Experimental comparison of features, analyses, and classifiers for Android malware detection

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Biniam Fisseha DEMISSIE

Mariano CECCATO

Naing Tun YAN
Singapore Management University, yannaingtun@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

See next page for additional authors

Noname manuscript No.
(will be inserted by the editor)

Experimental Comparison of Features, Analyses, and Classifiers for Android Malware Detection

Lwin Khin Shar · Biniam Fisseha Demissie · Mariano Ceccato · Yan Naing Tun · David Lo · Lingxiao Jiang · Christoph Bienert

Received: date / Accepted: date

Abstract Android malware detection has been an active area of research. In the past decade, several machine learning-based approaches based on different types of features that may characterize Android malware behaviors have been proposed. The usually-analyzed features include API usages and sequences at various abstraction levels (e.g., class and package), extracted using static or dynamic analysis. Additionally, features that characterize permission uses, native API calls and reflection have also been analyzed. Initial works used conventional classifiers such as Random Forest to learn on those features. In recent years, deep learning-based classifiers such as Recurrent Neural Network have been explored. Considering various types of features, analyses, and classifiers proposed in literature, there is a need

Lwin Khin Shar
Singapore Management University, Singapore
lkshar@smu.edu.sg

Biniam Fisseha Demissie
Technology Innovation Institute, 9639 Masdar City, Abu Dhabi, UAE
biniam.demissie@tii.ae

Mariano Ceccato
University of Verona, Verona, Italy
mariano.ceccato@univr.it

Yan Naing Tun
Singapore Management University, Singapore
yannaingtun@smu.edu.sg

David Lo
Singapore Management University, Singapore
davidlo@smu.edu.sg

Lingxiao Jiang
Singapore Management University, Singapore
lxjiang@smu.edu.sg

Christoph Bienert
Technical University of Munich, Germany
christoph.bienert@tum.de

of comprehensive evaluation on performances of current state-of-the-art Android malware classification based on a common benchmark.

In this study, we evaluate the performance of different types of features and the performance between a conventional classifier, Random Forest (RF) and a deep learning classifier, Recurrent Neural Network (RNN). To avoid temporal and spatial biases, we evaluate the performances in a time- and space-aware setting in which classifiers are trained with older apps and tested on newer apps, and the distribution of test samples is representative of in-the-wild malware-to-benign ratio. Features are extracted from a common benchmark of 7,860 benign samples and 5,912 malware, whose release years span from 2010 to 2020. Among other findings, our study shows that permission use features perform the best among the features we investigated; package-level features generally perform better than class-level features; static features generally perform better than dynamic features; and RNN classifier performs better than RF classifier when trained on sequence-type features.

Keywords Malware detection, machine learning, deep learning, Android

1 Introduction

Android platform has dominated the smart phone market for years now. With currently more than three billion devices running Android, it is the most popular end-user operating system in the world. Unsurprisingly, its enormous user base, coupled with the popularity of mobile apps led to the launch of several malicious applications by hackers. Symantec [60] reported that in 2018, it detected an average of 10,573 mobile malware per day; found that one in 36 mobile devices has high risk apps installed; and one in 14.5 apps accesses high risk user data.

To detect Android malware, several approaches have been proposed by the research community. These approaches have built detection models utilizing either sequence of API call features [62,33,47], use of API call features [54,16,71,9] or frequency of API call features [1,28]. API call features represent invocations of Android APIs. Some approaches [24,51,30,37,54,16,9,35] categorized Android APIs according to privilege levels (known as Android permissions). In Android, APIs are classified into four privilege levels — normal, signature, dangerous, and special. These approaches rely on the concept that malware typically require privileged operations (i.e., *dangerous* permissions) such as read/send SMS, read contact, read location, etc. Given that modern malware often use reflections and system (native API) calls, to hide their true behaviours and implement their malicious functionalities, some approaches such as [28,59,2] utilized features that represent native API calls and reflections, in an attempt to further distinguish malware from benign apps. In addition to permission uses, Kim et al. [34] also investigated the use of app components as features. Hence, a study of the significance of those features for Android malware detection on a common benchmark would be beneficial.

The API calls can be extracted at various abstraction levels such as method, class, package, and family. Since there are millions of unique methods in Android, some approaches [28,47,32] have proposed to abstract API calls at class and package levels. This reduced the number of features significantly and yet produced comparable or even better results [28,47,32] than using API calls at method level.

