MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform

Ruitao Feng[1], Sen Chen[1*], Xiaofei Xie[1], Lei Ma[2], Guozhu Meng[3,4], Yang Liu[1], Shang-Wei Lin[1]
[1]Nanyang Technological University, Singapore [2]Kyushu University, Japan
[3]Institute of Information Engineering, Chinese Academy of Sciences, China
[4]School of Cyber Security, University of Chinese Academy of Sciences, China

***Abstract*—Currently, Android malware detection is mostly performed on the server side against the increasing number of Android malware. Powerful computing resource gives more exhaustive protection for Android markets than maintaining detection by a single user in many cases. However, apart from the Android apps provided by the official market (i.e., Google Play Store), apps from unofficial markets and third-party resources are always causing a serious security threat to end-users. Meanwhile, it is a time-consuming task if the app is downloaded first and then uploaded to the server side for detection because the network transmission has a lot of overhead. In addition, the uploading process also suffers from the threat of attackers. Consequently, a last line of defense on Android devices is necessary and much-needed. To address these problems, in this paper, we propose an effective Android malware detection system, MobiDroid, leveraging deep learning to provide a real-time secure and fast response environment on Android devices. Although a deep learning-based approach can be maintained on server side efficiently for detecting Android malware, deep learning models cannot be directly deployed and executed on Android devices due to various performance limitations such as computation power, memory size, and energy. Therefore, we evaluate and investigate the different performances with various feature categories, and further provide an effective solution to detect malware on Android devices. The proposed detection system on Android devices in this paper can serve as a starting point for further study of this important area.**

***Index Terms*—Android malware, Malware detection, Deep neural network, Mobile platform**

I. Introduction

With the currently increasing number of Android devices and apps, more and more Android users store personal data such as online banking and shopping in their Android devices. Consequently, the security and privacy threats on Android platform draw much attention. Undoubtedly, Android malware is one of the most important security threats in this security field [34], [35].

Therefore, how to detect Android malware becomes a severe problem. End-users expect a secure environment which is maintained by the Android markets. In other words, they consider their app sources are all trustable and secure enough. It is not surprising that the demands of Android malware detection approaches have been proposed such as signature-based approach [36]–[38], behavior-based approach [39]–[42], information-flow analysis-based approach [43]–[45]. We note that machine learning-based approach [1], [46]–[51] is one of the most promising techniques in detecting Android malware. With the available big data and hardware evolution over the past decade, deep learning has achieved tremendous success in many cutting-edge domains, including Android malware detection. Actually, all of the above solutions are under server side for Android markets. However, when a new Android malware family is reported, not all the Android markets are able to respond in a reasonable time. The current analysis workflow always follows analyzing malicious behaviors in apps, building the detection models with the generated features and then performing the detection on the entire apps. Since the number of the real-world Android apps is extremely large, e.g., there are more than 3 million Android apps on Google Play Store, it is a time-consuming task to perform the complete detection with that large number of apps. Moreover, the app from unofficial markets and third-party resources like XDA[1] are more vulnerable in the wild. The security of these kinds of apps is indeed unpredictable and uncontrollable. The traditional server-side based malware detection is challenging to detect such applications: 1) it is time-consuming to upload the app to server before the installation, especially for larger apps; 2) the uploading process on the Internet is not secure. For example, attackers may modify the malware during the uploading period such that an incorrect "benign" result is returned. As a result, the user will install the malware. Hence, a last line of defense on Android devices is necessary and much-needed. To address the severe problem, we intend to conduct Android malware detection on Android devices.

Actually, machine learning-based approaches have achieved better performance compared with other approaches in Android malware detection. In this paper, we intend to deploy the trained deep learning (DL) models on server-side to Android devices. While a computational intensive deep learning software could be executed efficiently on server-side with the GPU support, such deep learning models usually cannot be directly deployed and executed on other platforms supported by small Android devices due to various computation resource limitations, such as the computation power, memory size, and energy. Therefore, we use TensorFlow Lite[2] for Android to migrate the deep learning trained models. Due to the performance limitations of Android devices, We first summarize

*Sen Chen is the corresponding author

[1]https://forum.xda-developers.com/
[2]https://www.tensorflow.org/lite/

and propose 7 feature categories such as permissions, API calls, and opcode sequences according to the existing work to investigate their corresponding performances with the deep learning algorithm. Based on the metrics of accuracy and time cost (i.e., time of feature extraction and model prediction), we propose an effective Android malware detection system, MobiDroid, leveraging deep learning models to provide a real-time secure and fast response environment on Android devices.

The proposed system is performance-sensitive due to the performance limitations of Android devices. Therefore, real users are able to trade off classification accuracy and time cost in practice. In our experiments, MobiDroid achieves a relatively higher classification accuracy (i.e., over 97% accuracy) with relatively lower overhead (i.e., 17.76 seconds in total).

Overall, this paper makes the following contributions.

- We propose MobiDroid, a device-end solution to protect Android devices from Android malware in real-time efficiently. To the best of our knowledge, this is the first work to detect Android malware directly on Android devices rather than server side.
- We evaluate and investigate the different performances with various feature categories for deep learning algorithm, and further provide an effective solution according to the classification accuracy and time cost. Moreover, the corresponding results can be used to help users to trade off classification accuracy and time cost in practice.
- In our experiments, we conduct a comparative study between machine learning algorithm and deep learning algorithm on Android devices in malware detection, demonstrating the usefulness of our approach.

In summary, existing techniques mainly focus on detecting Android malware on the server side based on the information from the APK file and the source code. Different with the existing techniques, this paper performs the first study on the Android malware detection performances with various feature categories on the mobile side, which serve as a starting point for further study of this important area.

The rest of this paper is organized as follows: Section II introduces the background of this work, Section III details our proposed approach to detect Android malware. We conduct our experiments in Section IV. Section V discusses the limitations of MobiDroid. We list the related work in Section VI. Finally, we conclude this paper and discuss the future work in Section VII.

II. Preliminaries

In this section, we briefly introduce the structure of Android apps, the existing security mechanisms of Android apps, and the migration/quantization procedure of trained deep leaning models.

