DeepDetect: A Practical On-device Android Malware Detector

Saurabh Kumar*, Debadatta Mishra*, Biswabandan Panda†, Sandeep Kumar Shukla*
* *Indian Institute of Technology Kanpur, Kanpur, India*
skmtr@cse.iitk.ac.in, deba@cse.iitk.ac.in, sandeeps@cse.iitk.ac.in
† *Indian Institute of Technology Bombay, Mumbai, India*
biswa@cse.iitb.ac.in

***Abstract*—Over the past few years, Android has become one of the most popular operating systems for smartphones as it is open-source and provides extensive support for wide variety of applications. This has led to an increase in the number of malware targeting Android devices. The lack of robust security enforcement in Play Store along with the rapid increase in the number of new Android malware presents a scope for a variety of diverse malicious applications to spread across devices. Furthermore, Android allows installation of an application from unverified sources (e.g., third-party market and sideloading), which opens up other ways for malware to infect the smartphones. This paper presents DeepDetect that enables on-device malware detection by employing a machine learning based model on static features. With effective feature engineering, DeepDetect can be used on-device. To classify an Android application as malware, it takes ∼5.32 seconds, which is 2.23X faster than API based malware detector, while consuming 0.45% (for 50 applications) of total device energy. DeepDetect provides a malware detection rate of 99.9% for known malware with a 0.01% false-positive rate. For unseen/new samples, it detects more than 97% malware with a false-positive rate of 1.73%. Further, in the presence of obfuscated malware, DeepDetect correctly detects 95.57% of malware samples. We have also evaluated our model against the Pegasus malware sample and with a new dataset after removing the potential biases across space and time.**

***Index Terms*—Android, static analysis, malware detection, security, machine learning.**

I. Introduction

In the recent years, Android has become one of the most popular operating systems (OSes) for smartphones because of its open source nature and large support for different applications (Apps). Recently, a report shared by the International Data Corporation (IDC) for smartphone OSes showed that in the year 2019, the total market share of Android was 86.1% [1]. As a consequence of such large-scale adoption of Android, the security of these devices has become a nontrivial challenge. In the year 2019, security experts at G DATA observed that around 4.18 million new Android malware samples have been discovered [2]. This shows that a new Android malware is born every eighth second [2].

The problem: Malware may get unleashed into the device bypassing the defense system of Google Play Store [3] or from an unverified sources (e.g., third-party market, sideloading). Therefore, on-device malware detection is crucial to stop malware affecting end-user devices. The performance of existing on-device malware detectors [4], [5] in presence of recent malware is unknown. Furthermore, these detectors utilize the API call information for malware detection that are susceptible to code obfuscation and require significant processing time (hence impact battery life). For example, API based malware detectors take ∼11.89 seconds to extract Restricted API information which is 2.23X slower than opcode based detector (see §VII-E). Moreover, most malware detectors like DroidSieve use the PRAGuard [6] dataset to evaluate their efficiency against obfuscated malware. PRAGuard dataset contains malware till March 2013. These samples are outdated and do not represent the current state of Apps.

Our goal: We believe, an efficient and accurate on-device malware detection mechanism should complement the existing offline analysis process to stop malware infecting the end-user devices in an effective manner.

Our approach: To design an on-device malware detector that is faster, consumes less device energy, provides high malware detection rate and low false-positive rate, we use the following approach:

(i) With code obfuscation, selection of correct features is a challenging proposition as many features either negatively impact the accuracy or unnecessarily increase the feature set size with negligible contribution towards accuracy. Therefore, we carefully select features that are either unaffected by obfuscation or we transform them into another form to make them independent of obfuscation.

(ii) The most time consuming step in malware detection is the feature extraction process while other computation such as executing the trained classification model on the feature vectors require significantly less time. Hence, we design a lightweight feature extraction module that can extract features efficiently. We also adapt the N-Gram method for feature construction to preserve the relationship between opcode.

(iii) With the outdated obfuscated malware samples, it is hard to measure a malware detector's efficiency with the current pace of obfuscation methods. Hence, we create a new obfuscated malware dataset that reflects the current state of obfuscation techniques and App design paradigm.

The accuracy of on-device malware detectors ([4], [5], [7],

This work is partially supported by Visvesvaraya Ph.D. Fellowship grant MEITY-PHD-999, SERB and DST through C3i center and C3i hub projects at IIT Kanpur.

[8]) built around machine learning algorithms with static features goes down in the presence of unseen[1]/new[2]/obfuscated Apps. Talos [26] uses only requested permissions to detect a malware, which can be extracted efficiently from an App by using the Package Manager (Android built-in feature). However, a malware detector based on only permissions is not a good solution because it can classify malware as benign that is obtained by introducing malicious code inside a benign App. Drebin [4], IntelliAV [5], Yuan et al. [8] include API call information (suspicious API and/or Restricted API), which requires significant processing time (see §VII-E) to extract from an App. Furthermore, the API call information is more susceptible to the code obfuscation attack, which impacts the accuracy of a model (see §VII-A). Also, in the newer versions of the Android OS, some APIs may go outdated or suppressed, and the same will not be present in the future/unseen Apps.

Techniques ([9]–[12]) that are primarily proposed for deployment in the market place provide good detection accuracy for both new malware samples and obfuscated samples. One choice for on-device malware detection could be the deployment of the same model on a real device. However, these methods extract various features (permissions, intent filters, API calls, native code, etc.), and the processing time required for the extraction of this information is relatively high. For example, DroidSieve [11] takes an average of 2.5 seconds, while Garcia et. al. [9] take around two seconds to analyze an App in an offline manner. Since the processing time requirement for these techniques are high on a server environment with huge processing capabilities, we speculate that the deployment of the same techniques on low-end devices will consume more time and therefore, it is not practical. Note that, Drebin takes 750 milliseconds to analyze an application on a server environment. However, when it is deployed on a real device, its reported analysis time for an App is 10 seconds. Similarly, DroidAPIMiner [12], an API based malware detector takes around 15 seconds for extracting features and 25 seconds of overall time to analyze an App in a server environment with a detection rate of ∼97%.

In this paper, we present DeepDetect that enables on-device malware detection by employing machine learning on static features with significantly less processing time and device energy. Overall, our contributions are as follows:
(i) We reduce the size of Dalvik opcode instruction set by combining the same semantic instruction and represent them as one instruction (§IV-A). We also develop a lightweight opcode information extraction module to extract the opcode sequence from an Android Application Package (APK) inside a real device efficiently (§IV-B).
(ii) We develop a feature engineering framework that can drastically reduce the feature set size (from 7,12,595 to 75 features) while achieving malware detection rate of more than 97% (§V).
(iii) We design an on-device malware detector that is capable of identifying a stand-alone malware (§VI). We show the efficacy of DeepDetect in the presence of known[3] (training samples), unseen, and new malware in terms of detection rate (recall), precision and F1-score (§VII-B). We also evaluate our model against the Pegasus malware samples that have been collected from the CloudSek organization.
(iv) We create a new dataset of obfuscated malware by obfuscating 4993 unique malware with six different categories. We evaluate DeepDetect against these obfuscated samples, DeepDetect correctly detects 95.57% malware, which is an overall drop of 1.55% compared to the same set of non-obfuscated samples (§VII-C). The malware samples in this new dataset are from the year 2019 and downloaded from the AndroZoo [13]. Additionally, we evaluate DeepDetect against a new dataset (spanning over four years 2016 to 2019) downloaded from Androzoo and eliminated both the potential biases i.e. spatial and temporal. In this evaluation, DeepDetect is able to detect ∼97% of malware while generating ∼1.4% false alarms (§VII-D). Further, we show the runtime performance in extraction of different features on a real device in terms of the execution time and device energy consumption. The opcode based malware detector is 2.23X faster than the Restricted API based detector and consume 2.17X less device energy (§VII-E).