To extract these features, in general, two types of techniques are used — static analysis [9,16,69,28,47,32] and dynamic analysis [23,62,2]. Typically, static analysis-based features cover more information since static analysis can reason with the whole program code whereas dynamic analysis-based features are limited to the code that is executed. On the other hand, static analysis may have issues dealing with complex code such as code obfuscation, and modern malware is usually crafted with obfuscated code [28]. In general, static analysis and dynamic analysis complement each other. Hence, some approaches such as [35] perform both analyses and use both types of features.

Once these features have been extracted using program analyses, machine learning classifiers, such as Support Vector Machines (SVM), K-Nearest Neighbours, and Random Forest, are used to train on the features to build malware detectors. For instance, DadiDroid [32] and MamaDroid [47] used all the three classifiers mentioned above; RevealDroid [28] used SVM; Huang et al. [30] used AdaBoost, Naive Bayes, Decision Tree, and SVM. In parallel, other studies [62,41, 33,68] have focused on the use of deep learning classifiers, such as Convolutional Neural Network and Recurrent Neural Network, to build malware detectors. Deep learning classifiers use several neural network layers to study various levels of representations and extract higher-level features from the given lower-level ones. Hence, in general, they have a built-in feature selection process and are better at learning complex patterns. On the other hand, it generally comes with a much larger cost in terms of computational resources. Deep learning classifiers also typically have more parameters to tune and typically require intensive fine-tuning to match the characteristics of datasets.

In terms of evaluating the malware detection performance, cross validation or random split schemes are commonly used in literature [35,9,2,33]. But, as reported by Allix et al. [4] and Pendlebury et al. [49], these evaluation schemes are biased because data from the 'future' is used in training the classifier. Fu and Cai [27] showed that F-measure drops from 90% to 30% when training and test data are split based on one year gap. Additionally, Pendlebury et al. [49] reported an issue with spatial bias where the evaluation does not consider the realistic distribution between malware and benign samples.

In view of the proposals of different types of features, different types of underlying analyses used for feature extraction, and different types of classifiers, there is a need for a comprehensive evaluation on the performance of current state-of-the-art in Android malware classification on a common benchmark. There is also a need to evaluate the performances in a time- and space-aware setting. Hence, in this study, we evaluate the malware detection accuracy of features, analyses, and classifiers based on a common benchmark. Our evaluation includes the comparison between 14 types of features, the comparison between conventional machine learning classifier and deep learning classifier, the study of the impact of additional features such as native API calls and reflection, and combined static and dynamic features, and the robustness of features over Android evolution.

The experiments are conducted on a benchmark of 13,772 apps (7,860 benign apps and 5,912 malware) that are released from 2010 to 2020. Benign samples were collected from Androzoo repository [6] while malware samples were collected from both Androzoo and Drebin [9] repositories. We extract static features from call graph of Android package (apk) codes and dynamic features by executing the app in an Android emulator using our in-house intent-fuzzer combined with Android's Monkey testing framework [8].

Our preliminary study, documented in our conference paper [53], evaluated the performance between sequence of API calls features and use of API calls features and evaluated the performance between *un-optimized* classifiers. This paper extends the previous work and makes the following new contributions:

- We conduct a more systematic evaluation of the performances of features and classifiers. More specifically, we evaluate the performances in a time- and space-aware setting in which classifiers are trained with older apps and tested on newer apps and the distribution of benign and malware samples is representative of in-the-wild malware-to-benign ratio. These biases were not considered in our previous work.
- We significantly increase the size of our dataset. Our earlier work used the dataset of 6,971 apps. In this extension, we use the dataset of 13,772 apps collected over a period of *11 years*.
- We analyze sequence/use/frequency of API calls features at two different abstraction levels — class and package. We consider additional features that characterize reflection, native API calls, and permission uses and app component uses in our evaluation.
- We perform a series of optimizations on the deep learning classifier and the conventional machine learning classifier and compare their performance.