A. Android Apps

To execute the code of Android apps, Android developers compile their source code and other components, like application structure files and other resources, etc., into an Android application package (APK). APK is a compressed application file for Android platform, which is used to deliver Android mobile applications. For each APK, it contains a manifest file, Dex files, resources, assets, and certificates.

The manifest file contains the meta-data for Android apps, which defines the package name and application ID, app components like Intent filters, activities, and services, etc., permissions, device compatibility, like uses-feature and uses-sdk, etc. Dex files as extension are Dalvik executable code, which can be executed on Dalvik virtual machine in Android OS and converted from Java bytecode via an alternative instruction set. However, the instruction format in Dex is quite complicated and hardly interpretable by a developer. To make them more accessible, they are often decompiled into smali files by reverse engineering, which contain the same contents as Dex, but have a better syntax format before manual analysis.

B. Security Mechanisms

The existing security mechanisms can be mainly divided into two categories, which are application market and Android OS platform aspects in practice.

From the aspect of Android market, the official market (i.e., Google Play Store) provides a security verification when the APK uploaded. For instance, Google provides protection backed by its machine learning algorithm. Some high-quality third-party markets also present security check for the uploaded applications. For example, ApkMirror[3] not only provides the signature verification, but also performs a protection service provided by GuardSquare. However, most of current security check service provided by third-party markets is very simple and limited. Some of them only contain a signature verification, which can be bypassed easily. Thus, users, who download applications from the third-party markets, install and use it at their own security risk.

On the device, there exist a lot of antivirus applications provided. The most famous applications, like Avast and Kaspersky, mainly provide their antivirus service by monitoring the privacy-sensitive components on the device and an application scanning with their on-cloud virus database. Besides the protection from outside, Android OS also provides some strong built-in security mechanism, like application sandbox, etc. Application sandbox mechanism provides an independent execution environment for every application. Hence, the attack from an application can only work on its own requested components. For instance, if Bluetooth permissions and actions liked activities are not required in the application, the attack can never access the functions provided by Bluetooth. Different from other systems, like Linux and Windows, malicious code in Android OS cannot easily hijack the whole system, unless a developer promises it every available system-level component hand by hand.

C. Deep Learning Model Migration and Quantization

After a DL model finishes the training process and is ready to deploy to a target device, it oftentimes goes through either

[3]https://www.apkmirror.com/

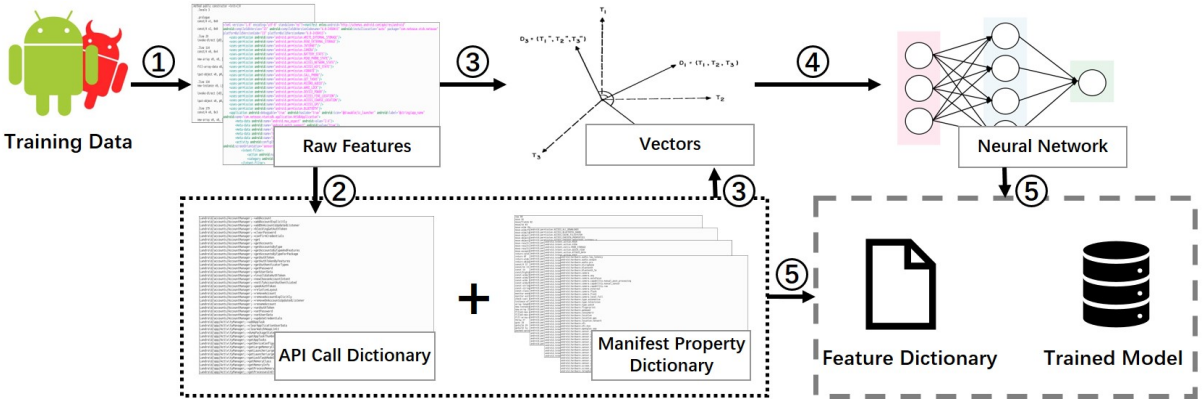

Fig. 1. The processes of feature preparation and deep learning model training

quantization, or platform migration, or both, before deployed to end-user applications, such as mobile devices, self-driving cars, and video surveillance. This is because the training phase requires a vast amount of computation and energy resources. As the model size and the complexity of the tasks grow, more data are needed to train the network till reaching optimality, which could spend days, if not weeks, in training on high-performance GPU clusters. On the other hand, the deployment of the DNN models is usually faced with the resource-constrained environment with limited computation, storage, and power.

Due to environment difference of a target platform (e.g., mobile phones, green energy embedded systems) and training platform (e.g., often equipped with GPUs), a DL model often goes through a customization phase to cater specific software and hardware constraints of a target platform. Quantization reduces the precision of a DL model so as to improve the computation efficiency, reduce memory consumption and storage size, which has become a common practice when migrating a large DL model trained on the cloud system to a mobile or IoT devices with low computation power.

Recently, the rapid development of system-on-chip (SoC) acceleration (e.g., Qualcomm Snapdragon, Kirin 970, Samsung Exynos9) for AI applications provides the hardware support and foundation for universal deployment across platforms, especially on mobile device, edge computing device and so forth. Some lightweight solutions are proposed for mobile platforms such as CoreML,[4] TensorFlow Lite, Caffe2 Mobile and Torch Android. Likewise, a solution is also proposed for deploying DL models in the web environment (e.g., TensorFlow.js). It proposes a chance to deploy the DL-based malware detection task on a mobile device directly. Hence, in this paper, we perform the first study on the performances of DL-based malware detection on mobile devices.

III. Approach

In this section, we first introduce the overview of our approach, and then detail each of the key phases.

A. Overview of MobiDroid

To achieve our approach, we propose MobiDroid, whose functionality could be divided into two major parts. As shown in Fig. 1, the first part of our system contains *feature preparation* and *DL model training*. We select 3 kinds of feature types based on the investigation of existing studies, which are `manifest properties`, `API calls`, `opcode sequences`, as the input of our deep neural network. The first part allows to generate a *DL trained model* and a *vector dictionary* for the second part. To make the model adaptive to Android devices, we then migrate the prebuilt DL model from the first part to a TensorFlow Lite model, which is mobile readable format. Also, a quantization phase[5], which is a general technique to reduce model size while also providing lower latency with little degradation in accuracy, is presented as a performance optimization for the mobile platforms.