II. Background

In this section, we discuss the APK building blocks and Android permission systems.

A. Android Application Package (APK)

APK comprises of five major components [14]. These components are as follows:
Classes.dex: It contains main execution logic of an App, which comprises of four components. These are:
(i) **Activities:** An activity represents a single screen through which a user can interact with the App.
(ii) **Content Providers:** A content provider supplies data from one App to another on request.
(iii) **Services:** These are the processes that keep running in the background without any need for interaction with the user.
(iv) **Broadcast Receivers:** It responds to broadcast messages from other Apps or from the system itself.
Native Libs: At times, a piece of code written in C/C++ is included in the native library. To invoke a native function in an Android App, Java Native Interface (JNI) is used.
Resources: Elements such as images, strings, color value, etc., used in an App fall under the category of resources.
META_INF: It contains meta-information about an App along with the public key of the signing certificate used to verify the integrity of an App.
Application Manifest [14]: It stores all the information about an App like broadcast receivers, content providers, activities, etc. All components need to be registered in this file. If

[1] Samples that are not the part of training set, but they may be from the same period or prior to the samples of training set.

[2] Samples that comes after the samples used for training.

[3] Samples that are the part of training data.

a component is not present in the Manifest file, then its functionality would not be visible to Android.

B. Permission System

Android works on the principle of least privilege. Hence, an App can only perform the task for which it is designed. Android assigns a unique user-ID to every App at the time of installation. This user-ID is then used to enforce the isolation between Apps at run-time such that an App cannot access the data of others. Access to other resources which do not belong to an App, is allowed based on the permissions. All such permissions must be declared in Manifest file [14].

III. Dataset

We use a class balanced dataset consisting of 96,748 Apps. The dataset has 40,402 unique malware distributed across more than 70 different classes of malware. These distributed set of malware helps the classifier to learn more about different variants to identify unseen malicious Apps. In the dataset, the malware samples are collected from AMD [15] and VirusShare [16]. For benign Apps, we have crawled through the Play Store and gathered 65,806 samples. These samples were passed to VirusTotal [17] to make sure they are benign. The Apps that are identified as malicious by even one antivirus engine on VirusTotal are discarded to create the benign dataset resulting in 56,346 benign samples. AMD dataset contains malware collected between the years 2012 to 2016, whereas the malware from VirusShare and benign Apps from Play Store were collected in April 2018. In §IV, we discuss the feature extraction process and the type of features extracted from the dataset.

IV. Feature Extraction

Feature extraction is an important step in machine learning based malware detection systems. In this section, we discuss about the type of features extracted from the dataset along with the process of feature extraction.

A. Type of Features

In this work, we extract features from two locations, i.e., (i) App Manifest file [14] and (ii) Dex code. The Manifest file contains information about the Android App, whereas the Dex code holds the main execution logic. Our primary goal is to design a malware detector based on the static features while ensuring that the prediction accuracy is not affected due to the obfuscation. Note that, here obfuscation includes all defense mechanisms like packed malware, dynamic code loading, native code, and others [18]. Generally, the obfuscation is applied to the code available in the Dex file, where a malware writer hides the use of actual API by transforming it into another form. The extraction of used API information from the Dex file may lead to miss classification, and the overall performance of the detection model may be negatively impacted. Therefore, we utilize low-level Dalvik bytecode (opcode) instead of the high-level API information and the features from the Manifest file to design an obfuscation-resilient malware detector.

TABLE I
Reduced instruction set with description.

Symbol	Description
A	Arithmetic operation instructions
B	Branch instruction (Conditional jump like if-eq)
C	Comparision instruction like cmpl-float
D	Data Definition instructions like const/4
F	Type conversion instructions (int-to-long, int-to-float)
G	Get instructions (aget, aget-wide)
I	Method call instructions (invoke-direct, invoke-virtual ...)
J	Jump instructions (Unconditional) like goto
L	Lock instruction, use to acquire/release a lock (monitor-enter and monitor-exit)
M	Data manipulation instruction like move and its variants
O	Exception instruction (through)
P	Put instructions (aput, aput-wide)
R	Return instruction like return-void
S	Bit-wise operation instructions (and-int, shl-int)
T	Type judgement like check-cast
V	Array operation instructions like array-length
X	Switch case instructions

Features from the Manifest file: There are four categories of features [4] that can be extracted from the Manifest file. The description of the features are as follows:
(i) **Requested permissions**: An App has to request permission to access important and sensitive information. Malicious software tends to request certain permissions more often than benign Apps.
(ii) App components: In an App, there are four types of components. Each component defines user interfaces or interfaces to the system. They are—Activities, Content Providers, Services, and Broadcast Receivers.
(iii) Intent filters: Using intent, inter-process and intra-process communication is performed. Malware often listens to such intents.
(iv) Hardware components: Access to a certain hardware may have some security implications.

Features from the Dex code: API call information extracted from the Dex code is prone to obfuscation attacks. As an API is treated as a string by the static analyser, a malware can bypass it very easily because the execution happens with a stream of Dalvik opcodes. Dalvik instruction set contains 230 instructions to perform a designated task. These instructions include method call instruction, branch instructions, data manipulation instructions, and others. We generate a 2-Gram (N-Gram [19]) sequence of Dalvik opcodes for each function to use them as features for malware detection. N-Gram is widely used in natural language processing, and it has also been adapted for malware analysis.

As the Dalvik instruction set is large enough (230 instructions), a possible number of unique features obtained using N-Gram is approx 230^N. Such a large number of features are not suitable to design an on-device malware detector. Therefore, we have clubbed several instructions to reduce them into one instruction based on their usages (Table I), like move instructions or method call instructions similar to the approach used in [20], [21]. However, our reduced instruction set contains 17 instructions (Table I), which is slightly more than the reduced instructions set of TinyDroid

TABLE II
EFFECT OF FEATURE SELECTION & ENCODING ON EXTRACTED FEATURES.

Category	#Features Original	 Encoding
Requested Permission	23,175	668
Hardware Component	245	245
Intent Filters	50,257	1
Activities	5,24,989	1
Services	57,202	1
Broadcast Receivers	49,751	1
Content Providers	6,659	1
Custom Permissions	0	1
2-Gram Opcode Sequence (2-Opc)	317	317
Total Features	**7,12,595**	**1,236**

[21] and Dong et al. [20]. Our reduced instruction set is based on the 228 instructions (excluding the NOP and empty method call instruction), whereas the simplified instruction set of TinyDroid and Dong et al. includes only 107 and 218 instructions, respectively. Apart from the size of the instruction set, we also include features from the Manifest file and reduce the feature set size through the feature engineering module which is different from the Dong et al. and TinyDroid.

The above mentioned features are extracted using Androguard [22] and represented as strings. From each sample, we have extracted all the information discussed above as features. In Table II, column named **Original** shows the number of unique features extracted from the dataset. There is a total of 7,12,595 unique features extracted from the Manifest file and Dex code. Using such a large number of features for an on-device malware detection leads to significant overhead in terms of processing time and computation requirements. To overcome this limitation, we need to reduce the dimension of the feature space to build an efficient on-device malware detector. We discuss the process involved in reducing the feature set size in §V. Next we discuss the efficient feature extraction method designed for a real device.

B. On-device Efficient Feature Extraction:

We use the features from two locations as discussed in §IV-A. To extract features from the Manifest file, Android provides an in-built functionality called Package Manager (referred to as PM). Whenever an App gets installed (or gets updated, which is done frequently) on an Android device, the PM maps all the information related to the Manifest file. Hence, we use PM directly to extract features from the Manifest file efficiently. However, Android does not provide any in-built functionality to extract information (opcode) from the Dex files of an APK. Hence, we have designed an efficient and lightweight opcode information extractor with the help of *DexLib2* library.

The *DexLib2* is a Java library to process the Dalvik executable code, which has been used by many heavy APK processing frameworks like APKTool to perform reverse engineering. One can argue that, if APKTool is available, then why a new feature extraction method is needed for on-device? Why can we not use APKTool directly? The reason is, APKTool disassembles an APK and dumps the disassembled code into the secondary storage in smali code (see listing 1, a smali code snippet of a method disassembled using APKTool). The generated smali code is then used to extract the features by parsing them, requiring a lot of string processing/comparison and file system operation. Both file operation and string comparison requires significant processing capability and time.

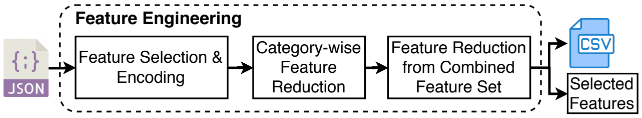

JSON
Feature Engineering
Feature Selection & Encoding
Category-wise Feature Reduction
Feature Reduction from Combined Feature Set
CSV
Selected Features

Fig. 1. Flow of feature engineering module.

```
const/4 v0, 5
const/16 v1, 10
add-int v2, v0, v1
invoke-virtual {v3, v2}, Activity;->setter(I)V
return-void
```

Listing 1. Sequence of Dalvik opcodes (smali code).

```
1250
1302 0a00
9002 0001
6e20 cc00 2300
0e00
```

Listing 2. Sequence of Dalvik opcodes (hex code).

To extract opcode features efficiently, we deal with the Dex code as follows:
(i) Dex file is a stream of Hex code. Hence, we operate directly on the sequence of hex stream (avoiding string operations).
(ii) We read Dex files one-by-one (in case of MultiDex App) from an APK and operate only in-memory (avoiding file system operation).

Listing 2 shows the Dalvik opcode sequence in the hex stream (little-endian format), equivalent to the smali code shown in listing 1. Compared to the smali code (listing 1), the hex stream (listing 2) does not contain any string. Hence, it does not require time-consuming string operations to extract opcode information (highlighted in blue color in both listings 1 and 2).

V. FEATURE ENGINEERING

A model built upon gathering as much information as possible helps the classifier to learn more. However, the computation capability and processing time required to build such models are enormous. Such models are not feasible to deploy on a real device to detect malware. Therefore, we need a mechanism such that, a machine learning model can be trained using less number of features, while maintaining good detection accuracy.

Often most of the features are usually irrelevant or redundant and increase the model complexity. Therefore, we consider only the essential and relevant features. Feature selection/reduction methods are often used to solve such problems. Fig. 1 shows the flow of the feature engineering module. This module has three major phases, which are discussed in subsequent sub-sections.

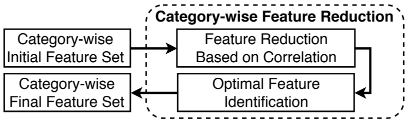

Category-wise Feature Reduction
Category-wise Initial Feature Set
Feature Reduction Based on Correlation
Category-wise Final Feature Set
Optimal Feature Identification

Fig. 2. Category-wise feature reduction process.

A. Feature Selection and Encoding

Android App developers can provide any name to custom permissions, services, and activities, etc. Such user-defined components have a massive number of features due to user-defined names. Despite having a large number of binary-valued features in each set for components, they do not show good prediction abilities concerning the number of features available in each feature set. Hence, in the quest of reducing the feature set, one should carefully handle loss of information. Therefore, instead of eliminating these binary-valued features, we transform them by maintaining a count of a component in each category. After the transformation, each of these set holds the frequency of their usage for an App. Another change can take place in requested permissions where we keep Android defined permissions as binary-valued features and make the count of remaining permissions (custom permission). Note that, we did not perform any transformation to the features extracted from the Dex code. As it is already done by reducing the instruction as discussed in §IV-A. We only use frequency of 2-Gram opcode sequence extracted from every function inside the Dex code.

Finally, after transformation and encoding of the extracted features, we have feature sets that contain either binary value or the frequency of their usage. The **Encoding** column in Table II shows the resulting set of features after this step.

B. Category-wise Feature Reduction

Most of the categorical sets individually have a lot of predictive powers. Hence, we optimize the performance of each of the binary-valued feature set and 2-Gram opcode sequence, individually and then combine their predictive power to build a better model.

Category-wise feature reduction involves two processes for feature reduction, as shown in Fig. 2. Both the binary-valued feature sets (see §V-A) and opcode sequence are passed through each of these processes to filter out the redundant and irrelevant features in each category. All the processes involved in the category-wise feature reduction are elaborated as follows.

Feature Reduction Based on Correlation: Often a dataset contains some features that are highly correlated with each other and provide the same information. Keeping all such features (correlated features) increases the complexity of the model without contributing towards classification efficiency. This step addresses the issue by finding all the correlated features. Correlation between two features is performed based on the Pearson's correlation coefficient [23] that lies between -1 to 1. Pearson value closer to 0 denotes weak correlation whereas value closer to 1 and -1 implies strong positive and negative correlations, respectively. In both the strong correlations (i.e. positive and negative), consideration of only one feature can reduce the feature set size while retaining the effectiveness of the model.

TABLE III
EFFECT OF PEARSON COEFFICIENT THRESHOLD (COR_t) ON ACCURACY (ACC) AND #FEATURES (#FEAT).

$\mathbf{COR_t}$	**Acc(%) / #Feat**		
	ReqP	**HWC**	**2-Opc**
0.5	93.69 / 533	60.58 / 194	86.32 / 39
0.6	94.00 / 562	60.79 / 207	90.05 / 55
0.7	93.99 / 575	60.82 / 212	94.82 / 72
0.8	**94.74 / 602**	**60.80 / 224**	95.50 / 104
0.9	94.86 / 626	60.79 / 226	**95.99 / 172**
1.0	94.89 / 668	60.85 / 245	96.28 / 317

Note: ReqP=Requested Permissions set, HWC=Hardware Component set, 2-Opc=2-Gram opcode sequence

TABLE IV
TERMINOLOGY USED IN FEATURE ENGINEERING.

Term	**Description**
COR_t	Threshold for eliminating the correlated values.
Acc	It represents the accuracy of a detection model.
Acc_R	Highest accuracy given by the RFECV [24].
$Feat_R$	Optimal #features used by RFECV to achieve accuracy Acc_R.
RFE_t	It is a penalty over the highest accuracy Acc_R while selecting less #feature in place of optimal #feature $Feat_R$.
$Feat_C$	Chosen #Feature with penalty RFE_t over Acc_R.
Acc_C	Accuracy achieved while taking only $Feat_C$ features.
Pre	Denotes the precision at which detection model can operate.
Rec	Denotes the malware detection rate (recall) of a detection model.

By this analogy, we apply different Pearson correlation values as a threshold in both the directions (i.e., negative and positive) to filter out highly correlated features. The effect of correlation value (referred to as COR_t[1]) for different threshold (0.5 to 1.0) against the accuracy[2] of the detection model has been shown in Table III. The COR_t value 1.0 represents the original features without any reduction process. The results shown in Table III are obtained by evaluating a RandomForest model on the training set (see §VII) with 10-fold cross-validation. We use a threshold of 0.8 for requested permissions and hardware components while 0.9 for 2-Gram opcode (highlighted in Table III) as it results in the correct tradeoff between accuracy and number of features (significant reduction in feature set size with minimum loss in accuracy). Note that, we use similar methods (using training set) for the remaining stages of feature engineering.

Optimal Feature Identification: This process utilizes RFECV (recursive feature elimination with cross validation) [24] to identify the optimal feature set. RFECV uses a feature ranking method and selects the best features that contributes significantly in solving the desired problem. We provide the classifier C as RandomForest, ranking function F as accuracy and the number of features N (set to 1) as input to RFECV for feature elimination. As a result, RFECV provides a grid of score and the set of optimal features that gives highest accuracy (see Fig. 3). Corresponding to requested permissions, hardware

[1] Summary of notations in Table IV used in the remaining sections.

