Journal of Network and Systems Management (2022) 30:22
https://doi.org/10.1007/s10922-021-09634-4

Effective and Efficient Hybrid Android Malware Classification Using Pseudo-Label Stacked Auto-Encoder

Samaneh Mahdavifar[1] · Dima Alhadidi[2] · Ali. A. Ghorbani[1]

Received: 2 January 2021 / Revised: 30 August 2021 / Accepted: 29 September 2021 /
Published online: 2 November 2021

Abstract

Android has become the target of attackers because of its popularity. The detection of Android mobile malware has become increasingly important due to its significant threat. Supervised machine learning, which has been used to detect Android malware is far from perfect because it requires a significant amount of labeled data. Since labeled data is expensive and difficult to get while unlabeled data is abundant and cheap in this context, we resort to a semi-supervised learning technique, namely pseudo-label stacked auto-encoder (PLSAE), which involves training using a set of labeled and unlabeled instances. We use a hybrid approach of dynamic analysis and static analysis to craft feature vectors. We evaluate our proposed model on CICMal-Droid2020, which includes 17,341 most recent samples of five different Android apps categories. After that, we compare the results with state-of-the-art techniques in terms of accuracy and efficiency. Experimental results show that our proposed framework outperforms other semi-supervised approaches and common machine learning algorithms.

Keywords Android malware · Category · Classification · Hybrid analysis · Semi-supervised learning · Stacked auto-encoder · Deep learning

✉ Samaneh Mahdavifar
smahdavi@unb.ca

Dima Alhadidi
dima.alhadidi@uwindsor.ca

Ali. A. Ghorbani
ghorbani@unb.ca

[1] Canadian Institute for Cybersecurity (CIC), Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada

[2] School of Computer Science, University of Windsor, Windsor, ON, Canada

1 Introduction

In the first quarter of 2020, around 86.3 percent of all smartphones sold to end users were phones with the Android operating system [1]. Android mobile devices have become ubiquitous, with trends showing that the recent pace of adoption is unlikely to slow down. Android popularity has its cost. The popularity has spurred an alarming growth in Android malware. Compromising a mobile device can have a damaging effect on enterprises and individuals. Most enterprises are still not prepared to deal with persistent mobile security threats. It is crucial to detect and prevent mobile security threats as it is the case with any other type of threats [2, 3] in order to keep employees' and customers' information secure. Targeting individuals, Android malware can read your contact list and messages, including the ones about bank transactions and one-time passwords. Android malware can also access your pictures and know your location, house address, email account details and restaurants you visited. Moreover, it can take over your device, encrypt its content, and hold the files hostage until you pay a ransom (usually in Bitcoin) to have them released. This demonstrates the need for security solutions to protect users from Android malware that can steal and access confidential data.

Although many malware detection systems have been proposed [4–6], there are common limitations to the existing solutions. First, detecting malicious apps is not enough anymore. Specifying the category of Android malware is an important step to prioritize mitigation techniques. Second, static-based malware detection [7–9] or dynamic-based malware detection is not enough. Static-based malware detection is ineffective against advanced malware programs that utilize sophisticated evasion detection techniques such as polymorphism (modifying code to defeat signature-based tools) or obfuscation (hiding evidence of malicious activities). On the other hand, the dynamic analysis is not scalable due to its limited coverage. Third, supervised learning techniques to detect malware depend on several features of both malicious and benign software [8–10] and they need a high number of labeled instances. For a real-world problem such as malware analysis, it is quite challenging to label data because it is a time-consuming process, and in some cases, pieces of malware can avoid detection. The cost associated with the labeling process thus may render a fully labeled training set infeasible, whereas the acquisition of unlabeled data is relatively inexpensive. Semi-supervised learning is one type of machine learning technique that is very useful when a limited amount of labeled data exists for each class. Fourth, machine learning models have proven to be insufficient to solve real-world problems with intrinsic complexity and massive amounts of data since they depend on a manual feature extraction process. On the other hand, deep learning algorithms take care of extracting abstract and flexible features automatically from raw data which helps generalization in the classification.

In this paper, we extend our prior research [11], where we have addressed some of these limitations by detecting and categorizing Android malware, and adopting semi-supervised and deep learning. In our prior research [11], we proposed the first work that applies a semi-supervised Deep Neural Network (DNN) for dynamic malware category classification namely Pseudo-Label Deep Neural Network (PLDNN) [11]. In this paper, we propose a simple, yet practical and efficient framework for Android malware category classification. The new characteristics of this approach in contrast to the previous one [11] are: (1) it is a hybrid approach that integrates both static and dynamic analysis of malware and (2) it applies a semi-supervised Pseudo-Label Stacked Auto-Encoder (PLSAE) for malware category classification. The hybrid approach utilizes the strengths of both static and dynamic analysis. Unlike PLDNN, PLSAE consists of stacking multiple Auto-Encoders (AEs) and benefits from the unsupervised pre-training that leads the network towards global minima and supports better generalization. Specifically, our contributions can be summarized as follows:

- We analyze both statically and dynamically extracted features of malware samples using PLSAE. We adopt CopperDroid, a Virtual Machine Introspection (VMI)-based analysis system [12], to extract the static and dynamic features. The results of the static and dynamic analysis are available for researchers together with malware samples.[1]
- We experimentally analyze the results and find that the model can detect and categorize malware with an accuracy of 98.28 percent and a false positive rate of 1.16 percent.
- We conduct a comparative study between PLSAE (semi-supervised learning and hybrid analysis), PLDNN (semi-supervised and dynamic) [11], Label Propagation (LP; semi-supervised and hybrid), and other common machine learning algorithms (supervised and hybrid) such as Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbor (k-NN). We show that our semi-supervised deep learning model significantly outperforms LP, which is a popular semi-supervised machine learning algorithm, PLDNN, RF, SVM, and k-NN for every number of labeled training samples. In particular, PLSAE shows outstanding performance (accuracy=95.19%) with only 1% of labeled training samples, i.e., 100 as compared with PLDNN (accuracy=91.63%).

The rest of the paper is organized as follows. Related work is discussed in Sect. 2. In Sect. 3, we present background information. Section 4 describes the threat model. Our proposed framework is detailed in Sect. 5. Sect. 6 discusses the performance analysis. Finally, Sect. 7 concludes the paper and includes some future research directions.

2 Related Work

The rising number of Android malwares has become a concern for both academia and industry. Some studies target Natural Language Processing (NLP) [23] and reused Free Open-Source Software (FOSS) [24] packages to detect malware.

[1] https://www.unb.ca/cic/datasets/maldroid-2020.html

Machine learning algorithms such as Naive Bayes (NB), SVM, Decision Tree (DT), and k-NN are commonly used by academic and industrial researchers in detecting Android malware [5, 25]. This traditional approach is based on information gathered from program analysis, where the performance depends on the accuracy of the extracted features. Deep learning, on the other hand, is a branch of machine learning that attempts to study high-level features effectively and efficiently from the original data [26]. This approach has been widely used in various areas including image processing, visual recognition, and object detection [27–29]. In this section, we review some of the attempts that focused on deep learning algorithms in Android malware detection.

One of the first attempts was conducted by Yuan et al. [13] where they utilized more than 200 dynamic and static features including required permissions, sensitive API calls, and dynamic behaviors, which are extracted from both the static and dynamic analysis of Android apps for malware binary detection. The proposed framework employed Deep Belief Network (DBN) based on stacked Restricted Boltzmann Machine (RMB) in two phases of unsupervised pre-training and supervised fine-tuning. The work demonstrated that the deep learning technique is suitable for Android malware detection and can achieve up to 96% accuracy with real-world Android application sets. This study was quickly followed by a series of more advanced detection approaches focused on deep learning algorithms. In 2016, Hou et al. [9] proposed DroidDelver, an Android malware detection system using DBN based on API call blocks. They categorized the API calls of the Smali code, which belong to the same method into a block, and employed a DBN built on stacked RBMs for unknown Android malware detection. The authors argued that a block of sensitive API calls can be used as a feature to detect Android malware. Promising experimental results demonstrated that DroidDelver method outperformed alternative Android malware detection techniques having an accuracy of 96.66%. In the same year, DroidDeep was presented by Su et al. [14]. They first extracted more than 30 thousand static features based on the five categories: requested permission, used permission, sensitive API call, action, and application component. They built the deep learning model, i.e., stacked RBMs to learn typical features for classification and used SVM classifier for detecting Android malware using static features. DroidDeep achieved a 99.4% detection accuracy with a reasonable running time that can be adaptable to large scale Android malware detection.

Another work by Nix *et. al.* [15] focused on the family classification of Android malware and applications using system API-call sequences and studied the effectiveness of Convolutional Neural Networks (CNNs) and Long Short Term Memory (LSTM). They designed a pseudo-dynamic program analyzer that generates a sequence of API calls along the program execution path and compared their approach with LSTM, n-gram based SVM, and bag of words-based NB. Both CNN and LSTM outperformed n-gram based methods while CNN showed a better performance in comparison with LSTM. In another approach, Huang et al. [16] aimed of automating Android feature extraction where they transformed

Android app bytecode into Red, Green, Blue (RGB) color code. The results of encoded color images are used as input features to a CNN that is trained with over one million malware samples and one million benign samples. In a similar way, Wang et al. [17] presented a hybrid model based on deep AE. CNN has been proposed to detect Android malware in which a deep AE is used as a pre-training method for the CNN. They implemented a multiple-CNN architecture during the training process and discovered that serial CNN showed better feature extraction ability by combining the convolutional layer and the pooling layer with the fully connected layer. Karbab et al. [10] proposed an Android malware detection and family attribution framework in which they extracted raw sequences of API calls from .dex assembly code. The vectorized API calls are then trained using a CNN comprising one convolutional layer using ReLU, one max-pooling layer, one fully connected layer having dropout and bench normalization, and one output layer for two-task classification. In another research contribution [19], important values of an Android Package Kit (APK) code are visualized on an image and then fed into a CNN to detect mutated Android malware. Xiao et al. [18] considered one system call sequence of an Android malware as a sentence and applied two LSTM language models to train malware and benign samples. To classify an APK under analysis, they measured the similarity score of the sequence using the two trained networks. Kim et al. [8] in 2019 proposed a multimodal deep learning method based on Feed-Forward Neural Network (FFNN) that employed seven features extracted by analyzing Android files such as a manifest file, a .dex file, and a .so file from an APK file to be used in Android malware detection. They showed that it was possible to maximize the benefits of encompassing multiple feature types. They conducted various experiments with a total of 41,260 samples and compared the accuracy of their model with other DNN and detection models and demonstrated that their detection model was effective and efficient to be used in Android malware detection. Lu et al. [20] extracted static features and dynamic behavioral features with strong anti-obfuscation ability. Then, they built a hybrid deep learning model for Android malware detection. The DBN is used to process the static features whereas the GRU is used to process the dynamic feature sequence. Finally, the training results of DBN and GRU are input into the Back-Propagation (BP) neural network, and the final classification results are output. Experimental results show that, compared with the traditional machine learning algorithms, the Android malware detection model based on hybrid deep learning algorithms has a higher detection accuracy, and it also has a better detection effect on obfuscated malware.