As shown in Fig. 2, the second part loads the migrated/quantized DL model and vector dictionary into mobile device. After that, when an application is downloaded from market or third-party resource, MobiDroid is able to extract feature vector from it and deliver the result to our detection system. Hence, after predicting with the loaded DL model, we obtain a certain level of confidence based on predictive output to know whether the downloaded Android app is a malware.

B. Feature Preparation

Android app is provided as a packed APK file, which contains the compiled binary files, XML files, like manifest file, and other resources, etc. In data processing progress, to get the features from the APK file, we first decode the package to separate files by using ApkTool.[6] Among the decoded application source files, we can extract three different feature types, which are manifest properties, API calls, and opcode sequences, from the raw application data. According to the different contents in each feature, we presented two different vector embedding methods to generate the inputs for the neural network.

[4]https://developer.apple.com/documentation/coreml/

[5]https://www.tensorflow.org/lite/performance/post_training_quantization/

[6]https://ibotpeaches.github.io/ApkTool/

Besides, to determine the features used in our detection system, we also perform a comparison of the extracting and analyzing performance for most commonly used features, which include features not used in our system, in previous malware detection approaches. Based on the result, we selected three performance-sensitive features as our model inputs. The details of the feature selection part will be illustrated in section IV-C.

Feature definition and extraction. In Fig. 1, step ① refers to feature extraction. From manifest file in application package, three features are extracted under XML tag *uses-permission*, *intent-filter*, and *uses-feature*. *uses-permission* contains Android system permissions, which are related to the privacy of an Android user. Android apps must request the relative permission before accessing sensitive user data or certain system feature. Information from *intent-filter* refers to intent objects. Each intent contains a message object which is used to request actions from the app component by developers. This feature define the basic actions, which may be used in attacks, like sending SMS or reading the pictures, etc. from Android devices. From *uses-feature* tags, we extract hardware features. Such as Audio Hardware Features and Bluetooth Hardware Features etc. Each feature refers to a basic hardware usage defined in Android operation system.

From the decompiled *smali* code files, we extract 2 types of features, which are API calls and opcode sequences. API call contains both the method name and the package name of the corresponding method. They may not only contain methods from Android and Java packages, but also involve third-party methods. The opcode sequences are generated by matching the function name in smali files with their opcode hex. Each sequence represents a single smali code file.

Feature vector generation. To represent the features extracted from raw data as a computable format, we built a dictionary for direct matching on permission, intent, hardware component, and API call features. Due to the complexity among opcode sequences, it is difficult to represent them through direct matching. To handle this problem, we embedding them to a one-hot vector per step in our neural network.

The dictionary for features in manifest are generated from the Android source code which are predefined by Google developers. Totally, there are 324 permissions, 213 intents and 76 hardware features listed in dictionary.

By extracting and analyzing the API calls from more than 50,000 real-world Android applications, we find that there are a lot of API calls are not related to any sensitive components in Android OS. For example, the *View* loading API is widely used in most applications, but it cannot participate in any kinds of attacks. Meanwhile, among our dataset, most API calls from the third-party packages appeared only few times. In other words, a third-party API call in an application may never appear in others. Thus, we remove all uncommon third-party packages and the API calls which are not related to any sensitive components manually by experience and get the API call dictionary, which contains 1509 APIs, in the step ② of Fig. 1.

According to the combined dictionary of manifest property and API call features, the size of vector is determined by the dictionary size of features. For each Android app, the vector is represented by mapping the retrieved values to the dictionary size dimension vector space in step ③.

TABLE I
DEEP NEURAL NETWORK ARCHITECTURE

Input		
Embedding Layer	input:	(None, None)
	output:	(None, None, 8)
Reshape	input:	(None, None, 8)
	output:	(None, 1, None, 8)
Convolutional Layer	input:	(None, 1, None, 8)
	output:	(None, 64, None, 1)
ReLU		
Reshape	input:	(None, 64, None, 1)
	output:	(None, 64, None)
Global Max Pooling	input:	(None, 64, None)
	output:	(None, 64)
Linear Dense Layer	input:	(None, 64)
	output:	(None, 16)
ReLU		
Linear Dense Layer	input:	(None, 16)
	output:	(None, 2)
Softmax Classification		

C. DL Model Training

We present a convolution neural network (CNN) to train the classifier for malicious and benign Android apps, with our generated input feature vector delivered by step ④. However, it is difficult to figure out the correlation between each dimension of feature vector and the detection target. Thus, we train 7 test deep neural network models for each single feature category and their combination, to determine which feature will be considered as the input of our training network. By comparing their accuracy, precision, and recall etc., we determine to use the 3 feature categories (i.e., manifest properties, API calls, and opcode sequences) combination model as our pre-trained model, which can be deployed on mobile devices. The details of our evaluation will be illustrated in section IV-C.

As shown in Table I, the first layer of our DL model is *feature embedding layer*. With input vector and sequences, we need to unite them as an entire input before sending to the training parts. Hence, we transform the converted hex sequences, which represent the opcode sequences, to a one-hot vector within a lookup table for each sequences bundle. We then combine the result vector with the vector, which is generated by direct matching, together in a combination layer. The resulting vector is reshaped to a matrix and send to the next layer. The second layer is the convolution layer, which receives the embedded matrix as its input and applies convolution filters to produce activation maps for each batch. As a result of the unfixed-length of the opcode sequences, batches surely have different length. Thus, to transform the results in fixed-length vector for the hidden layer, a global max pooling is used after activation. Finally, the fixed length vector is passed to a hidden full layer, which is a multi-layer perception, for classification. To detect the relation between the result vector, we construct two sublayers in the hidden

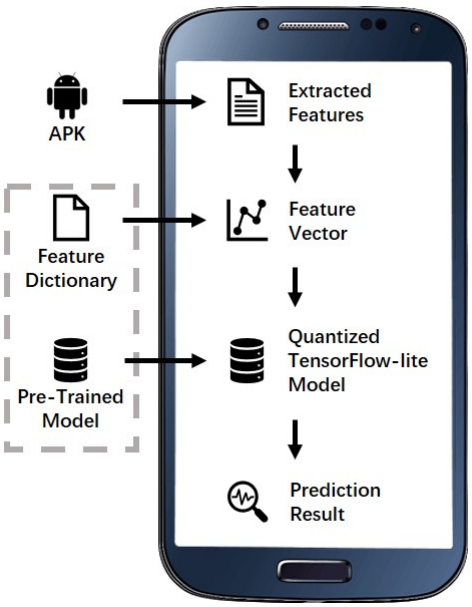

Fig. 2. The Overview and Workflow of MobiDroid

layer, each of them contains a Rectified Linear Unit activation function. At last, the result from the hidden layer is passed to a soft-max classifier function to get the final training result.