[2] Accuracy represents the number of samples (in percentage) correctly classified.

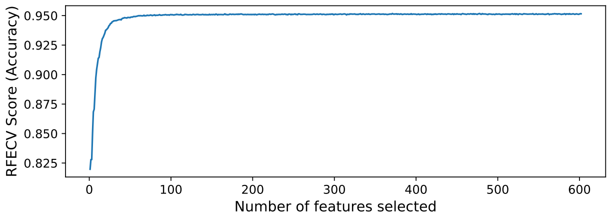


Fig. 3. Optimal #features Vs accuracy graphs for requested permissions.

components and 2-Gram opcode sequence, the optimal number of features selected with highest accuracy by the RFECV is 371, 185 and 169, respectively, which are still a lot of features. *However, if we carefully observe Fig. 3, we find that with a significantly less number of features result in an accuracy very close to the maximum achievable accuracy.* With this observation, we define a threshold RFE_t, which is a penalty in choosing less number of features in terms of accuracy.

For example, let the highest accuracy given by RFECV is Acc_R with optimal number of features $Feat_R$. However, we observe $Feat_R$ is still large and can be further reduced finding a sweet spots without significantly compromising on accuracy. Let the chosen accuracy from RFECV grid score be denoted by Acc_C and the corresponding number of features as $Feat_C$. Then the relation between the threshold RFE_t, highest accuracy Acc_R and the chosen accuracy Acc_C is shown in (1).

$$Acc_R - Acc_C \leq RFE_t \quad \text{and} \quad Feat_C < Feat_R \qquad (1)$$

The effect of the RFE_t with varying values from 0.0 to 0.5 is shown in Table V where 0.0 denotes the RFECV score with highest accuracy and the threshold is the difference between Acc_R and Acc_C in percentage. As shown in Table V, the changes in threshold RFE_t drastically reduces the feature set size while maintaining acceptable accuracy score. In this step, we have selected 0.5 as the RFE_t value where a drastic reduction in the feature set size can be observed. The remaining relevant features contributes effectively towards solving the desired problem. However, we do not know the features involved to achieve the same results except the count of those features. To extract the required features, we have used Recursive Feature Elimination (RFE) [25] that takes a classifier and the number of features we want to select (as information retrieved using RFECV and threshold RFE_t) as input to obtain the list of features without impacting the accuracy.

C. Feature Reduction from Combined Feature Set.

In §V-B, we have performed category-wise feature reduction where binary features from two categories (requested permissions and hardware components) along with the frequency of 2-Gram opcode sequence are involved. However, the original feature set contains three types of features viz. binary features, 2-Gram opcode sequence and the numeric features. This section combines all features and performs a feature reduction in the combined feature set. This process includes two steps — *(i)* combining the features and selecting the best combination of feature set and *(ii)* feature reduction on the selected combined feature set.

TABLE V
EFFECT OF RFE_t THRESHOLD ON ACCURACY (ACC) AND #FEATURES.

$\mathbf{RFE_t}$	**Acc(%) / #Features**		
	ReqP	**HWC**	**2-Opc**
0.0	94.77 / 371	60.79 / 185	96.01 / 169
0.1	94.76 / 86	60.75 / 17	95.92 / 69
0.2	94.68 / 60	60.72 / 13	95.90 / 62
0.3	94.57 / 52	60.65 / 13	95.76 / 48
0.4	94.47 / 41	60.62 / 13	95.76 / 37
0.5	**94.34 / 40**	**60.60 / 12**	**95.64 / 30**

Note: ReqP=Requested Permissions set, HWC=Hardware Component set, 2-Opc=2-Gram opcode sequence

TABLE VI
EFFECT OF COMBINING DIFFERENT FEATURE SET.

Combination	**#Features**	**Acc (%)**	**Pre (%)**	**Rec (%)**
Num+HC	18	86.45	86.55	85.46
Num+OP	36	96.90	96.93	96.90
HC+OP	42	96.07	96.19	96.07
Num+RP	46	96.45	96.46	96.24
Num+HC+OP	48	96.87	96.89	96.87
RP+HC	52	95.16	95.08	94.96
Num+RP+HC	58	96.57	96.59	96.37
RP+OP	**70**	**98.15**	**98.15**	**98.15**
Num+RP+OP	**76**	**98.14**	**98.15**	**98.14**
RP+HC+OP	82	98.12	98.12	98.12
Num+RP+HC+OP	88	98.12	98.12	98.12

Note: RP=Reduced Requested Permissions set, HC=Reduced Hardware Component set, Num=Numeric Feature (features for that we have taken frequency of their usage except n-Gram features), OP=Reduced 2-Gram Opcode Sequence, Acc=Accuracy, Pre=Precision, Rec=Recall

Combining Feature Set: In §V-A, we have selected obfuscation resilient features divided in four sets—one set corresponding to numerical features, one related to the opcode sequence and the remaining two sets related to the binary features. Using four different feature sets, we combine all the features in different combinations and select the best combination among them. In total, there are 11 unique combinations, which are possible for four sets. We have trained a RandomForest classifier on these combinations and the result for them is summarized in Table VI where performance of all the combinations are enlisted in three evaluation metrices—*(i)* accuracy (Acc), *(ii)* precision[1] (Pre), and *(iii)* recall[2] (Rec). As shown in Table VI, the hardware components feature set do not contribute towards accuracy in a significant manner; neither individually nor by combining with other features. If we compare detection results with two different combinations with high accuracy—*(i)* numeric feature combined with requested permissions & opcode sequence (76 features), and *(ii)* combining requested permission with opcode sequence only (70 features), the results indicates that the performance of both the sets are almost same but differs in the number of features. One could simply select the feature set which contains less features in this case combination *(ii)*. However, in place of combination *(ii)*, we select combination *(i)*, because the contribution of

[1] Precision denotes the fraction of malware correctly detected.

[2] Recall represents the malware detection rate for a model.

TABLE VII
ELIMINATION OF FEATURE FROM COMBINED FEATURE SET.

Feature Set	#Feature	Acc (%)	Pre (%)	Rec (%)
Num+RP+OP	76	98.14	98.15	98.14
Num+RP+OP-I	**75**	**98.18**	**98.18**	**98.18**
Num+RP+OP-C	75	98.08	98.08	98.08
Num+RP+OP-I-C	74	98.13	98.13	98.13

Note: RP=Reduced Requested Permissions set, Num=Numeric Feature, OP=Reduced 2-Gram Opcode Sequence, I=Intents Filter, C=Custom Permissions, Acc=Accuracy, Pre=Precision, Rec=Recall

numeric feature set is significant when it is combined with the requested permissions (see Table VI). Therefore, we select combination of Numeric feature, requested permissions and opcode sequence for the next phase of reduction. Note that, here selection of combination is based on the analyst point of view as both combination perform almost equally.

Feature Reduction in Selected Feature Set: In this step, we perform final optimization on the feature set of the selected combination. The aim of this optimization is to eliminate some features that increases the processing time and requires extra support for extraction. In the selected feature set, such features are Intent Filter (referred to as I) and Custom Permissions (referred to as C). We observe the effect on accuracy, precision, and recall (see Table VII) when eliminating either one or both features from the feature set. From Table VII, we find that the elimination of Custom Permissions (C) significantly impacts the accuracy of malware detection (recall). On the other hand, the elimination of the Intent Filter does not affect the detection rate of the model. After exclusion of Intent Filter, the resulted feature set (of size 75) is used to learn the final detection model.