The methods and techniques of the previous work mostly focused on malware detection (not malware family or category classification). They implemented static analysis by either analyzing or disassembling the code rather than running it on a real device or an emulated environment. In addition, they used supervised learning that needs a large volume of labeled data. Any malware filtering algorithm should be robust enough against injected unknown or noise data. In contrast to the previous studies as summarized in Table 1, we propose a simple, yet effective and efficient framework for Android malware category classification using DNNs that resort to static and dynamic analysis to utilize the benefit of both and semi-supervised

Table 1 Related work summary of Android malware detection

Year	Authors	Analysis type	Deep model	Type	Features	Functionality
2014	Yuan et al. [13]	Static and dynamic	DBN built on stacked RBMs	Supervised	Required permission, sensitive API , dynamic behavior	Binary detection
2016	Hou et al. [9]	Static	DBN built on stacked RBMs	Supervised	API call blocks extracted from the Smali codes	Binary detection
2016	Su et al. [14]	Static	Stacked RBMs, SVM	Supervised	Requested permissions, used permissions, sensitive API calls, actions, application components	Binary detection
2017	Nix et al. [15]	Static	CNN, LSTM	Supervised	API call sequences	Detection and family classification
2018	Huang et al. [16]	Static	CNN	Supervised	App bytecode transformed into images	Binary detection
2018	Wang et al. [17]	Static	DAE, CNN	Supervised	Requested permissions, filtered intents, restricted API calls, hardware features, code related patterns, suspicious API calls	Binary detection
2018	Karbab et al. [10]	Static	CNN	Supervised	API method calls	Detection and family classification
2019	Kim et al. [8]	Static	multimodal FFNN	Supervised	Strings, permission, components, environmental info, method opcode, shared lib opcode, API calls,	Binary detection
2019	Xiao et al. [18]	Dynamic	LSTM	Supervised	System call sequences	Binary detection
2019	Yen et al. [19]	Static	CNN	Supervised	Use text TF-IDF to convert code to RGB images	Binary detection
2020	Lu et al. [20]	Static and dynamic	DBN, GRU	Supervised	Resource features, semantic features, dynamic behavioral features	Binary detection

Table 1 (continued)

Year	Authors	Analysis type	Deep model	Type	Features	Functionality
2020	Mahdavifar et al. [11]	Dynamic	Pseudo-Label FFNN	Semi-supervised	System calls, binders, composite behaviors	Detection and family classification
2021	Our approach	Static and dynamic	Pseudo-Label SAE	Semi-supervised	Permissions, intents, services, receivers, ..., system calls, binders, composite behaviors	Detection and family classification

Year	Authors	Analysis type	Model	Semi-supervised type	Features	Functionality
2015	Ma et al. [21]	Static	SVM, ECASSL[a]	Machine Learning	Topic features and sensitive API usage	Binary detection
2017	Chen et al. [22]	Dynamic	MBSS[b]	Machine Learning	API calls	Binary detection
2020	Mahdavifar et al. [11]	Dynamic	Pseudo-Label FFNN	Deep Learning	System calls, binders, composite behaviors	Detection and family classification
2021	Our approach	Static and dynamic	Pseudo-Label SAE	Deep Learning	Permissions, intents, services, receivers,..., system calls, binders, composite behaviors	Detection and family classification

[a] Ensured Collaborative Active and Semi-Supervised Labeling

[b] Model-based Semi-Supervised

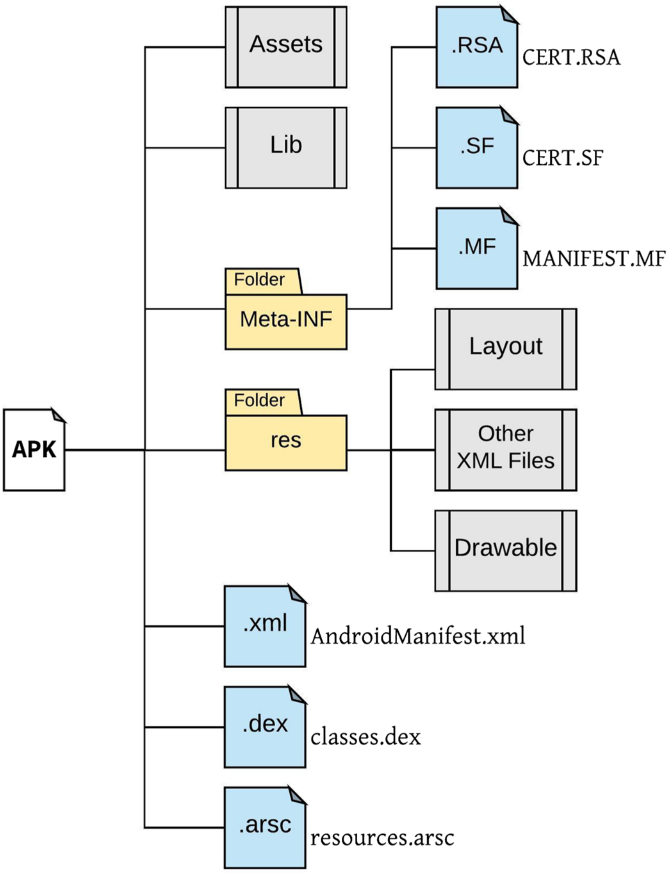


Fig. 1 Structure of an Android Package Kit (APK) [30]

learning to avoid expensive labeling process. There are some studies [21, 22] that target the semi-supervised learning for malware detection using machine learning techniques, however, our approach utilizes deep learning to classify Android malware into different categories (Table 1).

3 Background

In this section, we present some preliminaries on Android apps, deep learning, and semi-supervised learning to help understanding the proposed framework.

3.1 Android Apps

An Android app is packaged into an APK; a zip archive file format consisting of several files and folders including the application code, resources, as well as the application manifest file as illustrated in Fig. 1. Just like Windows (PC) systems that use a .exe file for software installation, Android systems use an APK file for distribution and installation of mobile applications. There are two primary sources for applications [30]:

- **Pre-installed applications:** Android includes a set of pre-installed applications as part of its open source platform or device manufacturer. e.g., phone, email, calendar, web browser, and contacts.
- **User-installed applications:** Android provides an open development environment that supports any third-party application from another market such as Yandex (Russian users) and 360 Mobile Assistant (Chinese users).

The structure of an APK file typically contains the following seven components:

1. **Assets:** This optional directory contains applications' assets, which can be retrieved by *AssetManager*.
2. **Lib:** This optional directory contains compiled code, e.g., native libraries that can be used through the Native Development Kit (NDK).
3. **META-INF**: This directory contains several files responsible for ensuring the integrity and security of the application. It includes three files: (1) MANIFEST. MF – the manifest file which stores metadata of the application, (2) CERT.RSA – the digital certificate of the application, and (3) CERT.SF – the file containing a list of resources and SHA-1 digest. The signature of the APK is also stored in this directory.
4. **Res:** This directory contains noncompiled resources (resources not compiled into *resources.arsc*), which defines UI layouts, menus, animations, languages, sound settings, etc. Typically, the directory contains the following three sub-directories: (1) drawable, (2) layout, and (3) other Extensible Markup Language (XML) files.
5. **AndroidManifest.xml:** An essential file in every Android application. It provides vital application details such as the unique application identifier, permissions required by the application, application version, referenced libraries, and description of several application components such as activities, services, broadcast receivers and content providers. These components are discussed below:
   - Activity: An activity is the user interface component of an application that is launched using the Intents. The number of activities that can be declared within the manifest depends on the developer requirements.
   - Service: The service component represents one of the two tasks; either to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use. It also performs background tasks without any UI; e.g., playing an audio or downloading data from the network.
   - Broadcast Receiver: This component can be used as a messaging system across apps where it listens to the Android system generated events. For instance, the Android system sends broadcasts when various system events

occur, such as when the system boots up (BOOT_COMPLETED) or the device starts charging (BATTERY_CHANGED_ACTION).

- Content Provider: Content provider (also known as the data-store) is a standard interface that connects data in one process with code running in another process.

6. **classes.dex:** The *.dex* file is a core of an APK, which refers to a binary container for the code and the associated data. It includes a header containing meta-data about the executable followed by identifier lists that contain references to strings, types, prototypes, fields, methods and classes employed by the executable. The final part of the *.dex* file is the data section that contains the code and the data (i.e., URLs).
7. **resources.arsc:** This file contains pre-compiled resources such as binary XML.

3.2 Deep Learning

For decades, conventional machine learning techniques were very limited in processing raw data and discovering the internal representations needed for pattern recognition or classification. Domain experts used to engineer features manually based on their expertise and transform the raw input data into feature vectors that, in turn, could be fed into a machine learning or pattern recognition system [31]. This process was so computationally expensive and error-prone because it depends on human engineers.