D. DL Model Migration and Quantization

To deploy our pre-trained DL model on Android platform, we convert and migrate the model from our server-side platform, which is implemented on Keras,[7] to TensorFlow-lite model, which is supported by Android operating system. To achieve this target, we firstly migrate our pre-trained model to a TensorFlow model first. Following the Google TensorFlow guidance, we then migrate the TensorFlow model to a mobile readable TensorFlow-lite model. Apart from the model migration, we also quantize our pre-trained model to improve the performance on the mobile platform, which does not affect the accuracy of detection.

E. Detection System Architecture (MobiDroid)

Before conducting a real-time detection, the quantized TensorFlow-lite model and feature dictionary should be deployed to the detection system in advance. According to Fig. 2, there are three steps before completing the prediction.

The first step of MobiDroid is feature preparation. While an APK file is received, MobiDroid first decodes it into original resources and *smali* files. Across the extraction step, there include three features described above, which contains manifest properties, API calls, and opcode sequences, generated as outputs. To generate the input vector from extracted features, we re-implemented our Python vector generation scripts in Java to retrieve the prediction inputs for the target application. Hence, we can get the manifest property vector and API call vector, except for the opcode sequence. To combine the multiple kinds of inputs, we perform a binarization, which is the same as the embedding idea used in our training layer, for the sequence information. As a result, an opcode sequence vector is generated. While all the features are transformed into a vector. We connect them together as the detection model input at the end of the second step. The third step is app prediction. With the help of our migrated and quantized detection model, which deployed from the training part, MobiDroid sends the combined application vector to the detection system and obtain the final prediction result.

[7]https://keras.io/

TABLE II
MALWARE DATASET

Malware Dataset	Orginal Size	Reorganized Size
Drebin	5,560	5,527
Genome	1,260	1,148
Contagio	360	338
Pwnzen	1,830	1,807
VirusShare	20,000	12,679
Total	29,010	21,499

IV. EXPERIMENTS

In this section, the goals of our experiments are to determine: (1) the different performance of different feature types (manifest properties, API calls, and opcode sequences); (2) the different accuracy of different feature categories; (3) the different performance between DL models and machine learning models.

A. Dataset

As shown in Table II, we collect more than 50,000 Android apps in total. Specifically, these apps consist of 29,010 malware, and others are benign apps crawled from Google Play Store. However, these might be malware on the official market. To filter the potential malware as far as possible, we upload them to VirusTotal[8], which is an online antivirus service with over 50 scanners, to make a verification. The 29,010 malicious samples contain 5,560 apps downloaded from Drebin [1], 1,260 apps validated in Genome project [35], 20,000 crawled from VirusShare, and the remaining are used in KuafuDet including 360 from Contagio Mobile Website and 1,830 from Pwnzen Infotech Inc. In summary, we collect a large-scale dataset of benign and malicious samples for the following experiments.

Since our dataset come from multiple sources, there have a lot of duplicated samples. Therefore, we perform a hash check for eliminating redundant applications among malicious and benign applications. During the data prepossessing, which has raw data decoding and feature vector generation steps, we receive some failed cases due to the capabilities of ApkTool and the vector generation scripts. While the rest of the failures are just caused by broken APK packages, we also remove them directly. As a result, we choose 21,499 benign and malicious samples respectively from our dataset to conduct the following experiments.

[8]https://www.virustotal.com/

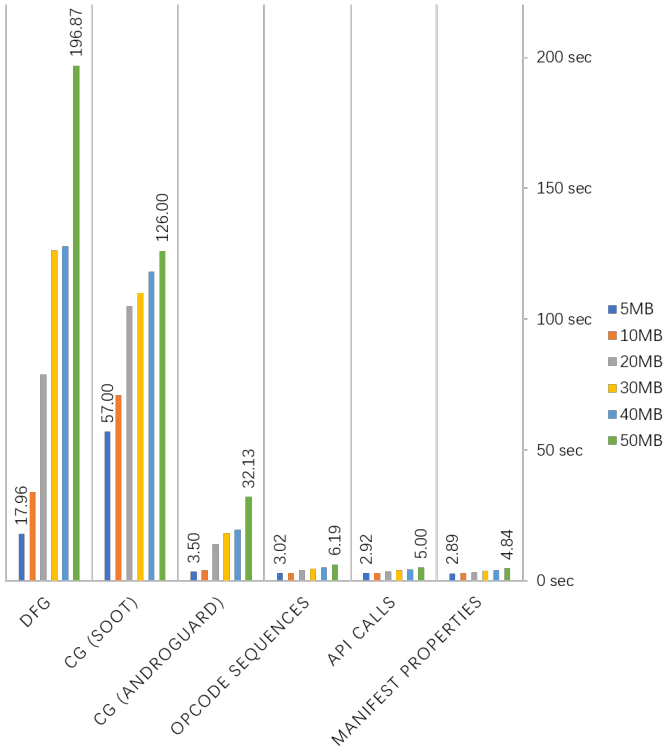

Fig. 3. Processing time of different feature types

B. Experimental environment

All the experiments are run on an Ubuntu 14.04 server with two Intel Xeon E5-2699 V3 CPUs, 192GB RAM, and NVIDIA Tesla P40 GPU and Nexus 6/6P mobile devices. The implementation language of our system on server side is Python. The data preprocessing is depended on AndroGuard[9] and ApkTool. The deep neural network and training project are implemented with Keras, Numpy, Scikit-learn and TensorFlow libraries.

