PetaDroid: Adaptive Android Malware Detection Using Deep Learning

ElMouatez Billah Karbab(✉) and Mourad Debbabi

Concordia Security Research Center, Montreal, Canada
{elmouatez.karbab,mourad.debbabi}@concordia.ca

Abstract. Android malware detection is a significant problem that affects billions of users using millions of Android applications (apps) in existing markets. Thiss paper proposes PetaDroid, a framework for accurate Android malware detection and family clustering on top of static analyses. PetaDroid automatically adapts to Android malware and benign changes over time with resilience to common binary obfuscation techniques. The framework employs novel techniques elaborated on top of natural language processing (NLP) and machine learning techniques to achieve accurate, adaptive, and resilient Android malware detection and family clustering. We extensively evaluated PetaDroid on multiple reference datasets. PetaDroid achieved a high detection rate (98–99% f1-score) under different evaluation settings with high homogeneity in the produced clusters (96%). We conducted a thorough quantitative comparison with state-of-the-art solutions MaMaDroid, DroidAPIMiner, MalDozer, in which PetaDroid outperforms them under all the evaluation settings.

1 Introduction

Android OS's popularity has increased tremendously since the last decade. It is undoubtedly an appropriate choice for smart mobile devices such as phones and tablets or the internet of things devices such as TVs due to its open-source license and the massive number of useful apps developed for this platform (about 4 Million apps in 2019 [2]). Nevertheless, malicious apps target billions of Android users through centralized app markets. The detected malicious apps increased by 40% in 2018-Q3 compared to the same period in 2017 [1]. Google Play employs a vetting system named `Bouncer` to detect malicious apps through static and dynamic analyses. Despite these analyses, many malicious apps[1] were able to bypass `Bouncer` and infect several hundred thousand devices[2]. Therefore, there is a dire need for accurate, adaptive, yet resilient Android malware detection systems for the app market scale.

1.1 Problem Statement

In this paper, we identify the following gaps in the state-of-the-art solutions for Android malware detection:

[1] https://tinyurl.com/y4qdtuy9.
[2] https://tinyurl.com/y4mckwxm.

L. Bilge et al. (Eds.): DIMVA 2021, LNCS 12756, pp. 319–340, 2021.
https://doi.org/10.1007/978-3-030-80825-9_16

P1: The accuracy of Android malware detection systems tends to decrease over time due to different factors: (1) variations in existing malware family, (2) new malware families, (3) and new Android APIs in benign and malicious apps. These factors are mostly reflected in the changes in Android API call sequences in malicious and benign apps. Nevertheless, these changes are incremental in most cases compared to the existing apps. In this context, we consider two problems: (1) The resiliency of the detection systems that use machine learning models [31] to changes over time, (2) and the possibility of automatic adaptation to the new changes [40].

P2: Android malware family attribution is an important problem in the realm of malware detection. The malware family attribution could be important essential to define the threats[3] of the detected malware [28]. However, few existing solutions [7] provide Android malware family attribution. Furthermore, these solutions rely on supervised learning where prior knowledge of the families is required [12]. However, such knowledge is hard to get and not realistic in many cases, especially for new malware families[4].

P3: Malware developers employ various obfuscation techniques to thwart detection attempts. Obfuscation resiliency is a key requirement in modern malware fingerprinting that applies static analyses. Few solutions address the obfuscation issue [36,40] in the context of Android malware detection, more specifically, the resiliency to common obfuscations and binary code transformations.

1.2 Proposed Solution

In this paper, we propose PetaDroid, an accurate, adaptive, resilient, and yet efficient Android malware detection and family clustering using natural language processing (NLP) and deep learning techniques on top of static analysis features. In PetaDroid, we aim to address the previously mentioned problems as follows:

1. Our fundamental intuition for time resiliency and adaptation is that Android apps are changing over time incrementally. Benign apps embrace new Android APIs, deprecations, and components gracefully to do not disturb the user experience. Malware developers aim to target the maximum devices by employing stable and cross-Android version APIs. We argue that PetaDroid can fingerprint malicious apps within a time window with high confidence because the application still contains enough patterns of similarity to known samples.

2. PetaDroid goes a step further in the detection process by clustering the detected samples into groups with high similarity. We *exclusively* group highly similar samples, most likely of the same malware family. PetaDroid family attribution is found upon the assumption that malicious applications tend to have similar characteristics in the Android Dalvik bytecode code. We leverage this assumption to build an automatic and unsupervised malware family tagging system using deep neural network auto-encoder for sample digest generation on top of *InstNGram2Bag* features (based on NLP bag of words). Using the DBScan

[3] https://tinyurl.com/yydg5vew.
[4] https://tinyurl.com/y8rc6q89.

[11] clustering algorithm, we cluster the *most similar samples* from the detected malicious apps.

3. PetaDroid introduces code fragments randomization during training and deployment phases to enhance the obfuscation resiliency. We artificially apply random permutations to change the order of code basic-blocks without altering the basic-block instructions. We consider a code basic-block as a possible micro-action in the app execution flows. Therefore, we randomize the app execution flows without affecting the micro-actions within the flow to emulate code transformation during the training and deployment phases. Code fragment randomization strengthens the obfuscation robustness of PetaDroid, as shown in Sect. 4.3.

1.3 Contributions and Outline

The main contributions of this paper are:

(1) We propose a novel adaptation technique for Android malware detection to automatically adapt the detection system. The proposed techniques rely on the confidence probability of the detection ensemble to collect extension training datasets from received samples during the deployment (Sect. 2.2).
(2) We propose a novel fragment randomization technique to boost the detection system resiliency to common code-obfuscation techniques. In this technique, we randomize the order of code basic-blocks without affecting the basic-blocks instructions during the training and the deployment phases (Sect. 2.2).
(3) We propose PetaDroid, an accurate and efficient malware detection and clustering framework based on code static analyses, NLP, and machine learning techniques. In PetaDroid, we propose an ensemble of CNN models on top of a code embedding model, namely *Inst2Vec*, to accurately detect malware with probability confidence (Sect. 2.2). We released the source code of PetaDroid to the community in https://github.com/mouatez/petadroid.
(4) We extensively evaluate PetaDroid to assess its effectiveness and efficiency on different reference datasets of PetaDroid under various evaluation settings (Sect. 2.1).

2 PetaDroid

In this section, we detail PetaDroid methodology and its components.