More specifically, the new research questions investigated in this study are:

- RQ1: Features. Which types of features perform the best? Are class-level features or package-level features better? Are static analysis-based features or dynamic analysis-based features better?
  Finding. Permission use features perform the best; Package-level features generally perform better than class-level features. Static features generally perform better than dynamic features.
- RQ2: Classifiers. When optimized, which type of classifiers — conventional machine learning (ML) classifier or deep learning (DL) classifier — performs better?
  Finding. In our previous work [53], the *un-optimized* DL classifiers did not perform as well as the best conventional ML classifier (Random Forest). In this evaluation, we observed that when optimized, the DL classifier (Recurrent Neural Network) performs better than the conventional ML classifier (Random Forest) on sequence-type features.
- RQ3: Additional features. Does the inclusion of features that characterize reflection, native API calls, and API calls that are classified as dangerous (dangerous permissions) improve the malware detection accuracy? Does combining static analysis-based and dynamic analysis-based features help?
  Finding. Overall, inclusion of reflection feature, native API calls features, dangerous permission features does not improve the performances significantly; combining static and dynamic-based features in a naive manner results in a worse performance.
- RQ4: Robustness. How robust are the malware detectors against evolution in Android framework and malware development?
  Finding. Generally, the performance of malware detectors is sensitive to changes in Android framework and malware development.

Data Availability. The scripts used in our experiments and sample datasets are available at our github page [1]. We provide more detailed results and the complete dataset upon request.

The rest of the paper is organized as follows. Section 2 discusses related work and motivates our work. Section 3 discuses the methodology — it explains the data collection and features extraction processes, and the machine learning and deep learning classifiers we optimized and used. Section 4 presents the empirical comparisons and discusses the results. Section 5 draws conclusions from this study and provides insights for Android malware researchers. Section 6 provides the concluding remarks and proposals for future studies.

2 Related Work on Android Malware Detection

Surveys. Naway and Li [46] reviewed the use of deep learning in combination with program analysis for Android malware detection. Recently, Liu et al. [38] also reviewed the use of deep learning for Android malware defenses. In contrast to Naway and Li [46], Liu et al. additionally reviewed critical aspects of using deep learning to prevent/defend against malicious behaviors (e.g., malware evolution, adversarial malware detection, deployment, malware families). However, the contributions of both studies is a literature survey, focusing on the use of deep learning for Android malware detection, rather than an empirical study like ours.

Empirical studies. There are a few empirical studies [5, 4, 40, 15] in literature, which contrast different types of features and classifiers to detect Android malware. Among them, Zhuo et al.'s study [40] is closely related to ours as it also investigates static sequence/use/frequency features extracted from control flow graph. The main differences between Zhuo et al.'s study and ours are a) we consider both static and dynamic analysis, b) we evaluate the use of native calls, reflection, permissions, and API calls at class level and package level, c) we evaluate a DL algorithm whereas we evaluate both conventional ML and DL algorithms, d) most importantly, Zhuo et al's study applied cross validation for performance evaluation, which introduces temporal and spatial biases whereas our evaluation takes measures to address these biases. In general, the other studies focus on a single dimension such as features, analyses, classifiers, or temporal and spatial aspects. By contrast, our study look at all those aspects and evaluate them on a common benchmark.

Allix et al. [4] conducts a large-scale empirical study on the dataset sizes used in Android malware detection approaches. Allix et al. [5] also investigates the relevance of timeline in the construction of training datasets. Both studies [5, 4] observed that performance of malware detector significantly dropped when they are tested against the malware in the wild, i.e., malware that were not seen in the training. Allix et al. [5] presents a critical literature review of Android malware classification based on supervised machine learning. They define a dataset to be *historically coherent* when the apps in the training set are all historically anterior to all the apps in the testing set. According to their experiment, when the dataset is not historically coherent, classification performances (e.g., F-measure) are artificially inflated. According to their literature review, a relevant portion

[1] https://github.com/Jesper20/msoftx

of the papers uses historically incoherent datasets, causing results to be biased. Another study [49] additionally discussed the importance of space-aware setting that consider the realistic distribution of malware and benign samples during both training and testing. We took measure to mitigate these two biases in our evaluations. The need of retraining an ML-based malware detector is defined by Cai [15] as the *sustainability* problem. Cai [15] compares five malware detectors, revealing limitations with respect to sustainability of the learned model. Our results confirm these findings. These existing studies were conducted on limited types of analyses (static analysis) and features (e.g., sequence of API calls), and limited span of app released years ($\leq$ 3 years). Our work addresses the gap by investigating the relevance of timeline in the construction of datasets representing different types of features extracted from apps released in a wide time span of 11 years. We provide complementary, additional findings to these existing studies.