C. Feature Selection

To get access to the necessary information for our experiments, we use 4 different kinds of existing tools, which are ApkTool, AndroGuard, Soot[10] and FlowDroid [43]. ApkTool is a tool for reverse engineering Android apk files, which can decode the apk file and generate the decompiled resources, which contains manifest etc., and *smali* files. AndroGuard is a python tool, which can not only decode the resources but also disassemble bytecode to Java source code. Also, with the help of AndroGuard, we can easily generate the call graph and data flow graph for an Android app. Soot is a Java optimization framework, which can be used with FlowDroid to extract the call graph.

Performance comparison of feature types. Nowadays, Android attacks are discovered to be more and more sophisticated. Consequently, Google is still improving the defense mechanism of Android OS. Most newly detected attacks are not limited to hijack the basic system components, and some of them are triggered by the behaviors among app components. Thus, The behaviors in Android apps become a significant characteristic. Therefore, malware detection with full-scale information graphs may have more semantics than that with weak semantic information like permissions and API calls, etc. Moreover, from the aspect of attacks, it is more difficult to evade malicious behaviors under semantic features. For example, a call graph represents calling relations between subroutines in the source code. Data flow graph is a graph not only represents calling relations, but also provides information about the inputs and outputs of each entity. Inter-component Communication Graph (ICCG) [33] is a graph provides the communications between components and threads inside Android applications and the components itself. The communications contain Intent, Message, Binder and Persistent storage, which construct the run-time inner relations of Android application.

Apart from the above graph-related feature types. Permission, intent filter, uses-feature are components defined in Manifest.xml, which provide essential information about the application. Permissions are related to the privacy of an Android user, which are the most important part of Android OS defense mechanism. Each Android app requests the relative permission before accessing sensitive user data or certain system feature. Therefore, the principle of a large number of attacks is focusing on bypass the permission checking, while invoking some sensitive components on devices. Intent filter contains a message object which used to request actions from the application component by developers. Basic actions, like sending SMS or reading the pictures, etc., are defined in their related Intents. Request these actions are the only way to perform a system level action or modify a basic configuration on devices, while an attack occurs. Uses-feature defines the basic hardware features, like Wi-Fi hardware features and Bluetooth hardware features, etc., for each Android application component.

For example, considering a spy application, the core idea of it should be monitoring the camera, microphone, etc. Hence, the attacks often hide in some components, which have hardware access defined in uses-feature. Otherwise, it is unable to access the target hardware without them. API calls represent the API calling information existed in the application, which provides the name of API call and the corresponding package name. Opcode sequences provide a whole map of opcode functions for the entire application.

Since mobile device is often performance-sensitive, to provide detection service on a mobile device directly, we take the performances of different feature types into consideration. As a result, we analyze the processing and analyzing time for each of the potential input features on both server-side and mobile device to decide the feature type selection. The result in Fig. 3 shows the time consuming of most full-scale information graphs are too large for our performance-sensitive approach on mobile device. For instance, the processing and analyzing time of DFG takes more 196.87 seconds on 50MB

[9]https://github.com/androguard/androguard/
[10]https://github.com/Sable/soot/

TABLE III
DIFFERENT PERFORMANCES OF FEATURE CATEGORIES

Feature Categories	Accuracy (%)	Precision (%)	Recall (%)
Manifest Properties	77.65%	77.47%	77.47%
API Calls	92.00%	92.00%	92.00%
Opcode Sequences	94.79%	94.79%	94.79%
Manifest Properties & Opcode Sequences	95.66%	95.66%	95.66%
Manifest Properties & API Calls	90.37%	90.37%	90.37%
API Calls & Opcode Sequences	95.48%	95.48%	95.48%
Manifest Properties & API Calls & Opcode Sequences	96.87%	96.87%	96.87%

application and even 17.96 seconds on 5MB application on average. In our approach, the detection should be performed in a reasonable period comparing to the application installing time, users cannot buy it if the reacting time takes too long. Other features processing and analyzing time costs are quite limited, compared with the average application installing time. Consider the time cost of API calls. 5MB application only takes 2.92 seconds and 50MB application takes 5 seconds. Comparing to the full-scale graphs, like DFG and CG, the time cost of opcode sequences, API calls, and manifest properties are more acceptable. Therefore, we decide to accept there 3 kinds of feature types as our model inputs.

Accuracy comparison of feature categories. As shown in Table III, to find out the correlation between selected features, we list 7 feature categories to investigate their corresponding accuracy. Consequently, we train 7 test models with both single feature type and combined feature types as inputs. For each test model input, we trained them with our convolution neural network. The only difference between them is we remove the embedding layer, while the input only contains manifest properties and API calls vector. We divide our dataset, which contains 21,499 malware and 21,499 benign applications, into three parts, 70% of them are configured as training data, other 30% are split into validating and testing set. Table III shows the model results of each input configuration. The results show the accuracy of combined features model is obviously higher than any single feature models. However, the accuracy of manifest properties & API calls categories is larger than the manifest properties-based model, but it is smaller than the accuracy of API calls-based model. Actually, the feature category of manifest properties is always used to detect Android malware. Consequently, the malware attackers intent to evade some of the features such as adding good features in Manifest file to attempt to bypass classifiers. So this kind of feature type has some interference effects. Considering both the 3 feature types (i.e., manifest properties, API calls, and opcode sequences) combined model and the combination of the manifest property features and opcode sequences have a better result than the single feature models. We finally select the 3 features combined model, which has the best result, as our detection model. There are some reasons that the 3 feature types-based model has better performance than others. For example, the selected category has more semantics than other categories, and some combinations of different features across these three feature types may trigger and reflect the sophisticated malicious behaviors.

TABLE IV
PERFORMANCES OF MOBIDROID

Devices	Quantization	Accuracy	Preparation Time (s)	Prediction Time (s)
Nexus 6	No	97.35%	16.60	9.35
	Yes	97.35%		7.23
Nexus 6P	No	97.35%	13.56	6.54
	Yes	97.35%		4.20

D. Effectiveness Evaluation of MobiDroid

Accuracy and time cost on mobile device. To evaluate the response time of our mobile detection system, MobiDroid, we measure both feature preprocessing and prediction time for both quantized and non-quantized DL models on our Android devices (i.e., Nexus 6 and Nexus 6P).