Deep learning methods, so-called representation-learning methods, learn high-level representations of input data with multiple levels of abstractions and a set of flexible features are obtained as the output of non-linear layer-wise processing. Although these non-linear modules are doing some simple calculations at each layer to transform the representation at one layer into a representation at a higher layer, one can learn very complex functions by composing these transformations. What distinguishes deep learning from other shallow learning algorithms is that different layers of features are not extracted by human agents, but they are learned hierarchically from input data using a general-purpose learning procedure. Deep learning models including, but not limited to FFNN, CNN, Recurrent Neural Network (RNN), SAE, and DBN have shown great promise in big data analysis with a wide range of applications, i.e., large scale image recognition tasks, NLP, bioinformatics, and speech recognition [32–36]. Deep learning today is mostly about pure supervised learning. A major drawback of supervised learning is that it requires a lot of labeled data and it is quite expensive to collect them. So, deep learning in the future is expected to be unsupervised, more human-like [37].

3.2.1 Stacked Auto-Encoder (SAE)

An AE is a generative neural network that encodes input x into a latent variable using an encoder and then decodes the code into an approximate reconstruction of the input vector x' using a decoder. Training the network is done in an unsupervised fashion where usually the aim is to learn a representation vector z for dimensionality reduction.

Suppose $x \in \mathbb{R}^m$ is an input sample, the encoder maps x into a representation vector $z \in \mathbb{R}^n$ through a deterministic function f:

$$z = f(Wx + b), \tag{1}$$

where W is a weight matrix, and b is a bias vector. The decoder maps back z into a reconstructed vector x' of the input x with identical dimensionality:

$$x' = f'(W'z + b'), \tag{2}$$

where f', W', and b' are the activation function, weight matrix, and bias vector, respectively, that might be different from the encoder's parameters. The objective of training an AE is to minimize the reconstruction loss l like Mean Squared Errors (MSE) as the equation below:

$$l(W, b, W', b') = \frac{1}{h} \sum_{i=1}^{h} \| x_i - x_i' \|^2, \tag{3}$$

where h is the total number of training samples, x_i is the input, and x_i' is the reconstruction of x_i.

An SAE consists of several layers of AE where output of each hidden layer is connected to the input of the next hidden layer. The hidden layers are first pretrained using an unsupervised method and then are fine-tuned by a supervised algorithm with a classifier at the top.

3.3 Semi-Supervised Learning

Semi-supervised learning is a class of machine learning methods that takes advantage of labeled data to predict the labels for unlabeled samples to increase detection accuracy. This method is specifically useful in many applications such as document retrieval, image search, genomics, and natural language parsing, where we are dealing with a limited amount of labeled data and an abundant amount of unlabeled data. In real-world scenarios, it is very expensive or impossible to obtain labels for the data, while unlabeled data can be acquired in large quantities.

One of the seminal contributions in the area of semi-supervised learning is co-training where two classifiers are trained separately on two different conditionally independent views of the training data. The most confident predictions of each classifier on new unlabeled examples are then used to iteratively enlarge the training set of the other [38, 39]. Another straightforward approach is self-training in which an initial model is incrementally trained using highly confident self-predictions over unlabeled data [40]. The main disadvantage of these methods is that in case of the existence of wrong predictions with high confidence, the prediction error is propagated into model learning. Transductive SVMs (TSVMs) [41, 42] are other common approaches that aim at finding a hyperplane which is far away from the unlabeled points. The most important drawback of TSVMs is the lack of efficient optimization

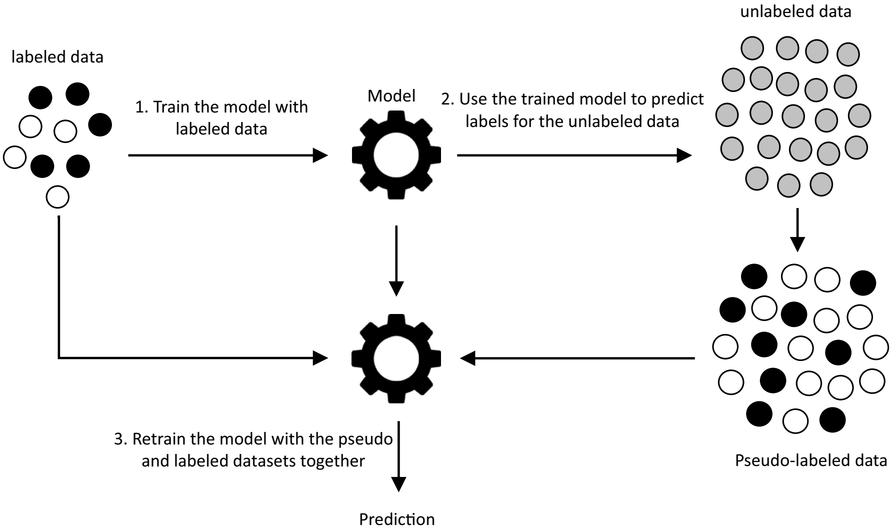

labeled data
1. Train the model with labeled data
Model
2. Use the trained model to predict labels for the unlabeled data
unlabeled data
Pseudo-labeled data
3. Retrain the model with the pseudo and labeled datasets together
Prediction

Fig. 2 Three main stages of the Pseudo-Label algorithm

because of non-convex objective functions. Graph-based methods utilize minimum energy configuration for label propagation throughout the graph, however, are not scalable enough because they are sensitive to the graph structure [43, 44]. Deep learning capability in learning internal representations of raw data has motivated several deep learning-based semi-supervised methods during the past decade [45–49]. They mostly focus on minimizing the sum of supervised and unsupervised cost functions simultaneously to enable semi-supervised training, thus avoiding the need for layer-wise pre-training. Most of the DNN algorithms consist of two separate phases for training, namely unsupervised layer-wise pre-training and supervised fine-tuning, which is not efficient in terms of computational complexity. Avoiding the need for layer-wise pre-training, these two phases can be combined into one, so-called semi-supervised learning, to minimize the sum of supervised and unsupervised cost functions. In addition to being computationally efficient, semi-supervised learning can improve generalization performance using unlabeled data.

3.3.1 Pseudo-label

Pseudo-Label [46] is an efficient method for training DNNs in a semi-supervised fashion. In this approach, for every weight update, the class which has the maximum predicted probability is selected as the label for unlabeled data. Then the deep network is trained with labeled and unlabeled data simultaneously in a supervised way.

Because of its simplicity and efficiency, Pseudo-Label could combine almost all neural network models and training methods. It is equivalent to Entropy Regularization that aims at creating a decision boundary in low-density regions by minimizing the conditional entropy of class probabilities for unlabeled data. As shown in Fig. 2, in the Pseudo-Label algorithm, we first train the model on a batch of labeled data to calculate the labeled loss. Then we employ the trained model to select the class with the maximum predicted probability as the predicted label for unlabeled data. These pseudo labels are used to calculate the unlabeled loss. Finally, we train the DNN with labeled and unlabeled data simultaneously in a supervised fashion. To be more precise, we combine the labeled loss with unlabeled loss and backpropagate the error to update the weights.

Let x be an unlabeled sample of an initial dataset T, the pseudo-label of x (y'_i) is calculated by picking up the class with maximum probability as follows [46]:

$$y'_i = \begin{cases} 1 & \text{if } i = \arg\max_{i'} p_{i'}(x) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The objective function that should be minimized in the fine-tuning stage is defined as:

$$L = \frac{1}{n}\sum_{m=1}^{n}\sum_{i=1}^{C} L(y_i^m, p_i^m) + \alpha(t)\frac{1}{n'}\sum_{m=1}^{n'}\sum_{i=1}^{C} L(y'^m_i, p'^m_i), \tag{5}$$

where n and n' are the number of labeled and unlabeled samples of each mini-batch in Gradient Descent, respectively. C is the number of classes. p_i^m and y_i^m are the output units and the label of sample m in labeled data, respectively, and p'^m_i and y'^m_i are the output units and the pseudo-label of sample m in unlabeled data, respectively. To prevent the optimization process from getting stuck in poor local minima, $\alpha(t)$ is gradually increasing using a deterministic annealing procedure [46].

$$\alpha(t) = \begin{cases} 0 & t \le T_1 \\ \dfrac{t - T_1}{T_2 - T_1}\alpha_f & T_1 \le t \le T_2 \\ \alpha_f & T_2 \le t \end{cases} \tag{6}$$

4 Threat Model

In this section, we discuss possible threats that an adversary can create against the proposed framework and how our framework can tackle these potential threats. One of the dominant ways attackers evade Android malware detection systems is repackaging. The malicious application can use secondary packaging to alter the source code or attach some malicious payload to the original app. Obfuscation and reflection are other methods to conceal malicious behavior in the source code. In our hybrid malware detection approach, we run the Android apps using CopperDroid emulator; so, any malicious behavior of the application that has been hidden using repackaging, obfuscation, or reflection techniques would be revealed. Since CopperDroid's reconstruction mechanism is independent of the underlying action invocation methods, it can reconstruct system calls initiated from both DEX bytecode or C/C++ native code. Therefore, it can capture all semantics from both OS and Dalvik

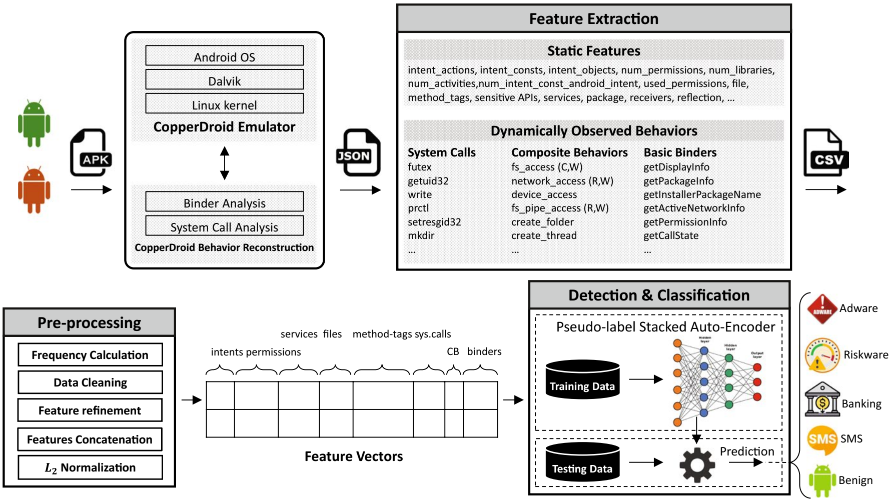


Fig. 3 Proposed framework for hybrid Android malware semi-supervised classification

viewpoints and detect every possible execution path of an Android app. The proposed framework also extracts dynamic behaviors such as binder calls that describe complex Intra- and Inter-Process Communications (IPC). This makes it resilient against multiple privilege escalation exploits.

Being based on a semi-supervised deep learning model, the framework is also effective in combating zero-day and unknown Android malware. As new malware samples arrive in batches, we do not need to engineer the features from scratch. The framework can automatically extract features from the new samples and incrementally update the deep learning model. Furthermore, due to its semi-supervised nature, the framework can be trained using a substantial number of unknown malware and a few labeled samples that match the real-world scenario. Due to the high efficiency of the proposed framework, we can run it as a detection module on mobile devices that identifies the category of malware. It can work as a proactive and preventive tool to help us mitigate the malware threat and take precautionary steps to prevent catastrophic abuse to the device.

5 Proposed Framework

In this section, we detail the steps followed to detect and classify Android apps into five categories using a hybrid method, as shown in Fig. 3.

5.1 Data Collection

We managed to collect more than 17,341 Android samples from several sources including VirusTotal service, Contagio security blog [50], AMD [6], and other datasets used by recent research contributions [10, 51, 52]. The samples were collected from December 2017 to December 2018. It is significant for cybersecurity researchers to classify Android apps with respect to the malware category to consider proper countermeasures and mitigation strategies. Hence, our dataset intentionally spans between five distinct categories: Adware, Banking malware, SMS malware, Riskware, and Benign. The five categories of Android malware applications encompass a broad range of Android malware types. This taxonomy is very comprehensive and almost include all avenues an Android malware might use to gain financial profit, namely product payment, ransom payment, fraud SMS charge, money transfer, and data theft [30]. The five categories include up-to-date Android malware threats namely, Command and Control (C&C) malware (botnets), Trojans, SMS phishing, fraud, scareware, ransomware, and adware. To foster more research in this area, we released the accumulated dataset (CICMalDroid2020) to the research community [2]. Each malware category is briefly described as follows:

- **Adware.** Mobile Adware refers to an unwanted program designed to pop up advertisements (ad) on the screen that typically hides inside a legitimate application to trick the user into installing it into her/his mobile device. Adware can make profits for its developers by repeatedly displaying ads on the user's screen even if the victim tries to force-close the application. Adware infects victim's device, traces the locations have been visited by the victim over the Internet, and presents ads related to his/her viewing habits. It can also sells the browsing behavior and personal information to third parties to target the victim with more customized ads. In our dataset, the Adware category consists of the following families: Adwo, Andup, Dowgin, Kemoge, Kuguo, Minimob, Mobidash, Shuanet, Utchi, and Youmi.
- **Banking Malware.** Mobile Banking malware is a specialized malware designed to gain access to the user's online banking accounts by mimicking the original banking applications or banking web interface. Most of the mobile Banking malware are Trojan-based, which is designed to infiltrate devices, to steal sensitive details, i.e., bank login and password, and to send the stolen information to a C&C server [51]. The malware families of this category in our dataset are as follows: Bankbot, Binv, Bankun, Citmo, Fakebank, Sandroid, Spitmo, SlemBunk, Svpeng, Wroba, ZertSecurity, and Zitmo.
- **SMS Malware.** SMS malware exploits the SMS service as its medium of operation to intercept SMS payload for conducting attacks. The attackers first upload malware to their hosting sites to be linked with the SMS. They use the C&C server for controlling their attack instructions, i.e., send malicious SMS, intercept SMS, and steal data. The SMS malware category includes the following families: Boxer, FakeInst, Gumen, Leech, RuMMS, SMSReg, SMSpay, SMSSend, SMSspy, and SMSkey.
- **Mobile Riskware.** Riskware refers to legitimate programs that have the potential to become a threat to a host device due to vulnerabilities or software incompatibility. Consequently, they can be turned into any other form of malware such

[2] https://www.unb.ca/cic/datasets/maldroid-2020.html

as Adware or Ransomware, which extends functionalities by installing newly infected applications. Uniquely, this category only has a single variant, mostly labeled as “Riskware” by VirusTotal.
- **Benign.** All other applications that are not in categories above are considered benign which means that the application is not malicious. For the sake of verification, we scanned all the benign samples with VirusTotal [53].

5.2 Data Analysis

CopperDroid aims at reconstructing system call semantics as precisely as possible regardless of whether they were initiated from Java or native code to demonstrate that all Android application behaviors manifest themselves through system calls [54]. We statically and dynamically analyzed our collected data using CopperDroid [12], a VMI-based analysis system, to automatically reconstruct low-level OS-specific and high-level Android-specific behaviors of Android samples. Furthermore, Copperdroid is able to extract a broad range of static features from Android samples. Out of 17,341 samples, 13,077 samples ran successfully while the rest failed due to errors such as time-out errors or others including bad UTF-8 bytes, problems installing the app, invalid APK files, list index out of range, bad unpack parameters, bad ASCII characters, bad CRC-32 values, and memory allocation failures. The dynamic analysis module observes the behavior of Android apps while they are being executed in a simulated environment. The information captured during runtime could demonstrate the underlying characteristics of the Android sample and reveal the sample’s intentions. Using dynamic analysis, we are capable of dissecting and comprehending every aspect of malware that result in a stable detection process. On the other hand, static analysis concentrates on disassembly and decompilation of the code, essentially Android manifest file. All the APK files are first executed in CopperDroid and the runtime behaviors are recorded in log files. The output analysis results of CopperDroid are available in JSON format for easy parsing and additional auxiliary information. They include (1) statically extracted information, e.g, intents, permissions and services, frequency counts for different file types, incidents of obfuscation, method tags, and sensitive API invocations and (2) dynamically observed behaviors which are largely broken down into three categories of system calls, binder calls, composite behaviors, and the PCAP of all the network traffic captured during the analysis [12]. In Appendix A, we provide the snippets of static features and dynamic behaviors of an APK file that are obtained from the CopperDroid analysis in the JSON format.

5.3 Feature Extraction

Hybrid analysis of Android applications benefits from the advantages of both static and dynamic analysis, therefore yield higher performance. In this paper, we used both statically and dynamically extracted information from CopperDroid to increase detection accuracy of PLDNN [11]. For the static features, we extract 179 items of high-level static information from the captured JSON log files related to

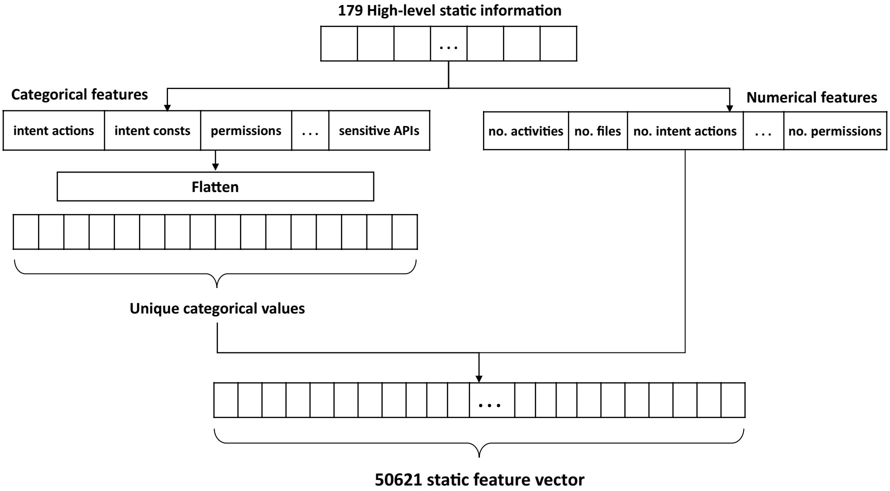


Fig. 4 Static feature vector construction

each Android APK. The high-level static information is categorized into categorical and numerical feature types. We then pull out all possible values of the categorical features from all APK files, such as intent actions, permissions, intent consts, permissions, file, method tags, sensitive APIs, services, package, receivers and construct the unique categorical values of all APK files as a feature vector. The result is concatenated with the numerical features including number of activities, number of files, number of intent actions, number of libraries, number of permissions, number of providers, number of receivers, and number of services resulting in a final feature vector with a length of 50,621. Fig. 4 elaborates the stages of constructing our static feature vector. Regarding dynamic features, we combine high resilient system calls with binder calls and composite behaviors from the captured log files to acquire valuable information from the malware behavior. By putting all distinct low-level behaviors of all APK files together, we create feature vectors of size 470 for each APK file. The dynamic features are similar to the ones used in our previous contribution [11]. Binder is a base class used for realizing an optimized IPC and lightweight Remote Procedure Call (RPC) mechanism. Composite behaviors such as *fs_access(create, write)*, *network_access(read, write)*, and *fs_pipe_access(read, write)* aggregate commonly associated low-level system calls. We provide samples of binder calls, namely *getDisplayInfo, registerCallback, getProxy* and composite behavior *network_access(read, write)* that are obtained from the CopperDroid analysis in the JSON format in Appendix A.

5.4 Pre-processing

In the pre-processing stage, we first calculate the frequencies of the distinct static information and dynamic behaviors. Then we clean the data by replacing NaN values with zero. We also encode categorical feature values using Pandas in Python and replace boolean feature values, i.e., True and False with one and zero, respectively. Since we have a very sparse feature matrix, we apply the Variance Threshold algorithm from SKlearn library [55] to remove all low-variance features. In other words, features with a training-set variance lower than a specified threshold will be removed. As a result of this refinement operation with variance threshold 0.1, static features length is shrunk from 50,621 to 302 and the dynamic feature vector is shrunk from 470 to 262. We then concatenate the static and dynamic feature vectors into a final feature vector of length 564. We finally normalize the feature vectors into values between [0, 1] using ℓ_2 normalization method which scales each feature vector such that the square root of the sum of squares of all the values, i.e., vector's ℓ_2 -norm, equals one.

Let $x = (x_1, x_2, \cdots, x_n)$ be a vector in the n-dimensional real vector space $\mathbb{R}^n$, the ℓ_2-norm of vector x, denoted by $|x|$, is defined as:

$$|x| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}. \tag{7}$$

To point out the most influential features for detecting a malicious Android application, we rank the top-40 dynamic behaviors using Mutual Information. As illustrated, binder *getInstallerPackageName* and system call *newselect* have the highest score with a meaningful distance from the subsequent features. Most of the malicious activities of an application are manifested through file and network management system calls [56]. File management system calls can be exploited by malware to communicate with file descriptors related to sensitive resources in an OS. Also, malware can use network-related system calls to communicate with C&C servers for establishing malicious activity. Hence, these types of system calls contain precious information about the malicious behavior of that application. Another important source of features are those related to binder transactions, i.e., IPC operations that actually transfer data and are also responsible for RPC [12]. Malware can employ binder calls to escalate the privileges of a process and gain the root access to an Android system. All Binder transactions are fired by invoking *ioctl* system calls sent from the client application to the IActivityManager. As can be seen from Table 2, the vast majority of the learned features are related to file and network management system calls and composite behaviors that aggregate commonly associated low-level system calls. For instance, *newselect, create_folder, mkdir, pwrite64, pread64, unlink, fdatasync, fs_pipe_access, fs_access(read), fs_pipe_access(write)*, and *fs_access* system calls, which are amongst the top twenty features, are required for every application for communicating with sensitive resources in a system [56]. Besides, *network_access* composite behavior can be used by malware to communicate with a C&C server [56] to obtain encryption keys in ransomware or send/receive SMS message to/from premium services owned by a malware author [57, 58]. There are also a small handful of system calls related to memory management (*mmap2* and *mummap*) and process management (*mprotect*, *nanosleep*, and *getuid*) that though selected as being significant, might not provide valuable information for usual malicious activities of malware. All applications could use these system calls for handling normal operations. A noticeable number of system calls are also

Table 2 Top-40 extracted features using Mutual Information algorithm

Rank	Feature	Merit	Type	Rank	Feature	Merit	Type
1	getInstallerPackageName	0.272	binder	21	fs_access()	0.138	file
2	newselect	0.263	file	22	pipe	0.136	file
3	mprotect	0.187	process	23	fcntl64	0.136	file
4	getPackageInfo	0.183	binder	24	fs_access(create_write)	0.134	file
5	create_folder	0.181	file	25	nanosleep	0.133	process
6	getReceiverInfo	0.172	binder	26	getApplicationInfo	0.131	binder
7	mkdir	0.163	file	27	fs_access(create_read_write)	0.131	file
8	pwrite64	0.164	file	28	rename	0.13	file
9	getServiceInfo	0.158	binder	29	access	0.131	file
10	pread64	0.158	file	30	chmod	0.129	file
11	unlink	0.155	file	31	gettid	0.128	process
12	munmap	0.15	memory	32	fs_access(create)	0.128	file
13	statfs64	0.146	file	33	brk	0.127	memory
14	getActivityInfo	0.146	binder	34	device_access	0.127	input/output
15	fdatasync	0.146	file	35	queryIntentServices	0.126	binder
16	fs_pipe_access	0.143	file	36	network_access	0.125	network
17	fs_access(read)	0.142	file	37	stat64	0.125	file
18	fs_pipe_access(write)	0.14	file	38	fs_access(write)	0.125	file
19	fs_access	0.139	file	39	open	0.125	file
20	mmap2	0.139	memory	40	lstat64	0.124	file

binders including *getInstallerPackageName, getPackageInfo, getReceiverInfo, getServiceInfo, getActivityInfo, getApplicationInfo,* and *queryIntentServices*.

5.5 Detection and Classification

The first step in this module is training the PLSAE with the provided normalized feature vectors. To train PLSAE, first we need to pre-train all layers by encoding x into a code and then decoding the code into x'. The network is trained by minimizing the reconstruction error, and then the weights and biases are frozen. In the fine-tuning step after the network is pre-trained, the decoder part is ignored, a Softmax classifier is put at the top of the network, and the whole network working as a typical FFNN is trained with the Pseudo-Label algorithm. Using Pseudo-Label, the feed forward network is trained in a supervised fashion with labeled and unlabeled data simultaneously. Pseudo-labels, which are recalculated with every weight update, are used along with the real labels to optimize the supervised loss function in each mini-batch. Fig. 5 illustrates the architecture of the PLSAE. The input layer has 564 neurons representing the total number of static

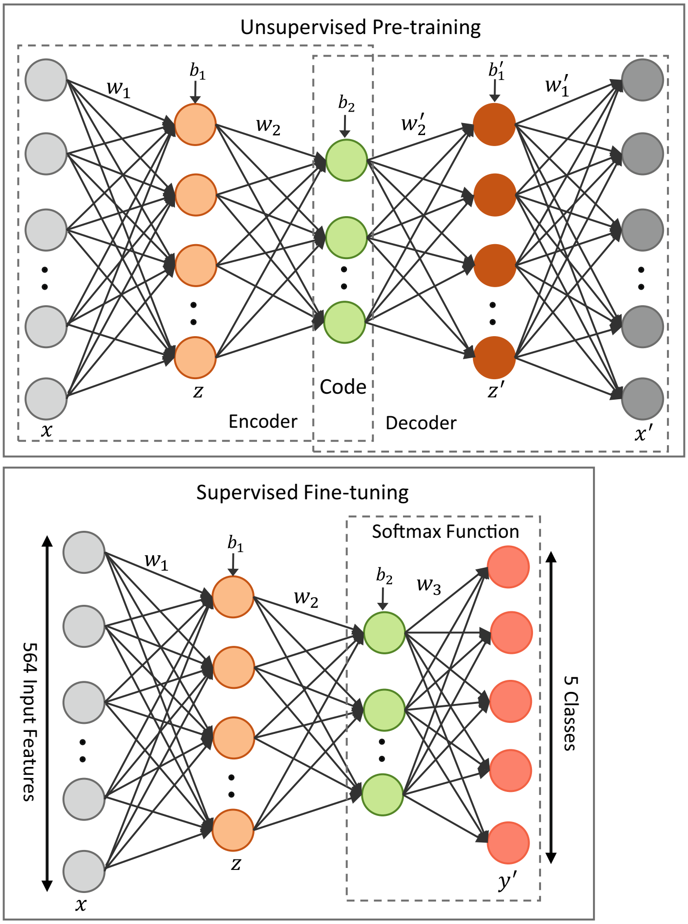


Fig. 5 Greedy layer-wise training of PLSAE

and dynamic features whereas the output layer consists of five neurons representing the number of classes in our classification task.

For the prediction step, the trained model is fed the test data, and then classifies the input feature vector into one of the five categories. Finally, the corresponding classification measures are calculated and reported as an output.

6 Performance Analysis

In this section, we evaluate how *effectively* and *efficiently* the semi-supervised deep learning-based framework can classify Android APK files into one of the pre-defined categories.

6.1 Implementation

We implemented the proposed semi-supervised deep SAE framework using a PC with 3.60 GHz Core i7-4790 CPU and 32 GB RAM. We implemented our code using Tensorflow in Python which is a major modification of our previous contribution [11] based on Pseudo-Label using FFNN. We added an implementation of a PLSAE with unsupervised pre-training and supervised fine-tuning. We also changed the pre-processing parts related to analyzing the static features, numericalizing the data, removing the features with low variance that carry a little information, features concatenation, and normalization. The input layer of the DNN consists of 564 neurons, with reference to the number of input feature vectors as detailed in Sect. 5, and the output layer consists of five neurons equivalent to the number of categories. The Deep PLSAE includes an encoder and decoder section whose hidden layers and hidden neurons need to be fine-tuned. We used the Sigmoid function as the activation function for all hidden layers in encoder and decoder and the mini-batch RMSPropOptimizer as the optimization algorithm. All the weights are initialized using normal distribution with zero mean and standard deviation of one. The mini-batch size was set to 100 for labeled samples, and the learning rate was set to 0.007. These parameters were determined using hyper-parameter tuning. As mentioned previously in Sect. 3, the balancing coefficient, $\alpha(t)$ was slowly increased to dynamically adjust the network performance with the parameter settings of $\alpha_f = 1.5$, $T_1 = 100$, and $T_2 = 400$ (We refer the reader to [46] for more details about these parameters). We trained each PLSAE model until convergence or the number of 1500 epochs was reached. We ran each set of experiments 20 times and computed the average over all runs to find the evaluation metrics. To make sure that the training samples in each mini-batch are randomly selected, we shuffled the entire dataset at the beginning of each epoch.

6.2 Dataset

As discussed earlier in Sect. 5, we ran 17,341 APKs in CopperDroid to observe the behavior of each sample. Out of this number of samples, 13,077 ran successfully while the rest failed because of different error types including but not limited to bad unpack parameters, bad ASCII characters, bad CRC-32 values, and bad UTF-8 bytes. Then we loaded all analysis results where about 12% of the JSON files failed to open mostly due to "unterminated string". The final remaining Android samples in each category are as follows: Adware (1,253), Banking (2,100), SMS malware (3,904), Riskware (2,546), and Benign (1,795).

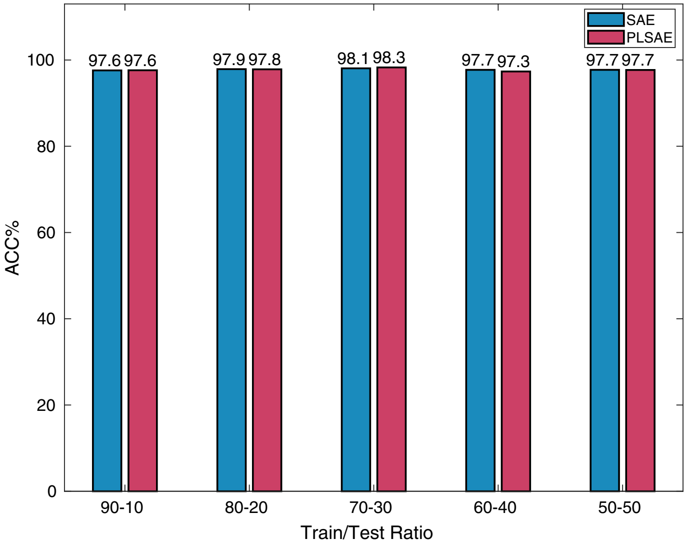


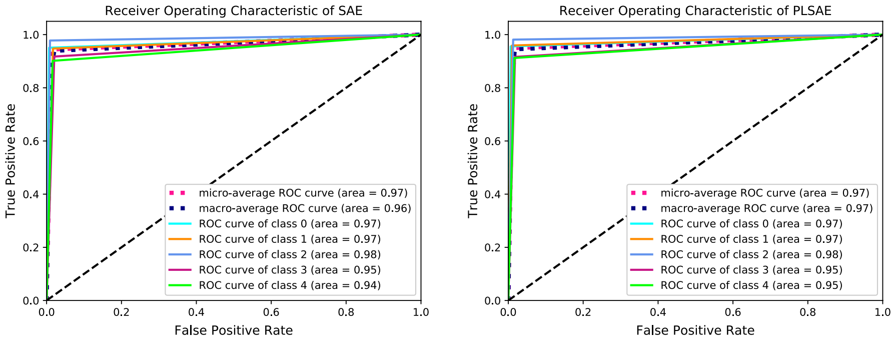
Fig. 6 ACC of SAE and PLSAE for different train/test split ratios

6.3 Experimental Results

In our experiments, we used Precision (PR), Recall (RC), F_1-Score (F_1), Accuracy (ACC), False Positive Rate (FPR), False Negative Rate (FNR), and classification error to assess the overall classification performance. We examined the classification performance of our proposed framework under different train/test split ratios. Afterward, we picked the best architecture and train/test split ratio (70% – 30%) and compared the effectiveness of our approach with other state-of-the-art machine learning and deep learning algorithms while changing the number of labeled samples. We plotted the ROC curves per class, the micro/macro average ROC curves, and computed the confusion matrix, as well. Furthermore, we compared the performance of the semi-supervised deep SAE with the supervised deep SAE, PLDNN, DNN, some common machine learning algorithms including the semi-supervised approach (LP) while changing the number of labeled samples. Finally, we estimated the average runtime of the feature extraction, pre-processing, and detection stages.

6.3.1 Malware Classification Performance

We investigated the impact of hidden layers and hidden neurons of the PLSAE on the classification performance of our malware detector. For this set of experiments, we set the number of labeled samples and train/test ratio to 1000 and (70% – 30%), respectively. The fine-tuning of PLSAE architecture demonstrated that the deep network with four hidden layers and neurons of [564 450 300 150 10 5] is superior to other deep networks, and from this point on, it was used as the deep network architecture for the subsequent experiments. In the experiments, SAE is the deep Stacked Auto-Encoder model trained without unlabeled data (supervised) whereas PLSAE is



(a) SAE **(b)** PLSAE

Fig. 7 Receiver Operating Characteristic (ROC). (Class 0 is Adware, class 1 is Banking, class 2 is SMS Malware, class 3 is Riskware, and class 4 is Benign)

the one trained with both labeled and unlabeled data (semi-supervised) considering the same dataset. Additionally, PLDNN is the semi-supervised DNN that is trained on the dynamic features [11].

Fig. 6 visualizes accuracy of SAE (supervised) and PLSAE (semi-supervised) for different train/test split ratios of $\%(90-10, 80-20, 70-30, 60-40, 50-50)$. As shown in Fig. 6, the classification performance under the 70-30 train/test percentage outperforms other sets for both SAE and PLSAE. However, the classification results are too close under all split percentages. This shows how robust our trained deep network is because it performs excellently in all train/test split percentages. Moreover, in all settings, the accuracy of PLSAE and SAE is almost similar, which demonstrates that the proposed semi-supervised framework (PLSAE) in this paper with unlabeled data can be used instead of the supervised one (SAE) that needs labeled data. This is really helpful because the labeling process is expensive when handling such a real-world problem where some pieces of malware can avoid detection.

Receiver Operating Characteristic (ROC) curve plots True Positive Rate (TPR) vs FPR which shows the overall quality of the classifier's output. Therefore, the top left corner of the plot is the optimum point which leads to the maximum Area Under Curve (AUC). Fig. 7 juxtaposes ROC curves of each class, micro average, and macro average. Micro averaging considers the metrics globally whereas macro averaging calculates metrics per label and finds their unweighted mean. As shown in both figures, the micro and macro averages AUC for both PLSAE and SAE are close to one, i.e., 0.96-0.97 . All the plots are clustered in the ideal area at the top left corner where the AUC of class 2 (SMS malware) is a bit higher (0.98) than those of other ROC curves. This means that our detection system could distinguish SMS malware slightly better than other classes, however the detection performance of the rest of the classes is excellent regarding the high degree of similarity between the categories of Adware, Riskware, and Benign. Overall, the ROC curves demonstrate that PLSAE precisely discovers the underlying behavior of the Android malware with only 1000 labeled training samples. This shows that regardless of the small number of labeled samples, our semi-supervised approach exhibits highly

Table 3 Confusion matrix of category classification

		Prediction Category				
		Adware	Banking	SMS	Riskware	Benign
Real category	Adware	0.96	0.01	0.01	0.01	0.02
	Banking	0.02	0.96	0.0	0.01	0.01
	SMS	0.0	0.01	0.99	0.0	0.0
	Riskware	0.0	0.01	0.01	0.95	0.03
	Benign	0.0	0.0	0.01	0.04	0.95

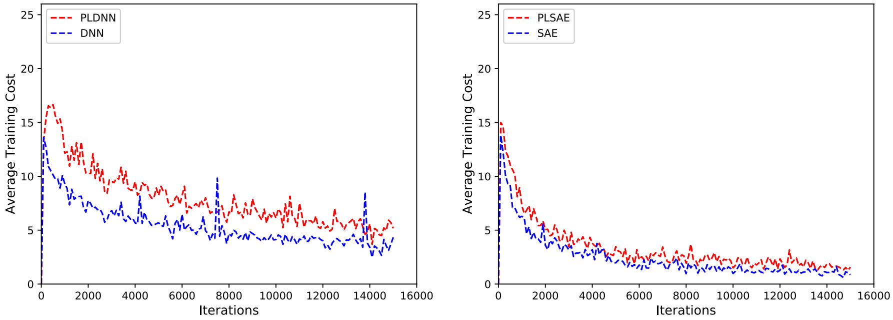
Table 4 Classification error (%) on the Android malware dataset with 100, 300, 500, 1000, 5000, and all labeled training samples. DNN and PLDNN are applied on dynamic features [11]

Method	Labeled training samples					
	100	300	500	1000	5000	All
RF	14.34	7.38	6.58	5.08	2.09	1.68
SVM	68.16	65.78	65.65	65.59	65.24	65.17
K-NN	30.47	20.54	17.36	13.33	7.23	5.60
LP	9.74	9.5	9.05	8.82	9.56	6.89
DNN_D	10.82	9.44	7.93	3.52	2.77	2.4
PLDNN_D	8.37	6.93	4.54	3.3	2.6	2.4
SAE	6.65	3.41	2.55	1.93	1.64	1.52
PLSAE	4.81	3.23	2.54	1.72	1.61	1.48

competitive performance for malware categorization, which is even higher than the supervised SAE.

Table 3 shows the confusion matrix of PLSAE for each Android app category. We have set the number of labeled training samples to 1000. We listed TPR, i.e., the fraction of predicted apps with respect to the total number of apps for each category. The diagonal illustrates the correct classification. SMS malware achieves the highest TPR of 99% whereas Riskware and Benign both achieve the lowest TPR among other categories (95%). Riskware and Benign have been mostly mistaken for each other, 3% and 4%, respectively, due to inherent similarity and common characteristics they share. As shown, PLSAE could remarkably improve the TPR of Adware samples (0.96) in comparison with PLDNN (0.85) [11].

Table 4 compares the classification error of PLSAE with that of basic SAE, PLDNN [11], DNN, semi-supervised approach LP, and four common machine learning classifiers, namely RF, SVM, and k-NN on the Android malware dataset with 100, 300, 500, 1000, 5000, and all labeled training samples. We applied all the machine learning methods on the hybrid feature vector used with PLSAE and SAE except for PLDNN and DNN. The results of PLDNN and DNN are both from our previous research contribution [11] where the features were merely dynamic. We applied stratified 5-fold cross-validation for all machine learning algorithms. The results presented in the table show that by increasing the number of labeled



(a) PLDNN and DNN

(b) SAE and PLSAE

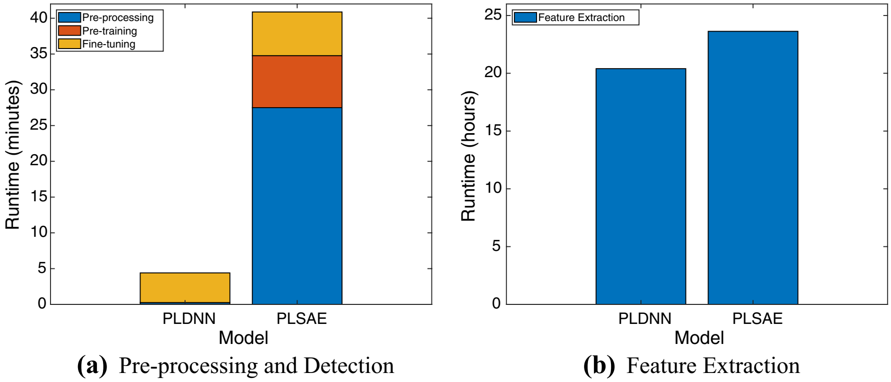
Fig. 8 Average training cost vs total number of iterations

samples the classification error of all algorithms decreases. For any number of labeled training samples, PLSAE (semi-supervised hybrid) prominently outperforms LP (semi-supervised hybrid). The classification error of PLSAE is 1.6% less in comparison with the error of PLDNN (semi-supervised dynamic) for 1000 labeled training samples. For the small number of labeled samples, the superiority of PLSAE over PLDNN becomes more clear.

For instance, for approximately 1% of the total number of labeled training samples (100 labeled samples), PLSAE achieves an ACC of 95.19% for malware categorization, justifying its stability even in the presence of scarce labeled data. In contrast, PLDNN obtains an ACC of 91.63% for 100 labeled samples. For almost all labeled samples, e.g., 300-5000, the gap between the classification error of SAE and PLSAE is almost negligible. For instance, for samples=5000, the error is decreased to 1.64% and 1.61% for SAE and PLSAE, respectively.