Static analysis-based features. Several approaches rely on static analysis to extract features from the app such as permissions [24,65,51,30,37,54,16,9,59], the sequence of API calls [41,18,55,33,47,56,76], the use of API calls [54,73,16,71,9, 59,32,67,12,64], or the frequency of API calls [1,18,26,28]. A few approaches [28, 59] also relied on features that characterize native API calls and reflections. Since these approaches evaluate various types of features independently and majority of these approaches were not evaluated in a time- and/or space-aware manner, our work addresses this by evaluating all these types of features on a common benchmark in a time- and space-aware manner. In addition, our study evaluates features extracted not only with static analysis but also with dynamic analysis and with both static and dynamic analysis combined. And we evaluate these features on both ML and DL classifiers. Considering that analysis at method level leads to millions of features, resulting in long training time and memory consumption, some approaches [47,32,69] abstracted features at class, package, family, or entity levels, to save memory and time. Our study evaluates features at class level and package level.

Dynamic analysis-based features. Dynamic analysis-based approaches such as [23, 62,2,58] have mainly focused on features at native API calls (system calls). Narudin et al. [45] evaluate the performance of five ML classifiers on network features (API calls that involve network communication) extracted with dynamic analysis. Most dynamic analysis approaches have largely used Monkey (UI) test generator [46]. But Monkey test generator only focuses on exercising UI components and could miss out component interactions. In contrast to these approaches, our approach employs a combination of Monkey test generator and intent fuzzing.

Hybrid analysis-based features. As reported in Liu et al. [38], possibly due to high computational cost, very few approaches [72,36,7,58,14] combine static analysis and dynamic analysis. And, these approaches focus on extracting specific features that are generally considered to be dangerous, such as sending SMS and connecting to Internet. For example, Droid-sec [72] uses features that characterize permission requested and permission use, which are coarse-grained and prone to false positives [24]. DDefender [7] uses features that are based on permissions, network activities and native API calls. Monkey tool was also used in the dynamic analysis; thus it may not be able to generate all the events that a malware can make. Mobile-Sandbox [58] applies static analysis of manifest file and bytecode to guide the dynamic analysis process. It then analyzes native API calls during the application's execution. AASandbox [14] uses static analysis to extract suspicious code patterns, such as the use of *Runtime.exec*() and functions related to reflection. During the dynamic step, AASandbox runs the app in a controlled environment and monitors system calls. In contrast to the above-mentioned approaches, we evaluate more types of features, and evaluate both conventional machine learning and deep learning classifiers. We also employ a combination of Monkey test generator and intent fuzzing to cover both UI events and component interactions. Marvin [35] also uses both static analysis and dynamic analysis to extract features that are similar to the features extracted by our work. The features extracted include permissions, reflection, native calls, Java classes, etc. But its classifier is evaluated by randomly splitting training and test data, without considering the timeline in the construction of training data, which could produce biased results.

Robust classifiers. While Zhang et al. [74] proposes a way to mitigate the problem of model aging, Fu and Cai [27], MaMaDroid [47], Afonso [2], and RevealDroid [28] propose the use of features that could be robust against the evolution of apps (timeline). Our empirical study complements their work by evaluating which combination of features, program analyses, and classifiers produces robust malware detectors, on a common benchmark.

3 Methodology

This section explains the workflow of our empirical study. As illustrated in Figure 1, it consists of three phases. In the first phase, static analysis is used to extract manifest files and call graphs; dynamic analysis is used to generate execution traces, from benign and malware apps. In the second phase, various features — sequence/use/frequency of API calls features at class level and package level, permission uses, and app component uses — are extracted from call graphs and execution traces. Each type of features forms a distinct dataset. Each record in the dataset, representing an app, is tagged with its known label. In the last phase, classifiers — Random Forest (RF) and Recurrent Neural Network (RNN) — are trained and tested on the labeled datasets in a time- and space-aware setting and produce the evaluation results.

The following subsections discuss each phase in detail. As a running example, we will use a malicious app called *com.test.mygame* released in year 2017, which has been flagged as malware by 27 anti-viruses. It is a variant of the *SmsPay* malware where a legitimate app is repackaged with covert functions to send and receive SMS messages, potentially causing unexpectedly high phone charges.

3.1 Program Analysis

In this phase, static analysis and dynamic analysis are performed on the given Android Application Packages (APKs).