2.1 Android App Representation

In this section, we present the preprocessing of Dalvik code and its representation into a canonical instructions sequence. We seek the preservation of the maximum information about apps' behaviors while keeping the process very efficient. The preprocessing begins with the disassembly of an app bytecode to Dalvik assembly code, as depicted in Fig. 1.

```
// Object Creation
new-instance v10, java/util/HashMap
// Object Access
invoke-direct v10, java/util/HashMap
if-eqz v9, 003e
..
// Method Invocation
// * = Android/telephony
invoke-virtual v4, */TelephonyManager.getDeviceId()java/lang/String
move-result-object v11
// Method Invocation
invoke-virtual v4, */TelephonyManager.getSimSerialNumber()java/lang/String
move-result-object v13
// Method Invocation
invoke-virtual v4 */TelephonyManager.getLine1Number()java/lang/String
move-result-object v4
...
// Object Creation
new-instance v20, java/io/FileReader
const-string v21, "/proc/cpuinfo"
invoke-direct/range v20, v21, java/io/FileReader.init(java/lang/String)
new-instance v21, java/io/BufferedReader
...
move/from16 v2, v20
// Field Access
// * = Android/content/pm
iget-object v0, v0, */ApplicationInfo.metaData Android/os/Bundle
move-object/from16 v19, v0
```

Fig. 1. Android assembly from a malware sample

We model the Dalvik assembly code as code fragments where each fragment is a class's method code in the Dalvik assembly. It is a natural separation because Dalvik code D is composed of a set of classes $D = \{C_1, C_2, \dots C_s\}$. Each class C_i contains a set of methods $C = \{M_1, M_2, \dots M_k\}$, where we find actual assembly code instructions. We preserve the order of Dalvik assembly instructions within methods while ignoring the global execution paths. Method execution is a possible *micro-behavior* for an Android app, while a global execution path is a likely *macro-behavior*. An Android app might have multiple global execution paths based on external events. In contrast, Android malware tends to have one crucial global execution path (malicious payload) and other ones to distract malware detection systems. The malware could produce variations for the payload global execution path. However, it still depends on the micro-behavior to produce another global one. PetaDroid assembly preprocessing produces a multiset of sequences $P = \{S_1, S_2, \dots S_h\}$ where each sequence S contains an ordered instruction sequence $S = \langle I_1, I_2, \dots I_v \rangle$ of a class's method. In other words, P contains instruction sequences $P = \{\langle I_1, I_2, \dots \rangle_1, \langle I_1, I_2, \dots \rangle_2, \dots \langle I_1, I_2, \dots \rangle_h\}$ where the order is only preserved inside individual sequences S_i (the methods instructions). Thus, a sequence S defines a possible micro-execution (or behavior) from the Android app's overall runtime execution.

As shown in Fig. 1, the Dalvik assembly is too sparse. We want to keep the assembly instruction skeleton that reflects possible runtime behaviors with less sparsity. In PetaDroid, we propose a canonical representation for Dalvik assembly code, as shown in Fig. 2. The key idea is to keep track of the Android platform APIs and objects utilized inside the method assembly. To fingerprint

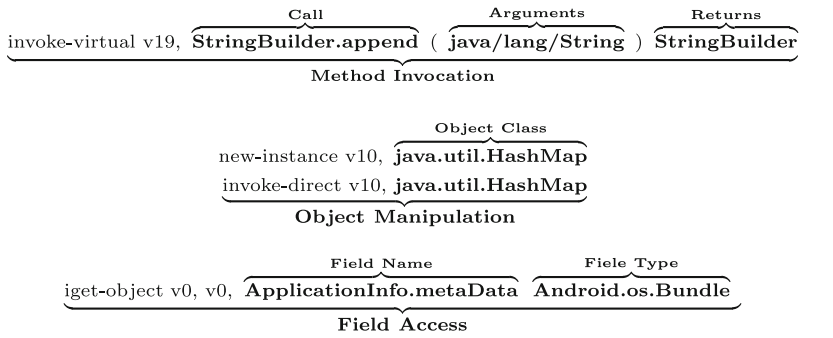

Fig. 2. Canonical representation of Dalvik assembly

malicious apps, the canonical representation will mostly preserve the actions and the manipulated system objects, such as sending SMS action or getting (setting) sensitive information objects. PetaDroid canonical representation covers three types of Dalvik assembly instructions, namely: *Method invocation*, *object manipulation*, and *field access*, as shown in Fig. 2. In the method invocation, we focus on the method call, *Package.ClassName.MethodName*, the parameters list, *Package.ClassName*, and the return type, *Package.ClassName*. In object manipulation, we capture the class object, *Package.ClassName*, that is being used. Finally, we track the access to system fields by capturing the field name, *Package.ClassName.FieldName*, and its type, *Package.ClassName*. Our manual inspections of Dalvik assembly for hundreds of malicious and benign samples shows that these three forms cover the essential of Dalvik assembly instructions.

PetaDroid instruction parser keeps only the canonical representation and ignores the rest. For example, our experiments show that Dalvik opcodes add a lot of sparsity without enhancing the malware fingerprinting performance. On the contrary, it could negatively affect overall performance, which is shown in previous solutions [29]. The final step in preprocessing a method M (see Fig. 1) is to flatten the canonical representation of a method into a single sequence S.

We keep only the Android platform related assets like API, classes, and system fields in the final method's sequence S. We maintain a vocabulary dictionary (key: value) in the form of $(Androidassets : identifier)$ (for example $(Android/telephony/TelephonyManager : 439)$) of all Android OS assets (all versions) to filter and map Android assets to unique identifiers (unique integer for a given Android assets) for the method instruction sequence during the preprocessing. The output of the app representation phase is a list of sequences $\hat{P} = \{S_{c,1}, S_{c,2}, \ldots S_{c,h}\}$. Each sequence is an ordered canonical instruction representation of one method. In the following, we summarize the notations used in the rest of the paper (Table 1):

2.2 Malware Detection

In this section, we present the PetaDroid malware detection process using CNN on top of *Inst2Vec* embedding features. The detection process starts from a

Table 1. Notation summary

Notation	Description	Format
D	Dalvik assembly code of one Android App	Raw text
C	Dalvik Java Class	Raw text
M	Dalvik Java Method	Raw text
S	Sequence of extracted instructions of one Dalvik Java Method M	List of Dalvik raw text instructions
P	Multiset of methods' sequences S	Multiset of sequences
S_c	Sequence of canonical instructions generated from S using V	List of canonical instruction IDs
$\hat{P}$	Multiset of methods' sequences S_c	Multiset of sequences
P_c	The result of shuffling and concatenating of all S_c	Sequence of canonical instructions
F	Fragment is a truncated portion from P_c	List of canonical instructions
$CNNModel$	Classification model based on Convolutional Neural Network (CNN)	Deep learning model
Φ	Ensemble of classification models $\Phi = \{CNNModel_1, CNNModel_2, \dots CNNModel_\phi\}$	Set of deep learning models
y	Dataset label	Malware or not
$\hat{y}$	Prediction likelihood of the classification models $\hat{y} = \Phi(F)$	Probability
ζ	Detection threshold for the general decision strategy	Probability threshold
η	Detection threshold for the confidence decision strategy	Probability threshold

multiset of discretized canonical instruction sequences $\hat{P} = \{S_{c,1}, S_{c,2}, \dots S_{c,h}\}$. Notice that $\hat{P}$ is a multiset and not a set since it might contain duplicated sequences. The duplication comes from having the same Dalvik method's code in two (or more) distinct Dalvik classes. PetaDroid CNN ensemble produces a detection result together with maliciousness and benign detection probabilities for a given sample. To achieve automatic adaptation, we leverage the detection probabilities to automatically collect an extension dataset that PetaDroid employs to build new CNN ensemble models.

$$\hat{P} = \{S_{c,1}, S_{c,2}, S_{c,2}\} \tag{4}$$

$$\{\ \underbrace{S_{c,3}, S_{c,1}, S_{c,2}}_{\textbf{Shuffled sequences}}\ \} \tag{5}$$

$$\{\overbrace{\langle I_1, I_1, \dots I_{|Sc3|}\rangle_3, \langle I_1, I_1, \dots I_{|Sc1|}\rangle_1, \langle I_1, I_1, \dots I_{|Sc2|}\rangle_2,}^{\textbf{while preserving the order inside sequences}}\} \tag{6}$$

$$\hat{P_c} = \{\overbrace{\underbrace{I_{3,1}, I_{3,1}, \dots I_{3,|Sc3|}}_{\textbf{Fragment truncation}}, I_{1,1}, I_{1,1}, \dots I_{1,|Sc1|}, \underbrace{I_{2,1}, I_{2,1}, \dots I_{2,|Sc2|}}_{\textbf{Rest}},}^{\textbf{Sequence concatenation}}\} \tag{7}$$

$$F = \{\overbrace{I_{3,1}, I_{3,1}, \dots I_{3,|Sc3|}, I_{1,1}, I_{1,1}, \dots I_{1,|Sc1|}, I_{2,1}}^{\textbf{on fragment size } |F|}\} \tag{8}$$

Fig. 3. Example of fragment generation

Fragment Detection. Fragment-based detection is a key technique in PetaDroid. A fragment F is a truncated portion from the beginning of the concatenation P_c of $\hat{P} = \{S_{c,1}, S_{c,2}, \ldots S_{c,h}\}$ as shown in Fig. 3. The size $|F|$ is the number of canonical instructions in the fragment F, and it is a hyper-parameter in PetaDroid. Our grid search for the best $|F|$ hyper-parameter result $|F| = 10k$ for the current version of PetaDroid. For a sequence $S_{c,i}$, the order of canonical instructions is preserved within a method. In other words, we guarantee the preservation of order inside the method sequence or what we refer to as a *micro-action*. However, no specific order is assumed between methods' sequences or what we refer to as *macro-action* (or behavior). On the contrary, before we truncate P_c into size $|F|$, we apply random permutations on $\hat{P}$ to produce a random order in the macro-behavior. The randomization happens in every access, whether it is during training or deployment phases. Each Android sample has $\frac{h!}{(h-k)!}$ possible permutations for the methods' sequences $\hat{P} = \{S_{c,1}, S_{c,2}, \ldots S_{c,h}\}$, where h is the number of methods' sequence in a given Android app, and k is the number of sampled sequences. The concatenation of the sampled k sequences must be greater than $|F|$.

Inst2Vec Embedding. *Inst2Vec* is based on *word2vec* [30] technique to produce an embedding vector for each canonical instruction in our sequences. *Inst2Vec* is trained on instruction sequences to learn instruction semantics from the underlying contexts. This means that *Inst2Vec* learns a dense representation of a canonical instruction that reflects the instruction co-occurrence and context. The produced embeddings capture the semantics of instructions (interpreted by geometric distances). Furthermore, embedding features show high code fingerprinting accuracy and resiliency to common obfuscation techniques [10].

Classification Model. Our single CNN model takes *Inst2Vec* features, which are a sequence of embeddings; each embedding captures the semantics of an instruction. The temporal CNN [23], or 1-dimensional CNN [42], is the working core component in the PetaDroid single classification model. We choose to build our classification models based on CNN architecture over recurrent neural networks (RNN) such as LSTM or GRU. Due to the efficiency of CNN during the training and the deployment compared to RNN. **In the training phase**, the CNN models take on average 0.05 s per batch (32 samples), which is five times faster than RNN models in our experiments. The CNN model converges early (starting from 10 epochs) compared to the RNN model (starting from 30 epochs). **In the deployment phase**, the CNN model's inference is, on average, five times faster than RNN models. Both neural network architecture gives very similar detection results in our experiments. However, our automatic adaptation technique will benefit from the efficiency of CNN models to rapidly build new models using large datasets. The non-linearity used in our model employ the rectified linear unit (ReLUs) $h(x) = \max\{0, x\}$. We used Adam [13] optimization algorithm with a 32 mini-batch size and a $3e-4$ learning rate for 100 epochs

in all our experiments. The chosen hyper-parameters are the results of empirical evaluations to find the best values.

Detection Ensemble. PetaDroid detection component relies on an ensemble $\Phi = \{CNNModel_1, CNNModel_2, \ldots, CNNModel_\phi\}$. Ensemble Φ is composed of ϕ single CNN models. The number of single CNN models in the ensemble ϕ is a hyper-parameter. We choose to be $\phi = 6$, which is a trade-off of between maximum effectiveness on malware detection with the highest efficiency possible base on our evaluation experiments.

As mentioned previously, PetaDroid trains each CNN model for the number of epochs ($epochs = 100$). In each epoch, we compute $Loss_T$ and $Loss_V$, the *training* and *validation* losses, respectively, and save a snapshot of the single CNN model parameters. $Loss_T$ and $Loss_V$ are the log loss across training and validation sets:

$$p = CNNModel_\theta(y = 1|F)$$

$$loss(y, p) = -(y \log(p) + (1 - y) \log(1 - p)),$$

$$Loss_T = \frac{-1}{m_{train}} \sum_{i=1}^{m_{train}} loss(y_i, p_i),$$

$$Loss_V = \frac{-1}{m_{valid}} \sum_{i=1}^{m_{valid}} loss(y_i, p_i),$$

Where p is the maliciousness likelihood probability given a fragment F (a truncated concatenation of canonical instructions P_c) and model parameters θ (Sect. 2.1). PetaDroid selects the top ϕ models automatically from the saved model snapshots that have the lowest *training* and *validation* losses $Loss_T$ and $Loss_R$, respectively.

$$\hat{y} = \Phi(x) \quad = \frac{1}{\phi} \left(\sum_{i}^{\phi} CNNModel_i(x) \right) \tag{1}$$

PetaDroid CNN ensemble Φ produces a maliciousness probability likelihood by averaging the likelihood probabilities of multiple CNN models, as shown in Eq. 1.

Confidence Analysis. PetaDroid ensemble computes the maliciousness probability likelihood $Prob_{Mal}$ given a fragment F, as follows:

$$\hat{y} = \Phi(F), \quad Prob_{Mal} = \hat{y}, \quad Prob_{Ben} = (1 - \hat{y})$$

Previous Android malware detection solutions, such as [18,31], utilize a simple detection technique (we refer to it as a *general decision*) to decide on the maliciousness of Android apps. In the *general decision*, we compute the general threshold $\zeta \in [0, 1]$ that achieves the highest detection performance on the validation dataset X_{valid}. In the deployment phase (or evaluation in our case

on X_{test}), The general decision D_ζ utilize the computed threshold ζ to make detection decisions:

$$D_\zeta = \begin{cases} Malware & Prob_{Mal} > \zeta \\ Benign & Prob_{Mal} <= \zeta \end{cases}$$

PetaDroid employs f1-score as a detection performance metric to automatically select ζ and to report the general detection performance on the test set X_{test} during our evaluation, in Sect. 4. We choose f1-score as our detection performance metric due to its simplicity, and its measurement reflects the reality under unbalanced datasets. *The general decision* provides a firm decision for every sample. However, security practitioners might prefer dealing with decisions that have associated confidence values and filter out less-confident decisions for further investigations. In a real deployment, we want as many detection decisions with high confidence and filter out the few uncertain apps with low confidence probability. Unfortunately, the *general decision* strategy that has been used by most previous solutions does not provide such functionality. For this purpose, we propose the **confidence decision strategy**, a mechanism to automatically filter out apps with uncertain decisions. PetaDroid computes a confidence threshold η that achieves a high detection performance (f1-score) and a negligible error rate (false negative and false positive rates) in the validation dataset. In other words, we add the error rate constraint to the system that computes the detection threshold η from X_{valid}. In the deployment, we make the confidence-based decision as follow:

$$D_\eta = \begin{cases} Uncertain & Prob_{Mal} < \eta \ \wedge \ Prob_{Ben} < \eta \\ Malware & Prob_{Mal} >= \eta \ \wedge \ Prob_{Mal} > Prob_{Ben} \\ Benign & Prob_{Ben} >= \eta \ \wedge \ Prob_{Ben} > Prob_{Mal} \end{cases}$$

Automatic Adaptation. In this section, we describe our mechanism to adapt to Android ecosystem changes over time automatically. The key idea is to re-train the CNN ensemble on new benign and malware samples periodically to learn the latest changes. To enhance the automatic adaptation, we leverage the confidence analysis to collect an extension dataset that captures the incremental change over time. Initially, we train PetaDroid ensemble using $X_{build} = \{X_{train} + X_{valid}\}$. Afterward, PetaDroid leverages the *confidence detection strategy* to build an extension dataset X_{exten} from test dataset X_{test} from high-confidence detected apps. In a real deployment, X_{test} is a stream of Android apps that needs to be checked for maliciousness by the vetting system. The test dataset $X_{test} = \{X_{Certain}, X_{Uncertain}\}$ is composed of apps having a high-confidence decision ($X_{Certain}$ or X_{exten}) and apps having uncertain decisions $X_{Uncertain}$. In the deployment, PetaDroid accumulates from high-confidence apps over time to form X_{exten} dataset. Periodically, PetaDroid utilizes the extension dataset X_{exten} to extend the original X_{build} and later updates the CNN ensemble models. In our evaluation, and after updating the CNN ensemble, we report

updated general performance and **updated confidence-based performance**, respectively the general and confidence-based performance of the new trained CNN ensemble on X_{test}. These metrics answer the question: what would be the detection performance on $X_{test} = \{X_{Certain}, X_{Uncertain}\}$ after we build the ensemble on $X_{NewBuild} = \{X_{Certain}, X_{build}\}$? In other words, PetaDroid reviews previous detection decisions using the new CNN ensemble and drives new general and confidence-based performance.

2.3 Malware Clustering

In this section, we detail the family clustering system. PetaDroid clustering aims to group the previously detected malicious apps (Sect. 2.2) into highly similar malicious apps groups, which are most likely part of the same malware family. PetaDroid clustering process starts from a multiset of discretized canonical instruction sequences $P = \{S_{c,1}, S_{c,2}, \ldots S_{c,h}\}$ of the detected malicious apps. We introduce the *InstNGram2Vec* technique and deep neural network auto-encoder to generate embedding digests for malicious apps. Afterward, we cluster malware digests using the DBScan [11] clustering algorithm to generate malware family groups. Notice that our clustering system (DBScan [11]) requires to represent malware samples by one feature vector for each sample instead of a list of embeddings as in *Inst2Vec* for PetaDroid classification. For this reason, we introduce *InstNGram2Vec* technique that automatically represents malware samples as feature vectors without an explicit manual feature selection. *InstNGram2Vec* is a technique that maps concatenated instruction sequences to fixed-size embeddings employing NLP bag of words (N-grams) and feature hashing [35] techniques.

Auto-Encoder. We develop a deep neural auto-encoder through stacked neural layers of encoding and decoding operations The proposed auto-encoder learns the latent representation of Android apps in an unsupervised way. The unsupervised learning of the auto-encoder is done through the reconstruction of the unlabeled hashing vectors $HV = \{hv_0, hv_1, \ldots hv_{DMal}\}$ of random Android apps. Notice that we do not need any labeling during the training of PetaDroid auto-encoder, off-the-self Android apps are sufficient.

Family Clustering. PetaDroid clusters the detected malware digests $\boldsymbol{Z} = \{\boldsymbol{z_0}, \boldsymbol{z_1}, \ldots, \boldsymbol{z_{DMal}}\}$ into groups of malware with high similarity and most likely belonging to the same family. In PetaDroid clustering: **First**, we use an **exclusive** clustering mechanism. The clustering algorithm only groups highly similar samples and tags the rest as non-clustered. This feature could be more convenient for real-world deployments since we might not always detect malicious apps from the same family, and we would like to have family groups only if there are groups of the sample malware family. To achieve this feature, we employ the *DBScan* clustering algorithm. **Second**, as an optional step, we find the best cluster for the non-cluster samples, from the clusters produced previously by computing the euclidean similarity between a given non-cluster sample and a

given cluster samples. We call this step the *family matching*. In the evaluation, we report *homogeneity* and *coverage* metric for the clustering before and after applying this optional step. *DBScan*, in contrast with clustering algorithms such as *K-means*, produces clusters with high confidence. The most important metrics in PetaDroid clustering is the homogeneity of the produces clusters.

3 Dataset

Our evaluation dataset contains 10 million Android apps as sampling space for our experiments (over 100 TB) collected across the last ten years from August 2010 to August 2019, as depicted in Table 2. The extensive coverage in size (10 M), time range (06-2010 to 08-2019), and malware families (+300 family) make the result of our evaluation quite compelling.

In Sect. 4.1 and 4.2, to evaluate PetaDroid detection and family clustering, we leverage malware from reference Android malware datasets, namely: MalGenome [44], Drebin [6], MalDozer [18], and AMD [38]. Also, we collected Android malware from VirusShare[5] malware repository. In addition, we use benign apps from AndroZoo [4] dataset (randomly sampling $\approx 100k$ apps from 7.4 Million benign samples in each experiment). In the family clustering evaluation (Sect. 4.2), we use only malware samples from the reference datasets.

Table 2. Evaluation datasets

Name	#Samples	#Families	Time
`MalGenome` [44]	1.3K	49	2010–2011
`Drebin` [6]	5.5k	179	2010–2012
`MalDozer` [18]	21k	20	2010–2016
`AMD` [38]	25k	71	2010–2016
`VirusShare`[8]	33k	/	2010–2017
`MaMaDroid` [31]	40k	/	2010–2017
`AndroZoo` [4]	9.5M	/	2010–Aug 2019

In the comparison (Sect. 5) between PetaDroid, MaMaDroid [27,31], and DroidAPIMiner [3], we apply PetaDroid on the same dataset (benign and malware) used in MaMaDroid evaluation[9] to measure the performance of PetaDroid against state-of-the-art Android malware detection solutions.

To assess PetaDroid obfuscation resiliency (Sect. 4.3), we conduct an obfuscation evaluation on PRAGuard dataset[10], which contains $11k$ obfuscated malicious apps using common obfuscation techniques [26]. Besides, we generate over

[5] https://VirusShare.com.
[9] https://bitbucket.org/gianluca_students/mamadroid_code/src/master/.
[10] http://pralab.diee.unica.it/en/AndroidPRAGuardDataset.

$100k$ benign and malware obfuscated Android apps employing DroidChameleon obfuscation tool [33] using common obfuscation techniques and their combinations.

To assess the adaptation of PetaDroid (Sect. 4.4), we employ the whole AndroZoo[11] [4] dataset (until August 2019), which contains 7.4 million benign apps and 2.1 million malware apps (at least detected as malicious by three vendors), by randomly sampling a dataset ($100k$ malware and benign) in each experiment. We rely on VirusTotal detection of multiple anti-malware vendors in (metadata provided by AndroZoo repository) to label the samples. The dataset covers more than ten years span of Android benign and malware apps [4].

4 Evaluation

In this section, we evaluate PetaDroid framework through a set of experiments and settings involving different datasets.

4.1 Malware Detection

In this section, we report the detection performance of PetaDroid and the effect of hyper-parameters on malware detection performance.

Detection Performance. Table 3 shows PetaDroid *general* and *confidence-based* performance in terms of f1-score, recall, and precision metrics on the reference datasets. In the general performance, PetaDroid achieves a high f1-score 96–99% with a low false-positive rate (precision score of 96.4–99.5% in the general detection). The detection performance is higher under confidence settings. The f1-score is 99% and a very low false-positive rate ($\approx 100k$ benign apps) with a recall score of 99.8% on average. The confidence-based performance causes the filtration of 1–8% low confidence samples from the testing set. In all our experiments, the confidence performance flags $\approx$6% on average, as uncertain decisions, which is a small and realistic value in a deployment with low false positives.

Table 3. General and confidence performances on various reference datasets

Name	General (%)			Confidence (%)		
	F1	P	R	F1	P	R
Genome	99.1	99.5	98.6	99.5	100.	99.0
Drebin	99.1	99.0	99.2	99.6	99.6	99.7
MalDozer	98.6	99.0	98.2	99.5	99.7	99.4
AMD	99.5	99.5	99.5	99.8	99.7	99.8
VShare	96.1	96.4	95.7	99.1	99.7	98.6

[11] https://androzoo.uni.lu/.

4.2 Family Clustering

In this section, we present the results of PetaDroid family clustering on reference datasets (only malware apps). Malware family clustering phase comes after PetaDroid detects a considerable number of malicious Android apps. The number of detected apps could vary from $1k$ (MalGenome [44]) to $+20k$ (Maldozer [18]) samples depending on the deployment. We use *homogeneity* [34] and *coverage* metrics to measure the family clustering performance. The homogeneity metric scores the purity of the produced family clusters. A perfect homogeneity means each produced cluster contains samples from only one malware family. By default, PetaDroid clustering aims only to generate groups with confidence-based while ignoring less certain groups. The coverage metrics score the percentage of the clustered dataset with confidence. We also report the clustering performance after applying the *family matching* (optional step) to cluster all the samples in the dataset (100% coverage).

Table 4. The performance of the family clustering

Clustering metrics	DBSCAN clustering	After family matching
	Homogeneity—Coverage	Homogeneity—Coverage
`Genome`	90.00%—37%	79.67%—100%
`Drebin`	92.28%—49%	80.48%—100%
`MalDozer`	91.27%—55%	81.58%—100%
`AMD`	96.55%—50%	81.37%—100%

Table 4 summarizes the clustering performance in terms of *homogeneity* and *coverage* scores before and after applying the *family matching*. **First**, PetaDroid can produce clusters with high *homogeneity* 90–96% while keeping an acceptable *coverage*, 50% on average. At first glance, 50% *coverage* seems to be a modest result, but we argue that it is satisfactory because: (i) we could extend the coverage, but this might affect the quality of the produced clusters. In the deployment, high confidence clusters with minimum errors and acceptable coverage might be better than perfect coverage (in the case of K-Means clustering algorithm) with a high error rate. (ii) The evaluation datasets have long tail malware families, meaning that most families have only a few samples. This makes the clustering very difficult due to the few samples (less than five samples) in each malware family in the detected dataset. In a real deployment, we could add non-cluster samples to the next clustering iterations. In this case, we might accumulate enough samples to cluster for the long tail malware families. **Second**, after applying the family matching, PetaDroid clusters all the samples in the dataset (100% coverage) and homogeneity decreased to 80–82%, which is acceptable.

4.3 Obfuscation Resiliency

In this section, we report PetaDroid detection performance on obfuscated Android apps. We experiment on: (1) PRAGuard obfuscation dataset [26] ($10k$) and (2) obfuscation dataset generated using DroidChameleon [33] obfuscation tool ($100k$). In the PRAGuard experiment, we combine PRAGuard dataset with $20k$ benign Android apps randomly sampled from the benign apps of AndroZoo repository. We split the dataset equally into build dataset $X_{build} = \{X_{train}, X_{valid}\}$ and test dataset X_{test}. Table 5 presents the detection performance of PetaDroid on different obfuscation techniques. PetaDroid shows high resiliency to common obfuscation techniques by having an almost perfect detection rate, 99.5% f1-score on average.