In Fig. 8, the average training cost of PLDNN and PLSAE are compared according to the number of iterations.

The number of iterations is calculated by multiplying the number of epochs (1500) and the number of labeled mini-batch (10), i.e., 15,000. As expected, the average cost of both methods constantly decreases as the number of iterations increases. However, the first iterations have massive improvements, but after a while, the cost slightly changes and is stabilized. As shown in both plots, the average cost of PLSAE is much lower than PLDNN, which can be justified by the higher detection capability and effectiveness of our hybrid semi-supervised deep AE compared with PLDNN. Besides, the average costs of training semi-supervised deep networks are higher than the ones of the supervised deep networks. The reason for this is that we expand the loss function of labeled samples by adding $(\alpha(t)\frac{1}{n'}\sum_{m=1}^{n'}\sum_{i=1}^{C}L(y'^{m}_{i}, p'^{m}_{i}))$ term to the overall loss function L in Eq. 5. As we proceed with training, this gap gradually shrinks, and the two diagrams approximately converge after 15,000 iterations which demonstrates that our semi-supervised DNN with many unlabeled and small amount of labeled samples could easily substitute all-labeled scenarios.



(a) Pre-processing and Detection **(b)** Feature Extraction

Fig. 9 Comparison of runtime performance for different steps of PLSAE and PLDNN

6.3.2 Runtime Performance

In this section, we evaluated the efficiency of PLSAE by estimating the detection, pre-processing, and feature extraction time. The detection time is the total time needed to predict the category of the test samples, including pre-training and fine-tuning stages. For the runtime performance, we set the number of labeled samples to 1000 and the batch size to 100. As presented in Fig. 9(a) and (b), the total time needed for feature extraction (23.63 h) and pre-processing (27.5 min) of PLSAE is approximately 3.7 h larger than PLDNN (20.4 h and 0.25 min) because of the huge size of initial static information added to the dynamic features. However, in estimating runtime performance, we are concerned about detection time rather than pre-processing and feature extraction stages. The total detection time of PLSAE is 13.38 min (including pre-training and fine-tuning) for the whole test data (3480) compared with PLDNN (4.15 min). Consequently, the average prediction time per sample for PLSAE is 230 ms as compared with 60 ms for PLDNN. This low amount of detection time is negligible considering the noticeable improvement we achieve with hybrid PLSAE versus PLDNN. Therefore, our proposed Android malware detection system is very efficient and could be easily deployed in computationally-limited devices.

7 Conclusion

In this paper, we have proposed an effective and efficient Android malware category classification approach based on semi-supervised DNNs. It is a hybrid approach that integrates both static and dynamic analysis of malware to utilize the strengths of both of them. The pre-training scheme of the deep network using SAE helps in better generalization. As a result, in spite of the small number of labeled training samples, the proposed detection approach is effective and superior in comparison to other state-of-the-art techniques. This eliminates the need for a high number of labeled instances, which is very expensive to acquire in the domain of malware analysis. Additionally, it is efficient in terms of execution time, and it helps us to prioritize our mitigation techniques by specifying the category of the malware. We have offered a new 17,341 Android malware dataset which includes the most complete captured static and dynamic feature sets and spans between five distinct categories of malware. As future work, we are planning to test if the proposed detection system can run on resource-limited IoT devices such as Raspberry Pi. Our approach also could be enhanced using advanced deep learning models like RNN or CNN. Additionally, we are planning to keep updating CICMalDroid2020 by including new samples together with their analysis and to study how increasing the number of samples will affect the experimental results.

Appendix

Listing 1 Static Features

```
{
  "behaviors": {
    "static": {
      "intent_actions": [
        "android.intent.action.MAIN",
        "android.intent.action.BOOT_COMPLETED",
        "android.intent.action.PHONE_STATE"
      ],
      "num_permissions": 10,
      "intent_consts": [
        "'android.intent.action.VIEW' ",
        "'android.intent.action.VIEW' ",
        "'android.intent.action.VIEW' ",
        "'android.intent.action.VIEW' ",
        "'android.intent.extra.shortcut.INTENT' ",
        "'android.intent.extra.shortcut.NAME' ",
        "'android.intent.extra.shortcut.ICON_RESOURCE' ",
        "'android.intent.action.VIEW' ",
        "'android.intent.action.VIEW' "
      ],
      "num_intent_const_android_intent": 6,
      "used_permissions": {
        "android.permission.VIBRATE": 2,
        "android.permission.INTERNET": 8,
        "android.permission.READ_PHONE_STATE": 1
      },
      "file": {
        "DEX": 1,
        "data": 10,
        "ASCII": 3,
        "UTF-8": 1,
        "PNG": 4
      },
      "method_tags": {
        "CONTENT": 21,
        "WIDGET": 4,
        "TEXT": 1,
        "APP": 6,
        "JAVA_REFLECTION": 2,
        "UTIL": 1,
        "TELEPHONY": 3,
        "PROVIDER": 1,
        "ANDROID": 33,
        "OS": 6,
        "NET": 7,
        "SMSMESSAGE": 1,
        "VIEW": 1
      },
```

Listing 2 Dynamic Behaviors

```
{
  "class": "SYSCALL",
  "low": [
    {
      "sysname": "clock_gettime",
      "type": "SYSCALL",
      "id": 160,
      "parameters": [
        1,
        "'\\x96\\x00\\x00\\x00Aq\\xb6\\x03\\xa4\\xfe\\xff\\xffC\\x93\\xa7\\xb5'"
      ],
      "ts": "1526579389.751"
    }
  ]
},
{
  "class": "SYSCALL",
  "low": [
    {
      "sysname": "mmap2",
      "type": "SYSCALL",
      "id": 161,
      "parameters": [
        "''",
        1040384,
        3,
        16418,
        -1,
        0
      ],
      "ts": "1526579389.751"
    }
  ]
},
{
  "class": "SYSCALL",
  "low": [
    {
      "sysname": "madvise",
      "type": "SYSCALL",
      "id": 162,
      "parameters": [
        "''",
        1040384,
        12
      ],
      "ts": "1526579389.751"
    }
  ]
}
```

Listing 3 Binder Calls

```
 {
  "method": "getDisplayInfo",
  "interfaceGroup": "android.hardware.display",
  "low": [
    {
      "methodName": "android.hardware.display.IDisplayManager.getDisplayInfo(displayId =
            0)",
      "type": "BINDER",
      "id": 414,
      "ts": "1526579390.815"
    }
  ],
  "interface": "android.hardware.display.IDisplayManager",
  "class": "BINDER",
  "arguments": [
    "displayId = 0"
  ]
},
{
  "method": "registerCallback",
  "interfaceGroup": "android.hardware.display",
  "low": [
    {
      "methodName": "android.hardware.display.IDisplayManager.registerCallback(callback
           = [IDisplayManagerCallback N/A])",
      "type": "BINDER",
      "id": 416,
      "ts": "1526579390.816"
    }
  ],
  "interface": "android.hardware.display.IDisplayManager",
  "class": "BINDER",
  "arguments": [
    "callback = [IDisplayManagerCallback N/A]"
  ]
},
{
  "method": "getProxy",
  "interfaceGroup": "android.net",
  "low": [
    {
      "methodName": "android.net.IConnectivityManager.getProxy()",
      "type": "BINDER",
      "id": 483,
      "ts": "1526579390.818"
    }
  ],
  "interface": "android.net.IConnectivityManager",
  "class": "BINDER",
  "arguments": []
},
```

Listing 4 Composite Behaviors

```
{
  "classType": 1,
  "operationFlags": 3,
  "procname": "com.cheaptravelnetwork.cheapflights",
  "low": [
    {
      "sysname": "socket",
      "blob": "{'socket domain': 1, 'socket type': 1, 'socket protocol': 0}",
      "type": "SYSCALL",
      "id": 1505,
      "ts": "1530103413.850"
    },
    {
      "sysname": "connect",
      "xref": 1505,
      "ts": "1530103413.850",
      "blob": "{'host': '/dev/socket/dnsproxyd', 'port': 'None', 'returnValue': 0}",
      "type": "SYSCALL",
      "id": 1507
    },
    {
      "sysname": "write",
      "xref": 1505,
      "ts": "1530103413.850",
      "blob": "{'size': 55L, 'filename': 'Socket (1, 1, 0)'}",
      "type": "SYSCALL",
      "id": 1517
    },
    {
      "sysname": "read",
      "xref": 1505,
      "ts": "1530103413.852",
      "blob": "{'size': 4096L, 'filename': 'Socket (1, 1, 0)'}",
      "type": "SYSCALL",
      "id": 1518
    },
    {
      "sysname": "close",
      "xref": 1505,
      "ts": "1530103413.852",
      "blob": "{'filename': 'Socket (1, 1, 0)'}",
      "type": "SYSCALL",
      "id": 1528
    }
  ],
  "tid": 1162,
  "class": "NETWORK ACCESS(READ, WRITE)"
},
```

Acknowledgements The authors would like to express their gratitude toward Dr. Lorenzo Cavallaro and Feargus Pendlebury (Systems Security Research Lab, King's College London) for generously analyzing a large number of Android APKs in CopperDroid.