Static analysis. Given an APK, we use apktool[2] to extract Android manifest file and use FlowDroid [10] to extract call graph. Call graph contains paths from public entry points of the app to the program termination. Those paths contain

[2] https://ibotpeaches.github.io/Apktool/

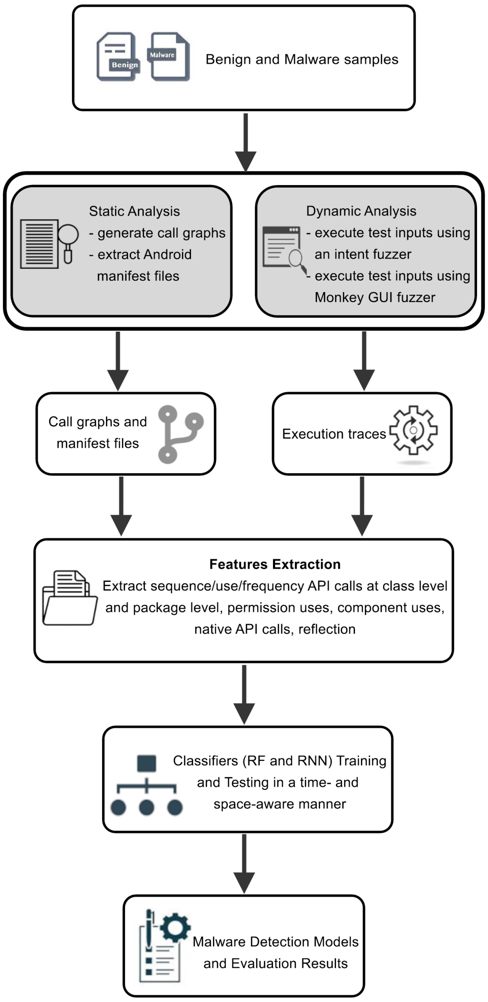


Fig. 1: The workflow of our experiments

sequences of API calls. FlowDroid is based on Soot [57]. Firstly, Soot converts a given APK (i.e., the DEX code) into an intermediate representation called Jimple and FlowDroid performs flow analysis on the Jimple code. The analysis is flow- and context-sensitive. FlowDroid also handles common native API calls. Using some heuristics, it tracks data flow across some commonly used native calls.

Dynamic analysis. Static call graphs characterize all possible program behaviors, in terms of API calls. But static analysis has inherent limitations, such as dealing with code obfuscation and reflection. FlowDroid can only resolve reflective API calls when the arguments used in the call are all string constants. Dynamic analysis can overcome this limitation. Hence, the goal of dynamic analysis here is to execute test inputs to observe concrete program behaviors. Since mobile apps are event driven in general, a good test generator needs to be able to generate various kinds of events. In Android, events are typically triggered by means of inter-component communication (intent messages sent by app components) or GUI inputs. Hence, we use two different test generators — an Intent fuzzer and a GUI fuzzer. Our Intent fuzzer was developed in our previous work [21]. Firstly, it analyzes call graph of the app to extract paths from public entry-points (i.e., inter/intra-component communication interfaces) to the leaf nodes. Similar to the static analysis phase, we generate the call graph of the app using Soot with FlowDroid plugin for Android. The call graph is then traversed forward in depth-first search manner starting from the root node until a leaf node is reached. The output of this step is paths from component entry points to the different leaf nodes (method calls without outgoing edges). Once the list of paths is available, the intent fuzzer generates inputs in an attempt to execute each path (target). The given app is installed and executed on a fresh Android emulator. The generated inputs are Intent messages that are sent to the app under test via Android Debug Bridge (ADB) commands. With ADB's privilege, we can also invoke private components as well as send events that can only be generated by the system (e.g., BOOT_COMPLETED). Execution traces are then collected using ADB logcat command. A genetic algorithm is used to guide the test generation, where fitness function is defined based on the coverage of nodes in the target path. To this end, we first instrument the app to collect execution traces and install the app on an Android emulator. We then run our intent fuzzer with statically collected values (such as static strings) from the app as seed (initial values). The generated inputs are Intent messages that are sent to the app under test via the Android Debug Bridge (ADB). Our goal is to maximize coverage and collect as many traces as possible. The traces are also used to guide the test generation.

While the Intent fuzzer exercises code parts that involve inter-/intra-component communications, it does not address user interactions through GUI. Therefore, to complement our intent fuzzer, we use Google's Android Monkey GUI fuzzer [8]. