prediction accuracy of methane yield in various machine learning models. *Applied Energy* 298 (2021), 117250. https://doi.org/10.1016/j.apenergy.2021.117250

[34] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 729–746. https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury

[35] PyTorch. . https://pytorch.org. [Online; accessed 30-August-2022].

[36] Alain Rakotomamonjy. 2017. Supervised Representation Learning for Audio Scene Classification. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25, 6 (2017), 1253–1265. https://doi.org/10.1109/TASLP.2017.2690561

[37] scikit learn. . https://scikit-learn.org. [Online; accessed 30-August-2022].

[38] scikit learn. . https://scikit-learn.org/stable/modules/classes.html. [Online; accessed 30-August-2022].

[39] Tejpal Sharma and Dhavleesh Rattan. 2021. Malicious application detection in android — A systematic literature review. *Computer Science Review* 40 (2021), 100373. https://doi.org/10.1016/j.cosrev.2021.100373

[40] Yan Song, Yibin Li, Lei Jia, and Meikang Qiu. 2020. Retraining Strategy-Based Domain Adaption Network for Intelligent Fault Diagnosis. *IEEE Transactions on Industrial Informatics* 16, 9 (2020), 6163–6171. https://doi.org/10.1109/TII.2019.2950667

[41] Tiezhu Sun, Nadia Daoudi, Kevin Allix, and Tegawendé F. Bissyandé. 2021. Android Malware Detection: Looking beyond Dalvik Bytecode. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering Workshops* (Virtual Event, Australia) *(ASE '21)*. https://doi.org/10.1109/ASEW52652.2021.00019

[42] Jiachen Tian, Shizhan Chen, Xiaowang Zhang, Zhiyong Feng, Deyi Xiong, Shaojuan Wu, and Chunliu Dou. 2021. Re-embedding Difficult Samples via Mutual Information Constrained Semantically Oversampling for Imbalanced Text Classification. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 3148–3161. https://doi.org/10.18653/v1/2021.emnlp-main.252

[43] Yonglong Tian. 2020. https://github.com/HobbitLong/SupContrast. [Online; accessed 30-August-2022].

[44] Austin Tripp, Erik Daxberger, and José Miguel Hernández-Lobato. 2020. Sample-Efficient Optimization in the Latent Space of Deep Generative Models via Weighted Retraining. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 11259–11272. https://proceedings.neurips.cc/paper/2020/file/81e3225c6ad49623167a4309eb4b2e75-Paper.pdf

[45] VirusTotal. . https://www.virustotal.com. [Online; accessed 30-August-2022].

[46] Mike Walmsley, Anna MM Scaife, Chris Lintott, Michelle Lochner, Verlon Etsebeth, Tobias Géron, Hugh Dickinson, Lucy Fortson, Sandor Kruk, Karen L

Masters, et al. 2021. Practical Galaxy Morphology Tools from Deep Supervised Representation Learning. *arXiv preprint arXiv:2110.12735* (2021). https://doi.org/10.1093/mnras/stac525

[47] Xiaohui Wan, Zheng Zheng, Fangyun Qin, Yu Qiao, and Kishor S. Trivedi. 2019. Supervised Representation Learning Approach for Cross-Project Aging-Related Bug Prediction. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 163–172. https://doi.org/10.1109/ISSRE.2019.00025

[48] Yinjun Wu, Edgar Dobriban, and Susan Davidson. 2020. DeltaGrad: Rapid retraining of machine learning models. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 10355–10366. https://proceedings.mlr.press/v119/wu20b.html

[49] Yueming Wu, Shihan Dou, Deqing Zou, Wei Yang, Weizhong Qiang, and Hai Jin. 2021. Obfuscation-resilient Android Malware Analysis Based on Contrastive Learning. *arXiv preprint arXiv:2107.03799* (2021). https://doi.org/10.48550/arXiv.2107.03799

[50] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin. 2019. MalScan: Fast Market-Wide Mobile Malware Scanning by Social-Network Centrality Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 139–150. https://doi.org/10.1109/ASE.2019.00023

[51] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. 2021. CADE: Detecting and Explaining Concept Drift Samples for Security Applications. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2327–2344. https://www.usenix.org/conference/usenixsecurity21/presentation/yang-limin

[52] Nan Zhang, Yu an Tan, Chen Yang, and Yuanzhang Li. 2021. Deep learning feature exploration for Android malware detection. *Applied Soft Computing* 102 (2021), 107069. https://doi.org/10.1016/j.asoc.2020.107069

[53] Q. Zhao. 2001. Training and retraining of neural network trees. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, Vol. 1. 726–731 vol.1. https://doi.org/10.1109/IJCNN.2001.939114

[54] Rui Zhu, Chenglin Li, Di Niu, Hongwen Zhang, and Husam Kinawi. 2018. Android malware detection using large-scale network representation learning. *arXiv preprint arXiv:1806.04847* (2018). https://doi.org/10.48550/arXiv.1806.04847

Received 2023-02-16; accepted 2023-05-03