The preprocessing time consists of raw data processing and features analyzing time for each feature. Raw data processing is the first step, which decodes the application into resource files and *smali* files with ApkTool. This step costs more than 80% of the prepossessing time. Because we have to run ApkTool, which is a .jar package, on a JVM instead of the original Android package compiling environment, the performance of the processing still has a lot of space to be optimized, if we have an implementation of decoding tool on Android. Analyzing time contains the time cost for generating manifest property and API call vector and opcode sequences from the decoded features. The predicting time is the time measured from loading inputs to get the result.

The test data contains 2,000 randomly selected applications, half of them are malware, the others are benign applications. In Table IV, by comparing quantized and non-quantized models, the result of prediction time shows that quantization reduces a lot of time cost (i.e., 16.60 vs. 13.56). Meanwhile, the accuracy of our test remains unchanged (i.e., 97.35% accuracy). The prediction time is also acceptable for mobile users (i.e., less than 10 seconds).

Comparison between DL and ML on mobile device. In addition, to show the strengths of our mobile malware detection system, MobiDroid, we investigate a similar mobile end malware detection approach which based on the machine learning algorithm (i.e., SVM). The result in Table V shows our approach can gain a much better accuracy with both Manifest Properties (77.65% vs. 65.00%) and API calls (92.00%

TABLE V
COMPARISON BETWEEN MOBIDROID AND ML CLASSIFIER UNDER DIFFERENT FEATURE CATEGORIES

Systems	Feature Categories	Accuracy (%)
MobiDroid	Manifest Properties	77.65%
	API Calls	92.00%
ML Classifier	Manifest Properties	65.00%
	API Calls	88.00%

vs. 88.00%) as input than the approach by machine learning-based approach. As a result of the unfixed-length and content of opcode sequences, we cannot apply the machine learning algorithm on this kind of feature type directly.

Remarks: The feature types of manifest properties, API calls, and opcode sequences have a better time performance (i.e., less than 10 seconds) than DFG and CG. The combination of feature types including manifest properties, API calls, and opcode sequences achieves a better detection accuracy (i.e., over 97%) than other combinations. Compared with machine learning-based approach, MobiDroid performs a better detection accuracy on mobile platform.

V. LIMITATIONS AND THREATS TO VALIDITY

In this section, we introduce the limitations of our approach and the threats to validity in this paper.

Limitations. Due to the limited application dataset, MobiDroid has a similar limitation as to other deep neural network-based malware detection ideas. Considering a new malware family detected, the situation may be that only a few malware in this family are confirmed in a long period. Thus, there are not many new malware can be used as part of the training dataset. If the proportion of this new family is quite limited in dataset, there may have an uncertain training result, which makes MobiDroid difficult to be applied as the first-order protection to against the new detected malware family.

Threats to validity. There are several threats may influence our validity. Currently, the most important threat is the hardware performance of the deployed device. Considering our experiment result in Table III, we apply our mobile malware detection system on two devices (i.e., Nexus 6 and Nexus 6P). Nexus 6 and Nexus 6P are two Android smartphones presented by Google in 2014 and 2015. We can consider their hardware performance as an average among Android devices. The preparation and prediction time on Nexus 6P is 13.56 and 4.2 seconds on average and the time on Nexus 6 is 16.6 and 7.23 seconds. Considering a worse case, if the device, which we want to deploy MobiDroid has an old hardware spec, the time cost may grow to an unacceptable number. However, the newest Android devices provide GPU support to TensorFlow-lite. With the performance promotion by GPU support added, the speed of depth computing[11] in Google camera can be improved for 10 times on Pixel 3. Thus, in the future, the performance of Android device will not be a threat to our approach.

[11]https://ai.googleblog.com/2018/11/learning-to-predict-depth-on-pixel-3.html

VI. RELATED WORK

In this section, we will summarize the current work about malware detection. Generally, traditional techniques adopt static analysis and dynamic analysis to classify benign applications and malware applications.

Some techniques are proposed based on analyzing the XML files from the APK file. C.-Y. Huang et al. [2] classify the benign data and malware data using the permission information in manifest and files structure as features. Similarly, Z. Aung et al. [3] also consider the permission. Differently, they concentrate on the permission requires in the source code, not only the static permission information in the manifest file. E. Chin et al. [12] propose ComDroid, which detects malware by analyzing the manifest file.

There are also some techniques which are based on the API analysis [54]. L. Deshotels [4] et al. classify the benign/malware applications based on the frequency of API calls. M. Zhang et al. [5] develop a classifier, DroidSIFT, which is based on the API dependency graphs. D. Arp et al. [7] propose Drebin, which is a classifier using features from both of XML files and API calls. In addition, Drebin can be used in the mobile end solution. Y. Zhongyang et al. [11] introduce DroidAlarm, which analyzes the inter-procedural call graphs constructed by the relationship between permissions and the interface to identify attacks. L. K. Yan et al. [20] propose DroidScope, which generates semantic information from API call traces and Dalvik opcode traces. D.-J. Wu et al. [8] propose the technique, DroidMat, to detect malware with API traces, intent, communication and some other the life-cycle information.

Another line of research is conducted based on the program analysis (e.g., control flow graph), which is more expensive than the XML-based and API-based approach. However, the result tends to be more precise. Narayanan et al. [18] present an online SVM classifier, which uses the control flow graph generated from the source code as input. W. Enck et al. [21] propose TaintDroid, which is a taint analysis tool for Android applications. It detects the leakages with the data flow analysis on target sensitive data. G. Z. Meng et al. [32] propose a deterministic symbolic automaton (DSA) based detection system, in which DSA contains the corresponding components of the target application. Furthermore, they develop a system, DroidEcho, which detects attacks with the inter-component communication graphs (ICCG). ICCG provides both the call graphs and sensitive data flow in applications. In addition, some CFG-based static analysis [55]–[57] could be useful to capture more fine-grained features for detecting malware.

Deep learning has achieved great success in many applications, there exist also a lot of neural network based approaches. Z. Yuan [28] et al. provide Droid-detector, which performs on a deep belief network, W. Yu [29] et al. present a malware detection system, which uses permission and API call traces as input. N. McLaughlin [30] et al. use the convolution neural

network in detection. The raw opcode sequences of target applications are used as the input feature. Kim [52] et al. present a malware detection framework based on multiple neural networks. Every network has a single feature input and output score. The final detection result is a combination of all the models. K. Xu [53] et al. proposed DeepRefiner, which is an efficient two layer malware detection system. They involved XML features as the first layer to perform a fast detection first. At the end of the first layer, if it cannot promise the result with a high rate, it will use some more complicated features, like bytecode information, etc., in the second layer to determine whether the target is a malware.

In addition, there are still some other techniques. A. Demontis et al. [19] propose an algorithm to mitigates attacks like malware data manipulation. T. Blsing et al. [22] introduce AASandbox, which performs detection with combination information of both static and dynamic analysis. A. Shabtai et al. [23] and A.-D. Schmidt et al. [24] provide the abnormalities identification systems, which use run-time device information, such as CPU usage etc.. J. Sun et al. [17] train a machine learning based classifier, which use the distance of keywords to detect the malware. L. Lu et al. [13], P. P. F. Chan et al. [14], K. Lu et al. [15] and F. Wei et al. [16] focus on detecting vulnerable components, which may hijack the applications. W. Zhou et al. [9] provide a malware detection system, DroidMoss, which uses hash comparison to detect repacked Apks. M. Grace et al. [6] propose RiskRanker, which performs detection via analyzing specific application behaviors.

Existing techniques mainly focus on detecting malware on the server side based on the information from the APK file and the source code. However, with the rapid development of AI chips on mobile devices, the research about malware detection on the mobile side is still rare and on demand. Different from the existing techniques, this paper performs the first study on the malware detection performances with various feature categories on the mobile side.

Recently, some deep learning testing techniques [58]–[64], which are used to test the quality of deep neuron networks, have been proposed. We will adopt such techniques to testing the trained model based on different features in the future.

VII. Conclusion and Future Work

This paper presents MobiDroid, a performance-sensitive Android malware detection system on the mobile platform. It consists of two parts. The first server-side part is designed for feature dictionary generation and deep neural network training. The second mobile end part applies the trained model and dictionary in a mobile detection system. Meanwhile, a conversion and quantization phase is performed as a middle adaptation for the trained model between two parts. According to the effectiveness of selected features and the efficiency of feature extraction, MobiDroid can provide a reliable (i.e., over 97% detection accuracy) and fast reactive (i.e., less than 10 seconds) detection service on mobile device directly. To validate the efficiency and reliability, we evaluate our MobiDroid on two real mobile devices and make a comparison to machines learning-based approach.

In the future, we will extend our current work from three directions. Due to the limitations we mentioned in section V, we would like to improve our system against new detected malware families by updating our training dataset timely. Another potential improvement is to improve the run-time performance of mobile detection system, which can bring the user a better user experience. We will also consider extending our feature selection method to provide more application information and increase feature semantics.

Acknowledgments

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001) administered by the National Cybersecurity R&D Directorate, and JSPS KAKENHI Grant 19H04086, and Qdai-jump Research Program NO.01277.

References

[1] D.Arp, M.Spreitzenbarth, M.Hubner, H.Gascon, and K.Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. NDSS, 2014

[2] C.-Y. Huang et al, Performance evaluation on permission-based detection for Android malware, in Advances in Intelligent Systems and Applications (Smart Innovation, Systems and Technologies), vol. 2. Berlin, Germany: Springer, 2013, pp. 111120.

[3] Z. Aung et al, Permission-based Android malware detection, Int. J. Sci. Technol. Res., vol. 2, no. 3, pp. 228234, 2013.

[4] L. Deshotels et al, DroidLegacy: Automated familial classification of Android malware, in Proc. ACM SIGPLAN Program Protection Reverse Eng. Workshop, 2014, Art. no. 3.

[5] M. Zhang et al, Semantics-aware Android malware classification using weighted contextual API dependency graphs, in Proc. ACM Conf. Comput. Commun. Secur. (CCS), 2014, pp. 11051116.

[6] M. Grace et al, RiskRanker: Scalable and accurate zero-day Android malware detection, in Proc. ACM 10th Int. Conf. Mobile Syst., Appl., Service (Mobisys), 2012, pp. 281294.

[7] D. Arp et al, DREBIN: Effective and explainable detection of Android malware in your pocket, in Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS), vol. 14, 2014, pp. 2326

[8] D.-J. Wu et al, DroidMat: Android malware detection through manifest and API calls tracing, in Proc. 7th Asia Joint Conf. Inf. Secur. (Asia JCIS), Aug. 2012, pp. 6269.

[9] W. Zhou et al, Detecting repackaged smartphone applications in third-party Android marketplaces, in Proc. ACM Conf. Data Appl. Secur. Privacy, 2012, pp. 317326.

[10] S. Hao et al, PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps, in Proc. ACM Int. Conf. Mobile Syst., Appl., Services (MobiSys), 2014, pp. 204217.

[11] Y. Zhongyang et al, DroidAlarm: An allsided static analysis tool for Android privilege-escalation malware, in Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Secur., 2013, pp. 353358.

[12] E. Chin et al, Analyzing interapplication communication in Android, in Proc. 9th Int. Conf. Mobile Syst., Appl., Services, 2011, pp. 239252.

[13] L. Lu et al, CHEX: Statically vetting Android apps for component hijacking vulnerabilities, in Proc. ACM Conf. Comput. Commun. Secur., 2012, pp. 229240.

[14] P. P. F. Chan et al, DroidChecker: Analyzing Android applications for capability leak, in Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw., 2012, pp. 125136.

[15] K. Lu et al, Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting, in Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS), 2015, pp. 4:14:15.

[16] F. Wei et al, AmAndroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps, in Proc. ACM Conf. Comput. Commun. Secur., 2014, pp. 13291341.

[17] J. Sun et al, Malware on Android smartphones using keywords vector and SVM, in Proc. IEEE/ACIS 16th Int. Conf. Comput. Inf. Sci., May 2017, pp. 833838.

[18] A. Narayanan et al, Adaptive and scalable Android malware detection through Online learning, in Proc. Int. Joint Conf. Neural Netw. (IJCNN), Jul. 2016, pp. 24842491.

[19] A. Demontis et al., Yes, machine learning can be more secure! A case study on Android malware detection, IEEE Trans. Dependable Secure Comput., to be published.

[20] L. K. Yan et al, DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis, in Proc. 21st USENIX Secur. Symp., 2012, pp. 569584.