VI. DEEPDETECT: BUILDING THE SYSTEM

So far, we have extracted the features from the dataset (see §IV) and selected the most relevant features (see §V) to design an on-device malware detector. In this section, we first provide an overview of DeepDetect followed by the off-device training process of the machine learning model and porting it for mobile devices to detect malware on real devices.

A. Overview

In DeepDetect, we train a machine learning model on a server machine. Firstly, we extract the static features from the Manifest file and Dex code (see §IV). These extracted features are then passed to the feature engineering process. In feature engineering (see §V), we first eliminate the effect of obfuscation with the help of transformation and then reduce the size of feature dimension by applying a multilevel feature selection/reduction process. At last, a detection model is learned and embedded into an App for on-device detection. We describe learning the malware detection model and detecting malware on a real device in §VI-B and §VI-C, respectively.

B. Learning Model

The final feature set obtained from §V-C contains obfuscation-resilient features, and obfuscation techniques used by the malware writer cannot affect the prediction ability of a model built upon them. As our primary goal is to detect malware on-device, the classifier's choice for the final detection model should be lightweight. Keeping this in mind, we use `TensorFlow` [26] library (developed by Google) to build the final model. Google also provides a light version of `TensorFlow`, i.e., `TensorFlow Lite`, which is designed for mobile devices. We use TensorForest [27] to learn final detection model. We use training set for learning the final model and serialize the model into a file. The saved model occupies 869 KB of space in the system. The saved model needs to be converted into the `.tflite` format for direct use with `TensorFlow Lite`. The `TensorFlow Lite` converter has been used to convert the learned output so that we can use it in an Android device to detect malware seamlessly. At last, we obtain a final `TensorFlow Lite` model of size ∼150 KB only. The final detection model is provided to the DeepDetect along with the features used in training to detect malware on-device.

Algorithm 1: Feature Vector Generation

```
Input  : $List_{pkgs}$, $Model_{perm}$, $Model_{2-opc}$
Output : $Vector_{feat}$
$Vector_{feat} \leftarrow \phi$ // Vector of features
$N_{act} \leftarrow ActivitiesCount(PM, List_{pkgs})$
$N_{serv} \leftarrow ServicesCount(PM, List_{pkgs})$
$N_{recv} \leftarrow ReceversCount(PM, List_{pkgs})$
$N_{prov} \leftarrow ProvidersCount(PM, List_{pkgs})$
$List_{perm} \leftarrow Permissions(PM, List_{pkgs})$
$Freq_{2-opc} \leftarrow GetTwoGramOpcodeFreq(List_{pkgs})$
$N_{CustPerm} \leftarrow CustPermCount(List_{perm})$
$append(Vector_{feat}, (N_{act}, N_{serv}, N_{recv}, N_{prov}, N_{CustPerm}))$
foreach $perm$ in $Model_{perm}$ do
    if $perm$ is in $List_{perm}$ then
        $bit \leftarrow 1$
    else
        $bit \leftarrow 0$
    $append(Vector_{feat}, bit)$
foreach $twoOP$ in $Model_{2-opc}$ do
    if $twoOP$ is in $Freq_{2-opc}$ then
        $count \leftarrow get(Freq_{2-opc}, twoOP)$
    else
        $count \leftarrow 0$
return $Vector_{feat}$
```

C. On-device Detection

To detect malware on a real device, we require feature vector from an APK, so that we can pass it to the already trained model (see §VI-B) for the detection result.

The procedure for extracting the feature from a list of App(s) (single App as well as multiple Apps) and embedding them into the feature vector is shown in Algorithm 1 by utilizing the Package Manager (referred to as PM), customised opcode information module (referred to as $GetTwoGramOpcodeFreq$ designed using the process discussed in §IV-B). Algorithm 1 takes the list of the Apps in terms of their package names (referred to as $List_{pkgs}$), list of requested permission (referred to as $Model_{perm}$) and the list of selected 2-gram opcode sequence (referred to as $Model_{2-opc}$) as input. In Algorithm 1, features are extracted from Apps by querying PM and custom opcode information

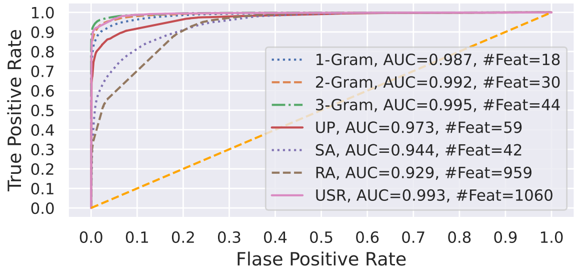


Fig. 4. AUC-ROC curve for the model build on various feature extracted from Dex file. UP: Used Permission, SA: Sensitive APIs, RA: Restricted APIs and USR: Combined features (UP, SA and RA). #Feat: Number of Features.

extractor (line 2 to 7), number of custom permissions are filtered (line 8), and all features are encoded into the feature vector (line 9 to 20). Note that, requested permissions that do not start with "android.permission" are called custom permissions. The feature vector obtained using Algorithm 1 is then passed to the detection model (see §VI-B). The detection model analyses the extracted features and provides the result in terms of a binary answer where a `true` implies malware and a `false` implies benign. If the targeted App is flagged as malware, the same is notified to the user. DeepDetect also provides an option to uninstall the App from the device.

VII. Evaluation

To evaluate the effectiveness of DeepDetect in terms of malware detection accuracy and runtime performance, we have separated 20% samples (referred to as evaluation set) from the dataset described in §III and assumed them to be unseen Apps. In all the experiments, we train every model using the remaining 80% samples (referred to as training set) of the dataset, which has samples till April 2018. *Note: only runtime performance experiments (§VII-E) are performed on real smartphones to measure execution time and device energy consumption. Rest of the experiments are performed on a server machine.* This section answers the following research question to evaluate the proposed detection system effectively:
***(i)* Robustness against known, unseen, and new samples** (§VII-B): Does DeepDetect maintains its prediction capability against known, unseen, and new samples?
***(ii)* Impact of obfuscation** (§VII-C): What is the impact of obfuscation on the malware detection rate?
***(iii)* Effect of experimental biases** (§VII-D): How DeepDetect performs after eliminating potential experimental biases?
***(iv)* Run-time overhead** (§VII-E): Does 2-Gram opcode sequence features consume less device energy as compared to other features from Dex code?

A. Performance Comparison of Features

In this experiment, we extract seven categories of features from the Dex code—*(i)* 1-Gram sequence of opcode, *(ii)* 2-Gram, *(iii)* 3-Gram, *(iv)* Used Permissions (UP), *(v)* Suspicious APIs (SA), *(vi)* Restricted APIs (RA), and *(vii)* combined features from UP, SA, RA (referred to as USR). To compare performance of these feature sets, we train a RandomForest model on the training set and evaluated against the evaluation set. Fig. 4 shows the Receiver Operating Characteristic (ROC) curves obtained from the evaluation results of the trained model along with the value of AUC (Area Under the Curve) and the number of features (#Feat) in a feature set. ROC curve represents the relationship between the true positive rate (TPR), and false-positive rate (FPR). The AUC measures the ability of a classifier (model) to distinguish between classes and is generally used as a summary of the ROC curve. *The higher the AUC, the better the model's performance at distinguishing between the positive and negative classes.* As shown in Fig. 4, high AUC scores are observed for 2-Gram, 3-Gram, and USR (more than 99%). However, the number of features in USR is relatively large as compared to 2-Gram and 3-Gram. Also, the device energy consumption and time required to extract features are also large (see §VII-E). Hence, USR is not a good choice of features to design an on-device malware detector. Therefore, from the remaining two feature sets, any one can be used for on-device malware detection. However, we select 2-Gram because it consumes ∼1.4X less device energy as compared to 3-Gram. *Note: The main aim of this experiment (and also experiment in §VII-E) is to select best features that can be efficiently extracted from the Dex code and combined with features extracted from the Manifest file.*

TABLE VIII
Evaluation of final model with known, unseen, new and Pegasus samples.

Dataset	Pre (%)	Rec (%)	F1 (%)	FPR (%)
Training Set	99.98	99.95	99.95	0.01
Evaluation Set	98.05	97.50	97.69	1.51
AndroZoo-2019	97.70	97.12	97.69	1.73
Pegasus (5 Sample)	–	100	–	–

B. Performance Against Known, Unseen, and New Samples

To show the effectiveness of DeepDetect in identifying unseen samples, we use the evaluation set. We have collected 10760 new samples from AndroZoo [13] (referred to as AndroZoo-2019) where the Dex file date is of the year 2019. In AndroZoo-2019, 5380 samples are malware, and rest of them are benign. Apart from the AndroZoo-2019, we have collected five samples of Pegasus malware from the CloudSek organization. For the known samples, we have used the same samples used for training the final model. Table VIII summarizes the evaluation results for the known (training set), unseen (evaluation set), new (AndroZoo-2019), and Pegasus malware samples with four evaluation metrices–*(i)* F1-score[1] (referred to as F1), *(ii)* precision, *(iii)* recall, and *(iv)* false-positive rate (referred to as FPR). When the model is evaluated against the known samples, the model correctly classifies 99.90% malware and generates 0.01% of false alarms. For the unseen samples, our detection model correctly detects 97.50% malware with a false positive rate of 1.51%. In the presence of new samples, our model detects 97.12% of new malware while generating

[1] F1-score represents the weighted average of recall and precision.

TABLE IX
OBFUSCATORS IMPLEMENTED IN OBFUSCAPK [28] TOOL.

Category	Obfuscators
trivial	Randomize Manifest file, Rebuild, New Alignment, Re-signing
renaming	Renaming the Class, Fields and Method
encryption	Encryption of Library, resource strings, Assets, and constant strings
reflection	Invoke user defined and framework APIs using the reflection APIs
code	Junk code insertion, instruction re-ordering, calls redirection, removing debug data, insertion of goto instruction, adding new method by exploiting method overloading.

TABLE X
EVALUATION OF FINAL MODEL AGAINST OBFUSCATED MALWARE.

Category	#Samples	#Sample Detected		Detection rate drop (%)
		Original	Obfuscated	
trivial	160	156	156	0
renaming	570	554	554	0
encryption	1135	1102	1096	0.53
reflection	252	241	239	0.79
code	2429	2358	2298	2.47
mix	447	438	429	2.01
Overall	**4993**	**4849**	**4772**	**1.55**

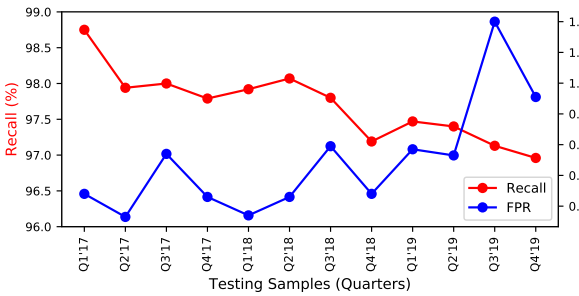


Fig. 5. Detection results after removing experimental bias (Space and Time).

1.73% of false alarms. Interestingly, our model is able to detect all the Pegasus malware samples. Maybe this is possible because Pegasus samples are pre-2019. In comparison to the state-of-the-art on-device detector (Drebin [4]), DeepDetect detects ∼3.5% more malware (unseen malware) using only 75 features, whereas Drebin uses 0.5 million features with a malware detection rate of ∼94%. If we compare DeepDetect with the recent on-device malware detector IntelliAV [5], DeepDetect outperforms in both type of samples, i.e., known and unseen/new samples. DeepDetect's malware detection rate in case of known samples is 0.15% more as compare to IntelliAV. In the presence of unseen/new samples, the detection rate of IntelliAV is still less than the best state-of-the-art detector Drebin. Further, we have analysed the importance of features in differentiating a malware from the benign. In our analysis we found that GET_ACCOUNT requested permission and SB (Bit-wise operation followed by branch instruction) opcode sequence play a major role. For more details about the list of features, their importance, more experiment (with other classifiers and performance metrics) along with more details about the used datasets, please refer to the weblink: https://skmtr1.github.io/DeepDetect.html.

C. Evaluation Against Obfuscated Malware

In order to evaluate our model against the obfuscated samples, we require an obfuscated malware dataset. One such dataset is the PRAGuard [6] which was available when we performed the experiments. The PRAGuard dataset contains 10479 samples obtained by obfuscating the MalGenome [29] and the Contagio Minidump [30] datasets. As this dataset is old and contains samples till March 2013, there may be overlapping samples with our training dataset collected in April 2018. Therefore, to avoid inclusion of known obfuscated malware samples in the training data, we created a new obfuscated malware dataset by obfuscating the malware samples downloaded from the AndroZoo for the year 2019 (5380 samples). To obfuscate malware samples, we have utilized the Obfuscapk [28] tool.

Obfuscapk is an open-source black-box obfuscation tool for Android App. Obfuscation techniques of Obfuscapk are classified into five categories, which are shown in Table IX. Out of the 5380 malware samples, we have successfully obtained 4993 obfuscated samples in six categories. Five categories are the same as provided by the Obfuscapk, whereas the sixth category comprises a mix of two or more obfuscation techniques (referred to as mix). The number of obfuscated samples in each category and their evaluation with our detection model are shown in Table X. We have also evaluated the same non-obfuscated samples in each category against our detection model. The number of samples detected in non-obfuscated and obfuscated samples are shown in the **Original** and **Obfuscated** columns of Table X, respectively. *Note: We have trained our model on the training set only, which contains samples till April 2018.*

The results shown in Table X indicate that the detection rate of DeepDetect does not go down for trivial and renaming obfuscation techniques. The main reason behind this is the feature encoding, where we take count of user-defined similar entities like activities. However, we observe some drop in detection rate for other categories. We see a maximum drop in detection rate (up to 2.47%) for code obfuscation techniques. In general, our detection model can detect 95.57% of malware, whereas, for the same non-obfuscated malware sample, it achieves a 97.12% of detection rate. Therefore, the overall drop in malware detection rate in the presence of obfuscated samples is 1.55%.

D. Evaluation After Elimination of Experimental Biases Across Space and Time

There are many Android malware detectors ([4], [9]–[11], [31]) available that publishes high detection results up to 99%. However, there are two potential experimental biases in the experiment (shown by Tesseract [32])–*(i)* Spatial bias and *(ii)* Temporal bias. Spatial bias occurs due to the incorrect distribution of malware and benign samples in the dataset, whereas temporal bias is caused by the incorrect time splits of training and testing samples. DeepDetect evaluation is free from the temporal bias when evaluated against AndroZoo-2019 and Obfuscated samples, but spatial bias is still present.

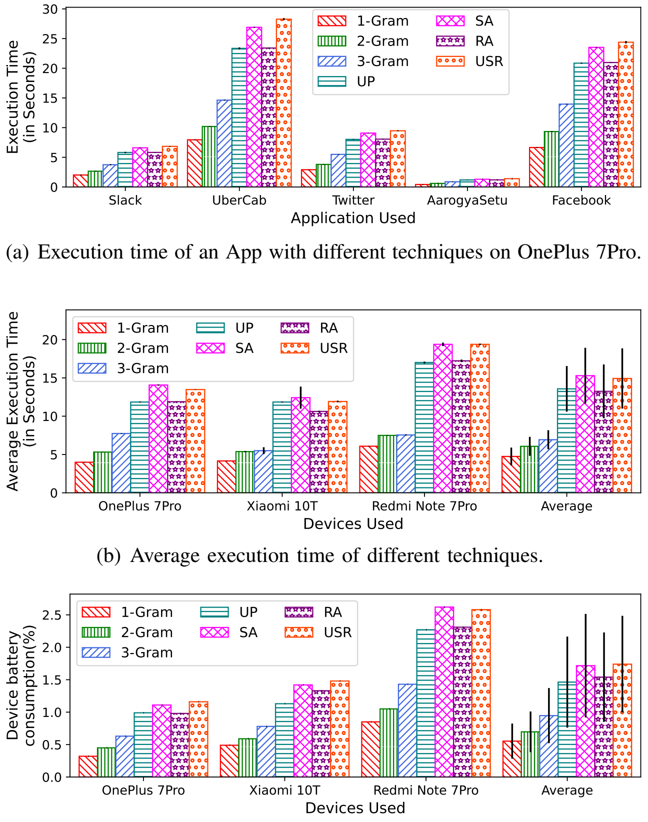
TABLE XI
ANDROID APPS USED FOR RUN-TIME PERFORMANCE AND DEVICE ENERGY CONSUMPTION.

App Name	#Dex Files	Size (MBs)	
		APK	Dex Files
Slack	3	61	24.6
UberCab	15	50	116.6
Twitter	5	19	33.6
AarogyaSetu	1	4.3	5
Facebook	11	55	81.2

Therefore, to evaluate appropriately against potential biases, we have downloaded 87,634 (with ∼10% malware) unique samples from AndroZoo spanning for four years (2016 to 2019). We train the DeepDetect model on the sample from year 2016 and test it quarter-wise against the years 2017, 2018, and 2019. Samples in each quarter contain ∼10% malware, and the remaining are benign. The evaluation result (see Fig. 5) shows that DeepDetect can detect ∼97% of malware while generating ∼1.4% false alarms when evaluated after elimination of experimental biases across space and time. It indicates that our model is also robust against experimental biases. However, when Tesseract evaluated Drebin and MaMaDroid [31] against potential biases, the performance decreased up to 50%.

E. Runtime Efficiency

To measure the runtime efficiency (execution time and energy consumption), we have used five Android Apps—Slack, UberCab, Twitter, AarogyaSetu, and Facebook. The selection of these Apps is based on their size and the number of Dex files used (SingleDex or MultiDex file App, see Table XI). We analyse these Apps on three different mobile devices—OnePlus 7Pro, Xiaomi 10T, and Redmi Note 7Pro. We execute each App ten times on each device and log the time taken to extract different features used in §VII-A and the device energy consumed by these methods (see Fig. 6). To obtain the battery utilization, we have used `dumpsys` utility through ADB shell and analyzed using the `battery-historian` tool. Fig. 6(a) shows the average time spend to extract features from an individual App executed on OnePlus 7Pro, which is obtained by taking the average of all the execution time (ten runs), whereas Fig. 6(b) denotes the average time taken to analyse all Apps on different devices. The energy consumption result (see Fig. 6(c)) shows battery utilization in analyzing these Apps ten times. For the OnePlus 7Pro device (Fig. 6(a)), the result shows that the 2-Gram feature set takes ∼5.32 seconds, which is 2.23X and 2.53X faster than the RA and USR feature set, respectively. The feature extraction time depends on the Dex file size and not on the size of APK because an APK also contains other resources and files like images, native code, etc. With respect to device battery consumption (see Fig. 6(c)), the 2-Gram approach also outperforms all the other methods that do not use opcode information and improves the device energy consumption by more than 2.1X (consumes only 0.45% of total device battery). However, the average execution time and energy consumption of all the devices (averaging the estimation of all the devices) for the 2-Gram feature set are 6.06 seconds and 0.7%, respectively. For the analysis of individual App execution time on other mobile devices, please refer to the weblink: https://skmtr1.github.io/DeepDetect.html.



(a) Execution time of an App with different techniques on OnePlus 7Pro.

(b) Average execution time of different techniques.

(c) Device energy estimation.

Fig. 6. Estimation of feature extraction time and device battery consumption of (i) 1-Gram, (ii) 2-Gram, (iii) 3-Gram, (iv) Used Permissions (UP), (v) Suspicious APIs (SA), (vi) Restricted APIs (RA), and (vii) USR (Combined UP, SA and RA).

F. Discussion and Limitations

Even though DeepDetect provides a strong defense system to detect unseen malware on a real device with a malware detection rate of more than 97.5%, it also has the following limitation:

(i) Currently, it is designed to work as a third-party App for detecting Android malware on an actual device. Hence it cannot stop an App from being installed on a device. To overcome this limitation, a device vendor or AOSP project can include it in their core system, and the installation of an App should start after getting a clean chit from DeepDetect.

(ii) It cannot detect packed malware where entire code is encrypted except for the unpacking logic.

(iii) As DeepDetect uses information from Manifest file and Dex code (opcode) to detect malware, it cannot detect malware that include malicious behaviour exclusively in the native code.

(iv) Static analysis based detection systems fail to detect malware that downloads malicious code from the external source and execute it at runtime. This is also true with DeepDetect. However, if a malware dynamically loads a code already present inside the APK and is not encrypted, DeepDetect can detect such malware efficiently as we extract opcode information from all the Dex file present inside an APK.

Evaluation summary: In a nutshell, DeepDetect can effectively detect malware and maintain its detection capability against the obfuscated samples with significantly low processing time when deployed on the device.

VIII. Related Work

With the rapid growth in Android malware, various defense systems have evolved to fight malware. Existing defense mechanisms [4], [5], [8], [12], [31], [33]–[41] use static, dynamic, or a combination of both analysis techniques to analyze Apps [42]. These methods can be deployed either on market place in an offline manner or on a real device. Since dynamic analysis is costly and requires significantly higher processing power, deploying the same techniques on a real device is impractical. For example, Malton [39] is an on-device dynamic malware analysis system that monitors an App on each layer of Android OS. For Malton, the maximum slowdown for different Java operations is ∼36X while observing taint propagation. This solution is good for the analysis environment but not suitable for the end-users devices. Therefore, we restrict our discussion to the static analysis based on-device malware detection.

With the consideration of low-end mobile devices, some on-device malware detectors [4], [5], [7], [8], [37] have been proposed and works only on static features. Drebin [4] extracts features from the Dex code and Manifest file with a detection rate of 94%. IntelliAV [5] uses framework level API along with the features extracted from the Manifest file. IntelliAV shows a high prediction rate of more than 99% in the case of known samples (training sample), while the detection rate falls to ∼72% for unseen samples. However, the average analysis times on real devices for Drebin and IntelliAV are 10 seconds and 3.5 seconds, respectively. Directly comparing the execution time of DeepDetect with Drebin and IntelliAV is not a good approach because the average App size has quintupled [43] every year. Talos [7] uses only requested permissions and trains a deep learning model while achieving an accuracy of more than 93% and takes negligible time to analyze an App (in milliseconds). DeepDetect analysis time is ∼5.32 seconds, which is more than the Talos, but DeepDetect's malware detection rate is more than 97% for unseen malware, which is significantly high compared to Talos, Drebin, and IntelliAV.

Mercaldo et al. [37] have also proposed an on-device malware detector using 1-Gram opcode sequence only, where they utilize six opcodes (details can be seen in [37]). Our experiment shows (see §VII-E) that 1-Gram opcode's device energy consumption and average execution time for feature extraction are relatively less than the 2-Gram opcode sequence. However, the malware detection accuracy is good for the 2-Gram method compared to that of 1-Gram. Furthermore, utilizing only single-source information to design a malware detector is not a good approach where the single source is a Dex file. Therefore in our solution, we also include information from the Manifest file in the feature set.

Similarly, Yuan et al. [8] use API call information, permissions and intent filters as the feature set. As shown in Fig. 4, API-based information is not good to detect malware as compared to Opcode because the API calls are the most susceptible to obfuscation attacks. However, this work is benefited from the on-device training to train a model incrementally to learn more malicious behavior. Even though on-device training is a good approach, an end-user device does not see a variety of samples in the live environment, which is a core requirement to learn different behavior.