Table 5. PetaDroid obfuscation resiliency on PRAGuard dataset

ID	Obfuscation techniques	General performance (%)		
		F1 (%)	P (%)	R (%)
1	`Trivial`	99.4	99.4	99.4
2	`String Encryption`	99.4	99.3	99.4
3	`Reflection`	99.5	99.5	99.5
4	`Class Encryption`	99.4	99.4	99.5
5	`(1) + (2)`	99.4	99.4	99.4
6	`(1) + (2) + (3)`	99.4	99.3	99.5
7	`(1) + (2) + (3) + (4)`	99.5	99.4	99.6
	Overall	99.5	99.6	99.4

In the DroidChameleaon experiment, we evaluate PetaDroid on other obfuscation techniques, as shown in Table 6. The generated dataset contains obfuscated benign ($5k$ apps randomly sampled from AndroZoo) and malware samples (originally from Drebin). In the building process of CNN ensemble, we only train with one obfuscation technique (Table 6) and make the evaluation on the rest of the obfuscation techniques. Table 6 reports the result of obfuscation resiliency on DroidChameleon generated dataset. The results show the robustness of PetaDroid. According to this experiment, PetaDroid is able to detect malware obfuscated with common techniques even if the training is done on non-obfuscated datasets. We believe that PetaDroid obfuscation resiliency comes from the usage of (1) Android API (canonical instructions) sequences as features in the machine learning development. Android APIs are crucial in any Android app. A malware developer cannot hide API access, for example *SendSMS*, unless the malicious payload is downloaded at runtime. Therefore, PetaDroid is resilient to common obfuscations as long as they do not remove or hide API access calls. (2) The other factor is fragment-randomization, which makes PetaDroid models

robust to code transformation and obfuscation in general. We argue that training machine learning models on dynamic fragments enhances the resiliency of the models against code transformation.

Table 6. Obfuscation resiliency on DroidChameleon dataset

Obfuscation techniques	General performance		
	F1 (%)	P (%)	R (%)
No Obfuscation	99.7	99.8	99.6
`Class Renaming`	99.6	99.6	99.5
`Method Renaming`	99.7	99.7	99.7
`Field Renaming`	99.7	99.8	99.7
`String Encryption`	99.8	99.8	99.7
`Array Encryption`	99.8	99.8	99.7
`Call Indirection`	99.8	99.8	99.7
`Code Reordering`	99.8	99.8	99.7
`Junk Code Insertion`	99.8	99.8	99.7
`Instruction Insertion`	99.7	99.8	99.7
`Debug Information Removing`	99.8	99.8	99.7
`Disassembling and Reassembling`	99.8	99.8	99.7

4.4 Automatic Adaptation

PetaDroid automatic adaptation goes a step further beyond time resiliency (100k benign and malicious apps every year). PetaDroid employs the confidence performance to collect an extension dataset X_{extend} during the deployment. PetaDroid automatically uses X_{extend} in addition to the previous build dataset as a new build dataset $X_{build(t)} = X_{build(t-1)} \cup X_{extend}$ to build a new ensemble at every new epoch. Table 7 depicts PetaDroid performance with and without automatic adaptation. PetaDroid achieves very good results compared to the previous section. PetaDroid maintains an f1-score in the range of 83–95% during all years. Without adaption, PetaDroid f1-score drops considerably starting from 2017. Table 7 shows the performance of revisiting detection decisions on previous Android apps X_{test} (benign and malware) after updating PetaDroid ensemble using $X_{build} \cup X_{extend}, X_{extend} \subseteq X_{test}$, where the samples in X_{extend} have been removed from X_{test}. The update performance is significantly enhanced in the overall detection during all years. Revisiting malware detection decisions is common practice in app market, (periodic full or partial scan the market's apps), which empowers the use case of PetaDroid automatic adaptation feature and the update metric.

Table 7. Performance of PetaDroid automatic adaptation

Year	No update (F1%)	General (F1%)	Confidence (F1%)	Update (F1%)
2014	98.2	97.0	97.9	99.7
2015	96.1	95.8	96.7	97.5
2016	93.0	93.3	94.8	96.4
2017	70.6	83.9	84.2	95.4
2018	54.8	87.6	91.6	93.8
2019	55.6	96.3	98.7	99.1

5 Comparative Study

In this section, we conduct a comparative study between PetaDroid and state-of-the-art Android malware detection systems, namely: MaMaDroid [27,31], DroidAPIMiner [3], and MalDozer [18]. Our comparison is based on applying PetaDroid on the same dataset (malicious and benign apps) and settings that MaMaDroid used in the evaluation (provided by the authors in [31]). The dataset is composed of 8.5K benign and 35.5K malicious apps in addition to the Drebin [6] dataset. The malicious samples are tagged by time; malicious apps from 2012 (Drebin), 2013, 2014, 2015, and 2016 and benign apps are tagged as oldbenign and newbenign, according to MaMaDroid evaluation.

5.1 Detection Performance Comparison

Table 8 depicts the direct comparison between MaMaDroid and PetaDroid different dataset combinations. In PetaDroid, we present the general and the confidence performance in terms of f1-score. For MaMaDroid and DroidAPIMiner, we present the original evaluation result [31] in terms of f1-score, which are equivalent to the general performance in our case. Notice that we present only the best results of MaMaDroid and DroidAPIMiner as reported in [31].

Table 8. Performance of MaMaDroid, PetaDroid, and DroidAPIMiner

	Peta (F1%)	MaMa (F1%)	Miner (F1%)
	General-Confidence		
`drebin& oldbenign`	**98.94–99.40**	96.00	32.00
`2013& oldbenign`	**99.43–99.81**	97.00	36.00
`2014& oldbenign`	**98.94–99.47**	95.00	62.00
`2014& newbenign`	**99.54–99.83**	99.00	92.00
`2015& newbenign`	**97.98–98.95**	95.00	77.00
`2016& newbenign`	**97.44–98.60**	92.00	36.00

As depicted in Table 8, PetaDroid outperforms MaMaDroid and DroidAPIMiner in all datasets in the general performance. The detection performance gap increases with the confidence-based performance. Notice that the coverage in the confidence-based settings is almost perfect for all the experiments in Table 8.

5.2 Efficiency Comparison

In Table 9, we report the required average time for MaMaDroid and PetaDroid to fingerprint one Android app. PetaDroid takes 03.58 ± 04.21 s on average for the whole process (DEX disassembly, assembly preprocessing, CNN ensemble inference). MaMaDroid, compared to PetaDroid, tends to be slower due to the heavy preprocessing. MaMaDroid preprocessing [31] is composed of the call graph extraction, sequence extraction, and Markov change modeling, which require 25.40 ± 63.00, 1.73 ± 3.2, 6.7 ± 3.8 s respectively for benign samples and 09.20 ± 14.00, 1.67 ± 3.1, 2.5 ± 3.2 s respectively for malicious samples. On average, PetaDroid (3.58 s) is approximately eight times faster than MaMaDroid.

Table 9. MaMaDroid and PetaDroid runtime

	PetaDroid (seconds)	MaMaDroid (seconds)
`Malware`	02.64 ± 03.94	$09.20 \pm 14.00 + 1.67 \pm 3.1 + 2.5 \pm 3.2$
`Benign`	05.54 ± 05.12	$25.40 \pm 63.00 + 1.73 \pm 3.2 + 6.7 \pm 3.8$
`Average`	03.58 ± 04.21	$\approx$23 s

5.3 Time Resiliency Comparison

MaMaDroid evaluation emphasizes the importance of time resiliency for modern Android malware detection. Table 10 depicts the performance with different dataset settings, such as training using an old malware dataset and testing on a newer one. PetaDroid outperforms (or obtains a very similar result in few cases) MaMaDroid and DroidAPIMiner in all settings. Furthermore, the results show that PetaDroid is more robust to time resiliency compared to MaMaDroid [31].

Table 10. Time Resiliency of MaMaDroid, PetaDroid, DroidAPIMiner.

Testing Sets	drebin & oldbenign			2013 & oldbenign			2014 & oldbenign			2015 & oldbenign			2016 & oldbenign		
Training Sets	Miner	MaMa	Peta	Miner	MaMa	Peta	Miner	MaMa	Peta	Miner	MaMa	Peta	Miner	MaMa	Peta
drebin&oldbenign	32.0	96.0	**99.4**	35.0	95.0	**98.6**	34.0	72.0	**77.5**	30.0	39.0	**44.0**	33.0	42.0	**47.0**
2013&oldbenign	33.0	94.0	**97.8**	36.0	97.0	**99.6**	35.0	73.0	**85.4**	31.0	37.0	**59.3**	33.0	28.0	**56.6**
2014&oldbenign	36.0	92.0	**95.8**	39.0	93.0	**98.6**	62.0	95.0	**99.4**	33.0	78.0	**91.4**	37.0	75.0	**88.9**
	drebin & newbenign			2013 & newbenign			2014 & newbenign			2015 & newbenign			2016 & newbenign		
Training Sets	Miner	MaMa	Peta	Miner	MaMa	Peta	Miner	MaMa	Peta	Miner	MaMa	Peta	Miner	MaMa	Peta
2014&newbenign	76.0	98.0	**99.3**	75.0	98.0	**99.7**	92.0	99.0	**99.8**	67.0	85.0	**91.4**	65.0	81.0	**82.1**
2015&newbenign	68.0	97.0	**97.1**	68.0	97.0	**97.8**	69.0	**99.0**	98.9	77.0	95.0	**99.0**	65.0	88.0	**95.4**
2016&newbenign	33.0	**96.0**	95.6	35.0	98.0	**98.2**	36.0	**98.0**	97.9	34.0	92.0	**95.2**	36.0	92.0	**98.3**

5.4 PetaDroid and Maldozer Comparison

In this section, we compare PetaDroid with MalDozer [18] to check the effectiveness of the proposed approach. Specifically, we evaluate the performance of both detection systems on raw Android datasets without any code transformation. Afterward, we evaluate the systems on randomization transformation (Sect. 2.2). Table 11 shows the effectiveness comparison between the detection systems. **First**, PetaDroid outperforms MalDozer in all the evaluation dataset without code transformation. One major factor to this result is the usage of the machine learning model ensemble to enhance the detection performance. **Second**, this gap significantly increases when we use code transformation in the various evaluation datasets. PetaDroid preserves the high detection performance due to the fragment randomization technique used in the training phase. As depicted in Table 11, the evaluation result shows the enhancement that the fragment randomization technique adds to the Android malware detection overall to enhance the resiliency.

Table 11. PetaDroid and MalDozer Comparison

	PetaDroid (F1 %)	MalDozer (F1 %)
	Raw-Randomization	Raw-Randomization
`MalGenome`	99.6-99.3	98.1-92.5
`Drebin`	99.2-99.1	97.4-91.6
`MalDozer`	98.5-98.6	95.2-89.3
`AMD`	99.4-99.5	96.1-90.1
`VShare`	95.8-96.0	94.2-88.1

6 Related Work

The Android malware analysis techniques can be classified to *static analysis*, *dynamic analysis*, or *hybrid analysis*. The static analysis methods [5,6,20,39] use static features that are extracted from the app, such as: requested permissions and APIs to detect malicious app. The dynamic analysis methods [8,16,21,36] aim to identify behavioral signature or behavioral anomaly of the running app. These methods are more resistant to obfuscation. The dynamic methods offer limited scalability as they incur additional cost in terms of processing and memory. The hybrid analysis [15,25], combine between both analyses to improve detection accuracy, which costs additional computational cost. Assuming that malicious apps of the same family share similar features, some methods [17,19,22], measure the similarity between the features of two samples (similar malicious code). The deep learning techniques are more suitable than conventional machine learning techniques for Android malware detection [41]. Research

works on deep learning for Android malware detection are recently getting more attention [18,43]. These deep learning models are more venerable to common machine learning adversarial attacks as described in [9]. In contrast, PetaDroid employs the ensemble technique to mitigate such adversarial attacks [37] and to enhance the overall performance. In DroidEvolver [40], the authors use online machine learning techniques to enhance the time resiliency of the Android malware detection system. In contrast, PetaDroid employs batch training techniques instead of online training, which means that in each epoch t PetaDroid builds new models using the extended dataset at once. We argue that batch learning could generalize better since the training system has a complete view of the app dataset. It is less venerable to biases that could be introduced by the order of the apps in online training.

PetaDroid provides Android malware detection and family clustering using advanced natural language processing and machine learning techniques. PetaDroid is resilient to common obfuscation techniques due to code randomization during the training. PetaDroid introduces a novel automatic adaption technique inspired from [24] that leverages the result confidence to build a new CNN ensemble on confidence detection samples. Our automatic adaptation technique aims to overcome the issue of new Android APIs over time, while other methods could be less resilient and might require updates with a manually crafted dataset. The empirical comparison with state-of-the-art solutions, MaMaDroid [31] and MalDozer [18], shows that PetaDroid outperforms MaMaDroid and MalDozer under the various evaluation settings in the malware detection effectiveness and efficiency.

7 Limitation

Although the high obfuscation resiliency of PetaDroid showed in Sect. 4.3, PetaDroid is not immune to complex obfuscation techniques. Also, PetaDroid most likely will not be able to detect Android malware that downloads the payload during runtime. PetaDroid focuses on the fingerprinting process on DEX bytecode. Therefore, Android malware, which employs C/C++ native code, is less likely to be detected because we do not consider native code in our fingerprinting process. Covering native code is a possible future enhancement for PetaDroid. We consider including selective dynamic analysis for low confidence detection as future work. The latter will empower PetaDroid against sophisticated obfuscation techniques. Also, PetaDroid system needs more validation on real world deployments to check the performance as proposed in previous investigations [14,32]. Also, we need to check the correctness of the dataset split to prevent bias results as a result of *spatial bias* and *temporal bias* [32]. In Sect. 5.3 and 7, we partially addressed this issues by (1) evaluating the system on temporal splits from AndroZoo dataset and (2) employing collected samples dataset (VirusShare) in addition to multiple references datasets.