References

1. “Mobile OS market share Statista ,” https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, online; accessed 30 April 2019
2. Otoum, Y., Nayak, A.: As-ids: Anomaly and signature based ids for the internet of things. J. Netw. Syst. Manag. **29**, 07 (2021)
3. Afzal, S., Asim, M., Javed, A.R., Beg, M.O., Baker, T.: Urldeepdetect: a deep learning approach for detecting malicious urls using semantic vector models. J. Netw. Syst. Manag. **29**(3), 21 (2021). https://doi.org/10.1007/s10922-021-09587-8
4. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: DREBIN: effective and explainable detection of Android malware in your pocket. In: Network and Distributed System Security Symposium (NDSS) (2014)
5. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware Android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1105–1116 (2014)
6. Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current Android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 252–276 (2017)
7. Kang, H., Jang, J.-W., Mohaisen, A., Kim, H.K.: Detecting and classifying Android malware using static analysis along with creator information. Int. J. Distrib. Sens. N. **11**(6), 479174 (2015)
8. Kim, T., Kang, B., Rho, M., Sezer, S., Im, E.G.: A multimodal deep learning method for Android malware detection using various features. IEEE Trans. Inf. Forensics Secur. **14**(3), 773–788 (2019)
9. Hou, S., Saas, A., Ye, Y., Chen, L.: DroidDelver: an Android malware detection system using Deep Belief Network based on API call blocks. In: International Conference on Web-age Information Management. Springer, pp. 54–66 (2016)
10. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: MalDozer: automatic framework for Android malware detection using deep learning. Digit. Invest. **24**, S48–S59 (2018)
11. Mahdavifar, S., Abdul Kadir, A.F., Fatemi, R., Alhadidi, D., Ghorbani, A.A.: Dynamic android malware category classification using semi-supervised deep learning. In: 2020 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing. International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), pp. 515–522 (2020)
12. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of Android malware behaviors. In: Network and Distributed System Security Symposium (NDSS) (2015)
13. Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-Sec: deep learning in Android malware detection. In: ACM SIGCOMM Comput. Commun. Rev., vol. 44, no. 4. ACM, pp. 371–372 (2014)
14. Su, X., Zhang, D., Li, W., Zhao, K.: A deep learning approach to Android malware feature learning and detection. In: Trustcom/BigDataSE/ISPA, 2016 IEEE. IEEE, pp. 244–251 (2016)
15. Nix, R., Zhang, J.: Classification of Android apps and malware using deep neural networks. IEEE International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1871–1878 (2017)
16. Hsien-De Huang, T., Kao, H.-Y.: R2-d2: color-inspired Convolutional Neural Network (CNN)-based Android malware detections. In: 2018 IEEE International Conference on Big Data. IEEE, pp. 2633–2642 (2018)
17. Wang, W., Zhao, M., Wang, J.: Effective Android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Amb. Intel. Hum. Comp. **10**(8), 3035–3043 (2018)
18. Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K.: Android malware detection based on system call sequences and LSTM. Multimed. Tools Appl. **78**(4), 3979–3999 (2019)
19. Yen, Y.-S., Sun, H.-M.: An Android mutation malware detection based on deep learning using visualization of importance from codes. Microelectron. Reliab. **93**, 109–114 (2019)
20. Lu, T., Du, Y., Ouyang, L., Chen, Q., Wang, X.: Android malware detection based on a hybrid deep learning model. In: Secur. Commun. Netw., vol. 2020, pp. 1–11, 08 (2020)

21. Ma, S., Wang, S., Lo, D., Deng, R.H., Sun, C.: Active semi-supervised approach for checking app behavior against its description. In: IEEE 39th Annual Computer Software and Applications Conference, vol. 2. IEEE, pp. 179–184 (2015)
22. Chen, L., Zhang, M., Yang, C.-Y., Sahita, R.: Semi-supervised classification for dynamic Android malware detection. arXiv preprint arXiv:1704.05948 (2017)
23. Karbab, E.B., Debbabi, M., Alrabaee, S., Mouheb, D.: Dysign: dynamic fingerprinting for the automatic detection of android malware. In: Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE), pp. 1–8 (2016)
24. Alrabaee, S., Shirani, P., Wang, L., Debbabi, M.: Fossil: a resilient and efficient system for identifying foss functions in malware binaries. ACM Trans. Priv. Secur. **21**(2), 1–34 (2018)
25. Cai, H., Meng, N., Ryder, B., Yao, D.: DroidCat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. **14**(6), 1455–1470 (2018)
26. Mahdavifar, S., Ghorbani, A.A.: Application of deep learning to cybersecurity: a survey. Neurocomputing **347**, 149–176 (2019)
27. Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E.: Deep learning for computer vision: a brief review. In: Comput. Intel. Neurosc., Vol. 2018 (2018)
28. Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., Darrell, T.: Long-term recurrent convolutional networks for visual recognition and description. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2625–2634 (2015)
29. Yang, W., Liu, Q., Wang, S., Cui, Z., Chen, X., Chen, L., Zhang, N.: Down image recognition based on deep convolutional neural network. Inf. Process. Agric. **5**(2), 246–252 (2018)
30. Fitriah Abdul Kadir, A.: A detection framework for android financial malware. Ph.D. Dissertation, University of New Brunswick (2018)
31. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
32. Collobert, R., Weston, J.: A unified architecture for natural language processing: deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning. ACM, pp. 160–167 (2008)
33. Min, S., Lee, B., Yoon, S.: Deep learning in bioinformatics. Brief Bioinform. **18**(5), 851–869 (2017)
34. Noda, K., Yamaguchi, Y., Nakadai, K., Okuno, H.G., Ogata, T.: Audio-visual speech recognition using deep learning. Appl. Intell. **42**(4), 722–737 (2015)
35. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
36. Mahdavifar, S., Ghorbani, A.A.: Dennes: deep embedded neural network expert system for detecting cyber attacks. In: Neural Computing and Applications, pp. 1–28
37. "Introduction to semi-supervised learning with ladder networks," http://rinuboney.github.io/2016/01/19/ladder-network.html/ (2016)
38. Nigam, K., Ghani, R.: Analyzing the effectiveness and applicability of co-training. Cikm **5**, 3 (2000)
39. Blum, A., Mitchell, T.: Combining labeled and unlabeled data with co-training. In: Proceedings of the 11th Annual Conference on Computational Learning Theory, ser. COLT' 98. New York, NY, USA: ACM, pp. 92–100 (1998). http://doi.acm.org/10.1145/279943.279962
40. Rosenberg, C., Hebert, M., Schneiderman, H.: Semi-supervised self-training of object detection models (2005)
41. Joachims, T.: Transductive inference for text classification using support vector machines. In: Proceedings of the 16th International Conference on Machine Learning, ser. ICML '99. San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., pp. 200–209 (1999)
42. Chapelle, O., Zien, A.: Semi-supervised classification by low density separation. In: AISTATS 2005. Max-Planck-Gesellschaft, pp. 57–64 (2005)
43. Blum, A., Lafferty, J., Rwebangira, M.R., Reddy, R.: Semi-supervised learning using randomized mincuts. In: Proceedings of the 21st International Conference on Machine Learning, ser. ICML '04. ACM, New York, NY, p. 13 (2004)
44. Zhu, X., Ghahramani, Z., Lafferty, J.: Semi-supervised learning using Gaussian fields and harmonic functions. In: Proceedings of the 20th International Conference on Machine Learning, ser. ICML'03. AAAI Press, pp. 912–919 (2003)

45. Ranzato, M.A., Szummer, M.: Semi-supervised learning of compact document representations with deep networks. In: Proceedings of the 25th International Conference on Machine Learning, ser. ICML '08. ACM, New York, NY, pp. 792–799 (2008)
46. Lee, D.-H.: Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In: Workshop on challenges in representation learning. ICML Vol. **3**, p. 2 (2013)
47. Rasmus, A., Berglund, M., Honkala, M., Valpola, H., Raiko, T.: Semi-supervised learning with ladder networks. Adv. Neural. Inf. Process. Syst. **28**, 3546–3554 (2015)
48. Sajjadi, M., Javanmardi, M., Tasdizen, T.: Regularization with stochastic transformations and perturbations for deep semi-supervised learning. CoRR, vol. abs/1606.04586 (2016)
49. Wu, W., Yu, Z., He, J.: A semi-supervised deep network embedding approach based on the neighborhood structure. Big Data Min. Anal. **2**(3), 205–216 (2019)
50. Contagio Mobile Malware Mini Dump (2019). http://contagiominidump.blogspot.ca/ online. Accessed 6 May 2019
51. Kadir, A.F.A., Stakhanova, N., Ghorbani, A.A.: An empirical analysis of Android banking malware. In: Protecting Mobile Networks and Devices: Challenges and Solutions, p. 209 (2016)
52. Abdul Kadir, A.F., Stakhanova, N., Ghorbani, A.: Android botnets: what URLs are telling us. In: Qiu, M., Xu, S., Yung, M., Zhang, H. (eds.) Network and System Security, pp. 78–91. Springer, Cham (2015)
53. Kadir, A.F.A., Stakhanova, N., Ghorbani, A.A.: Understanding Android financial malware attacks: taxonomy, characterization, and challenges. J. Cybersecur. Mobil. **7**(3), 1–52 (2018)
54. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android security. IEEE Secur. Priv. **7**(1), 50–57 (2009)
55. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
56. Surendran, R., Thomas, T., Emmanuel, S.: On existence of common malicious system call codes in android malware families. IEEE Trans. Reliab. **70**(1), 248–260 (2020)
57. Malik, S., Khatter, K.: System call analysis of android malware families. Indian J. Sci. Technol. **9**(21), 1–13 (2016)
58. Vinod, P., Zemmari, A., Conti, M.: A machine learning based approach to detect malicious android apps using discriminant system calls. Futur. Gener. Comput. Syst. **94**, 333–350 (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Samaneh Mahdavifar has received the B.Sc. and M.Sc. degrees in computer engineering-software. She received her Ph.D. degree in Computer Science from University of New Brunswick in 2021. She is an AI researcher at the Canadian Institute for Cybersecurity. Her research interests include deep learning, machine learning, trustworthy AI, cybersecurity, and privacy.

Dima Alhadidi is an Assistant Professor in the School of Computer Science at the University of Windsor. She received her Ph.D. degree in Computer Science and Software Engineering from Concordia University. Before joining the University of Windsor, she was an assistant professor at the University of New Brunswick and Zayed University, a researcher at the Canadian Institute for Cybersecurity, and a research associate at Concordia University. Her research addresses data privacy and security issues in emerging technologies such as cloud computing and healthcare.

Ali A. Ghorbani has held a variety of positions in academia for the past 37 years and is currently a Professor of Computer Science, Tier 1 Canada Research Chair in Cybersecurity, the Director of the Canadian Institute for Cybersecurity, which he established in 2016, and an IBM Canada Faculty Fellow. He served as the Dean of the Faculty of Computer Science at the University of New Brunswick from 2008 to 2017. Dr. Ghorbani is also the founding director of the laboratory for intelligence and adaptive systems research. His current research focus is Cybersecurity, Web Intelligence, and Critical Infrastructure Protection.