Some other on-device malware detectors [38], [44] are also proposed using dynamic analysis or a mix of static and dynamic techniques (also known as hybrid analysis). Sinha et al. [44] insert instrumentation information into an App with the help of Dynalog [45] and then execute the modified App on an actual device. This method requires modification in an App due which the integrity of the App is compromised which can be exploited by malware to bypass dynamic analysis.

Similarly, BRIDEMAID [38] uses hybrid techniques to detect malware on-device. For the static analysis, BRIDEMAID uses the n-Gram opcode sequence as a feature set (similar to our approach), whereas they rely on a kernel-level modification (kernel module) for the collection of dynamic information (adapting the MADAM [46] solution). As the collection of dynamic information requires modification in the Android kernel, BRIDEMAID requires rooting of the device or support from the device vendor/AOSP.

IX. Conclusion

As Android's official market place (Play Store) itself is not free from malware, we have proposed DeepDetect: an on-device malware detector that is capable of detecting malware on a real device. In DeepDetect, we have designed a feature engineering framework with the aim of reducing the feature set size by finding the right tradeoff of feature set size and detection accuracy. We have shown the effectiveness of the feature engineering framework by reducing the feature set size from 712K to 75 features that are free from the impact of code obfuscation. We have performed experiments to demonstrate the effectiveness of DeepDetect against the known, unseen, and obfuscated malware samples, and shown that DeepDetect can effectively detect more than 97% new malware with an FPR of 1.73%. For the obfuscated malware, DeepDetect achieves a malware detection rate of 95.57%. However, the malware detection rate of DeepDetect for known malware is 99.90%, with an FPR of 0.01%. Additionally, DeepDetect performance is not impacted significantly when evaluated without the spatial and temporal biases, and achieves malware detection rate of ∼97%. Finally, we have shown that when DeepDetect deployed on a real device, it can analyze an application in ∼5.32 seconds on an average, which is 2.23X faster than API based malware detector, while consuming 0.45% (for 50 Apps) of total device battery.

REFERENCES

[1] (2020) IDC: Smartphone Market Share-OS. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] (2020) G DATA Mobile Malware Report 2019: new high for malicious android apps. [Online]. Available: https://www.gdatasoftware.com/news/g-data-mobile-malware-report-2019-new-high-for-malicious-android-apps

[3] A. Orlowski. (2019) Google play store spews malware onto 9 million 'Droids. [Online]. Available: https://www.theregister.co.uk/2019/01/09/google_play_store_malware_onto_9m_droids/

[4] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Symposium on Network and Distributed System Security (NDSS)*, 02 2014.

[5] M. Ahmadi, A. Sotgiu, and G. Giacinto, "IntelliAV: toward the feasibility of building intelligent anti-malware on android devices," in *Machine Learning and Knowledge Extraction*. Springer International Publishing, 2017, pp. 137–154.

[6] Maiorca et al., "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16 – 31, 2015.

[7] H. C. Takawale and A. Thakur, "Talos App: on-device machine learning using tensorflow to detect android malware," in *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*, 2018, pp. 250–255.

[8] W. Yuan, Y. Jiang, H. Li, and M. Cai, "A lightweight on-device detection method for android malware," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–12, 2019.

[9] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, Jan. 2018.

[10] S. Jaiswal, "Feature engineering & analysis towards temporally robust detection of android malware," Master's thesis, Indian Institute of Technology, Kanpur, 2019.

[11] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: fast and accurate classification of obfuscated android malware," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, p. 309–320.

[12] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," in *Security and Privacy in Communication Networks*. Springer International Publishing, 2013, pp. 86–103.

[13] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, p. 468–471.

[14] S. Kumar and S. K. Shukla, *The State of Android Security*. Springer Singapore, 2020, pp. 17–22.

[15] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2017, pp. 252–276.

[16] VirusShare. (2018). [Online]. Available: https://virusshare.com/

[17] VirusTotal. (2018). [Online]. Available: https://www.virustotal.com/

[18] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, Jan. 2017.

[19] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Comput. Linguist.*, vol. 18, no. 4, p. 467–479, Dec. 1992.

[20] H. DONG, N. qiang HE, G. HU, Q. LI, and M. ZHANG, "Malware detection method of android application based on simplification instructions," *The Journal of China Universities of Posts and Telecommunications*, vol. 21, pp. 94–100, 2014.

[21] T. Chen, Q. Mao, Y. Yang, M. Lv, and J. Zhu, "TinyDroid: a lightweight and efficient model for android malware detection and classification," *Mobile Information Systems*, vol. 2018, pp. 1–9, 2018.

[22] Desnos et al. (2019) Welcome to Androguard's documentation! - androguard 3.3.5 documentation. [Online]. Available: https://androguard.readthedocs.io/en/latest/

[23] W. Kirch, Ed., *Pearson's Correlation Coefficient*. Springer Netherlands, 2008, pp. 1090–1091.

[24] (2019) SKLEARN: RFECV. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html

[25] (2019) SKLEARN: RFE. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

[26] (2019) Tensorflow. [Online]. Available: https://www.tensorflow.org/

[27] T. Colthurst, G. Hendry, Z. Nado, and S. D., "TensorForest: scalable random forests on tensorflow," in *Machine Learning Systems Workshop at NIPS*, 2016, pp. 1–9.

[28] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020.

[29] Y. Zhou and X. Jiang, "Android malware genome project," 2012. [Online]. Available: http://www.malgenomeproject.org/

[30] Contagio. (2019) Contagio mobile - mobile malware mini dump. [Online]. Available: http://contagiominidump.blogspot.com/

[31] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: detecting android malware by building markov chains of behavioral models (extended version)," *ACM Trans. Priv. Secur.*, vol. 22, no. 2, Apr. 2019.

[32] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium*, 2019, p. 729–746.

[33] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian, "EC2: ensemble clustering and classification for predicting android malware families," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 2, pp. 262–277, 2020.

[34] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "ANASTASIA: android malware detection using static analysis of applications," in *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016, pp. 1–5.

[35] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[36] X. Wang, J. Wang, and X. Zhu, "A static android malware detection based on actual used permissions combination and api calls," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 10, no. 9, pp. 1652–1659, 2016.

[37] F. Mercaldo, C. A. Visaggio, G. Canfora, and A. Cimitile, "Mobile malware detection in the real world," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, 2016, pp. 744–746.

[38] F. Martinelli, F. Mercaldo, and A. Saracino, "BRIDEMAID: an hybrid tool for accurate detection of android malware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, 2017, p. 899–901.

[39] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for ART," in *26th USENIX Security Symposium*, Aug. 2017, pp. 289–306.

[40] A. Fatima, S. Kumar, and M. K. Dutta, "Host-server-based malware detection system for android platforms using machine learning," in *Advances in Computational Intelligence and Communication Technology*. Springer Singapore, 2021, pp. 195–205.

[41] S. Kumar, D. Mishra, B. Panda, and S. K. Shukla, "Stdneut: Neutralizing sensor, telephony system and device state information on emulated android environments," in *Cryptology and Network Security*. Cham: Springer International Publishing, 2020, pp. 85–106.

[42] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, 2017.

[43] (2017) Shrinking APKs, growing installs. How your app's APK size impacts install. — by Sam Tolomei — Google Play Apps & Games — Medium. [Online]. Available: https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2

[44] A. Sinha, F. Di Troia, P. Heller, and M. Stamp, *Emulation Versus Instrumentation for Android Malware Detection*. Springer International Publishing, 2021, pp. 1–20.

[45] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dynalog: an automated dynamic analysis framework for characterizing android applications," in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, 2016, pp. 1–8.

[46] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2016.