[21] W. Enck et al., TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones, ACM Trans. Comput. Syst., vol. 32, no. 2, p. 5, 2014.

[22] T. Blsing et al, An Android application sandbox system for suspicious software detection, in Proc. 5th Int. Conf. Malicious Unwanted Softw. (MALWARE), Oct. 2010, pp. 5562.

[23] A. Shabtai et al, Andromaly: A behavioral malware detection framework for Android devices, J. Intell. Inf. Syst., vol. 38, no. 1, pp. 161190, 2012.

[24] A.-D. Schmidt et al, Monitoring smartphones for anomaly detection, Mobile Netw. Appl., vol. 14, no. 1, pp. 92106, 2009.

[25] R. Pascanu et al, Malware classification with recurrent networks, in Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP), Apr. 2015, pp. 19161920.

[26] O. E. David et al, DeepSign: Deep learning for automatic malware signature generation and classification, in Proc. Int. Joint Conf. Neural Netw. (IJCNN), Jul. 2015, pp. 18.

[27] J. Saxe et al, Deep neural network based malware detection using two dimensional binary program features, in Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE), Oct. 2015, pp. 1120.

[28] Z. Yuan et al, Droiddetector: Android malware characterization and detection using deep learning, Tsinghua Sci. Technol., vol. 21, no. 1, pp. 114123, Feb. 2016.

[29] W. Yu et al, Towards neural network based malware detection on Android mobile devices, in Cybersecurity Systems for Human Cognition Augmentation. Cham, Switzerland: Springer, 2014, pp. 99117.

[30] N. McLaughlin et al., Deep Android malware detection, in Proc. ACM Conf. Data Appl. Secur. Privacy (CODASPY), 2017, pp. 301308.

[31] H. Fereidooni et al, ANASTASIA: Android malware detection using static analysis of applications, in Proc. 8th IFIP Int. Conf. New Technol., Mobility Secur., Nov. 2016, pp. 15

[32] Guozhu Meng et al, "Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection, and Classification," in The International Symposium on Software Testing and Analysis (ISSTA), Saarbrcken, Germany, 2016, pp. 306–317.

[33] G. Meng et al. "DroidEcho: an in-depth dissection of malicious behaviors in Android applications." Cybersecurity, 2018, 1(1), 4.

[34] Tang, C. et al. (2019, May). A large-scale empirical study on industrial fake apps. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (pp. 183-192). IEEE Press.

[35] Y. Zhou et al, "Dissecting android malware: Characterization and evolution." In 2012 IEEE symposium on security and privacy, 2012, (pp. 95-109). IEEE.

[36] Schlegel, R. et al. (2011, February). Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In NDSS (Vol. 11, pp. 17-33).

[37] Zhou, Y. et al. (2012, February). Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In NDSS (Vol. 25, No. 4, pp. 50-52).

[38] Zhou, W. et al. (2013, February). Fast, scalable detection of piggybacked mobile applications. In Proceedings of the third ACM conference on Data and application security and privacy (pp. 185-196). ACM.

[39] Yan, L. K. et al. (2012). DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In Presented as part of the 21st USENIX Security Symposium (USENIX Security 12) (pp. 569-584).

[40] Wu, C. et al. (2014, February). AirBag: Boosting Smartphone Resistance to Malware Infection. In NDSS.

[41] Tam, K. et al. (2015, February). CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In NDSS.

[42] Rasthofer, S. et al. (2016, February). Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In NDSS.

[43] Arzt, S. et al. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices, 49(6), 259-269.

[44] Li, L. et al. (2015, May). Iccta: Detecting inter-component privacy leaks in android apps. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 280-291). IEEE Press.

[45] Wong, M. Y.,et al. (2016, February). IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In NDSS (Vol. 16, pp. 21-24).

[46] Yang, C. et al. (2014, September). Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In European symposium on research in computer security (pp. 163-182). Springer, Cham.

[47] Chen, S. et al. (2016, May). Stormdroid: A streaminglized machine learning-based system for detecting android malware. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (pp. 377-388). ACM.

[48] Chen, S. et al. (2018). Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. computers & security, 73, 326-344.

[49] Chen, S. et al. (2016, October). Towards adversarial detection of mobile malware: poster. In Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (pp. 415-416). ACM.

[50] Fan, L. et al. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (pp. 1748-1750). ACM.

[51] Mariconti, E. et al. (2016). Mamadroid: Detecting android malware by building markov chains of behavioral models. arXiv preprint arXiv:1612.04433.

[52] Kim, TaeGuen, et al. "A multimodal deep learning method for Android Malware detection using various features." IEEE Transactions on Information Forensics and Security 14.3 (2018): 773-788.

[53] K. Xu et al, "DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks," 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, 2018, pp. 473-487.

[54] Li, Li et al, "Characterising Deprecated Android APIs," Proceedings of the 15th International Conference on Mining Software Repositories, pp. 254–264, 2018.

[55] Xie, Xiaofei et al. "Automatic loop summarization via path dependency analysis." IEEE Transactions on Software Engineering (2017).

[56] Xie, Xiaofei et al. "S-looper: automatic summarization for multipath string loops." In Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 188-198. ACM, 2015.

[57] Xie, Xiaofei et al. "Proteus: Computing disjunctive loop summary via path dependency analysis." In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 61-72. ACM, 2016

[58] Ma, Lei et al. "Deepgauge: Multi-granularity testing criteria for deep learning systems." In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 120-131. ACM, 2018.

[59] Ma, Lei et al. "DeepMutation: Mutation Testing of Deep Learning Systems." The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE) (2018).

[60] L. Ma et al. "DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems." 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2019).

[61] Xie, Xiaofei et al. "DeepHunter: A Coverage-guided Fuzz Testing Framework for Deep Neural Networks." Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (2019).

[62] Du, Xiaoning et al. "DeepStellar: Model-based Quantitative Analysis of Stateful Deep Learning Systems." Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp.477–487, Tallinn, Estonia,2019.

[63] Xie Xiaofei et al. "DiffChaser: Detecting Disagreements for Deep Neural Networks." Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI),2019.

[64] Zhang, Jie M. et al. "Machine Learning Testing: Survey, Landscapes and Horizons." arXiv e-prints 1906.10742,2019.