8 Conclusion

In this paper, we presented PetaDroid, an Android malware detection and family clustering framework for large scale deployments. PetaDroid employs supervised machine learning, an ensemble of CNN models on top of *Inst2Vec* features, to fingerprint Android malicious apps accurately. DBScan clustering on top of *Inst-NGram2Vec* and deep auto-encoders features, to cluster highly similar malicious apps into their most likely malware family groups. In PetaDroid, we introduced fragment-based detection, in which we randomize the macro-action of Dalvik assembly instructions while keeping the inner order of methods' sequences. We introduced the automatic adaption technique that leverages confidence-based decision making to build a new CNN ensemble on confidence detection samples. PetaDroid achieved high detection (98–99% f1-score) and family clustering (96% cluster homogeneity) performance. Our comparative study between PetaDroid, MaMaDroid [31] and MalDozer shows that PetaDroid outperforms state-of-the-art solutions on various evaluation settings.

References

1. Cyber attacks on Android devices on the rise (2018). https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise
2. Mobile OS market share (2019). http://gs.statcounter.com/os-market-share/mobile/worldwide
3. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICST, vol. 127, pp. 86–103. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-04283-1_6
4. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: AndroZoo: collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories (2016)
5. Amira, A., Derhab, A., Karbab, E.B., Nouali, O., Khan, F.A.: Tridroid: a triage and classification framework for fast detection of mobile threats in android markets. J. Ambient Intell. Humaniz. Comput. **12**, 1731–1755 (2021)
6. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., et al.: DREBIN: effective and explainable detection of Android malware in your pocket. In: Symposium Network and Distributed System Security (2014)
7. Bai, Y., Xing, Z., Ma, D., Li, X., Feng, Z.: Comparative analysis of feature representations and machine learning methods in android family classification. Comput. Netw. **184**, 107639 (2021)
8. Canfora, G., Medvet, E.: Acquiring and analyzing app metrics for effective mobile malware detection. In: Proceedings of the 2016 ACM on International Workshop on Security and Privacy Analytics (2016)
9. Chen, X., et al.: Android HIV: a study of repackaging malware for evading machine-learning detection. IEEE Trans. Inf. Forensics Secur. **15**, 987–1001 (2020)
10. Ding, S.H.H., Fung, B.C.M., Charland, P.: Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: Security and Privacy (2019)

11. Ester, M., Kriegel, H., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press (1996)
12. Garcia, J., Hammad, M., Malek, S.: Lightweight, obfuscation-resilient detection and family identification of Android malware. ACM Trans. Softw. Eng. Methodol. **26**, 1–29 (2018)
13. Goodfellow, I., Bengio, Y., et al.: Deep Learning. MIT Press, Cambridge (2016)
14. Jordaney, R., et al.: Transcend: detecting concept drift in malware classification models. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16–18, 2017 (2017)
15. Karbab, E.B., Debbabi, M.: ToGather: automatic investigation of android malware cyber-infrastructures. In: Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES (2018)
16. Karbab, E.B., Debbabi, M.: Maldy: portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports. Digit. Investig. **28**, S77–S87 (2019)
17. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: Cypider: building community-based cyber-defense infrastructure for Android malware detection. In: ACM Computer Security Applications Conference (ACSAC) (2016)
18. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: MalDozer: automatic framework for Android malware detection using deep learning. Digit. Investig. **24**, S48–S59 (2018)
19. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: Scalable and robust unsupervised android malware fingerprinting using community-based network partitioning. Comput. Secur. **97**, 101965 (2020)
20. Karbab, E.B., Debbabi, M., Mouheb, D.: Fingerprinting Android packaging: generating DNAs for malware detection. Digit. Investig. **18**, S33–S45 (2016)
21. Karbab, E.M.B., Debbabi, M., Alrabaee, S., Mouheb, D.: DySign: dynamic fingerprinting for the automatic detection of Android malware. In: International Conference on Malicious and Unwanted Software (2016)
22. Kim, J., al. Structural information based malicious app similarity calculation and clustering. In: Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems (2015)
23. Kim, Y.: Convolutional neural networks for sentence classification. CoRR (2014)
24. Lakshminarayanan, B., Pritzel, A., Blundell, C.: Simple and scalable predictive uncertainty estimation using deep ensembles. In: Annual Conference on Neural Information Processing Systems (2017)
25. Lindorfer, M., Neugschwandtner, M., et al.: Andrubis-1,000,000 apps later: a view on current Android malware behaviors. In: Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS). IEEE (2014)
26. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: an extended insight into the obfuscation effects on Android malware. Comput. Secur. **51**, 16–31 (2015)
27. Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: MaMaDroid: detecting Android malware by building Markov chains of behavioral models. In: NDSS (2017)
28. Massarelli, L., Aniello, L., Ciccotelli, C., Querzoni, L., Ucci, D., Baldoni, R.: Android malware family classification based on resource consumption over time. In: 12th International Conference on Malicious and Unwanted Software, MALWARE 2017, Fajardo, PR, USA, October 11–14, 2017 (2017)
29. McLaughlin, N., et al.: Deep Android malware detection. In: CODASPY (2017)

30. Mikolov, T., Sutskever, I., et al.: Distributed representations of words and phrases and their compositionality. In: NIPS Neural Information Processing Systems (2013)
31. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G.J., Stringhini, G.: MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version). ACM Trans. Priv. Secur. **22**, 1–34 (2019)
32. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: TESSERACT: eliminating experimental bias in malware classification across space and time. In: USENIX (2019)
33. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: evaluating android anti-malware against transformation attacks. In: 8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2013 (2013)
34. Rosenberg, A., Hirschberg, J.: V-measure: a conditional entropy-based external cluster evaluation measure. In: EMNLP-CoNLL (2007)
35. Shi, Q., et al.: Hash kernels. In: International Conference on Artificial Intelligence and Statistics (AISTATS) (2009)
36. Suarez-Tangil, G., et al.: DroidSieve: fast and accurate classification of obfuscated Android malware. In: Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY 2017), pp. 309–320 (2017)
37. Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I.J., Boneh, D., McDaniel, P.D.: Ensemble adversarial training: attacks and defenses. In: 6th International Conference on Learning Representations, ICLR 2018 (2018)
38. Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current Android malware. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 252–276. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_12
39. Wu, Y., Li, X., Zou, D., Yang, W., Zhang, X., Jin, H.: MalScan: fast market-wide mobile malware scanning by social-network centrality analysis. In: 34th IEEE/ACM International Conference on Automated Software Engineering (2019)
40. Xu, K., Li, Y., Deng, R., Chen, K., Xu, J.: DroidEvolver: self-evolving android malware detection system. In: IEEE European Symposium on Security and Privacy (2019)
41. Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-Sec: deep learning in android malware detection. In: ACM SIGCOMM Computer Communication Review (2014)
42. Zhang, X., Zhao, J.J., LeCun, Y.: Character-level convolutional networks for text classification. In: Advances in Neural Information Processing Systems (2015)
43. Zhang, Y., et al.: Familial clustering for weakly-labeled Android malware using hybrid representation learning. IEEE Trans. Inf. Forensics Secur. **15**, 3401–3414 (2020)
44. Zhou, Y., Jiang, X.: Dissecting Android malware: characterization and evolution. In: IEEE Symposium on Security and Privacy (SP) (2012)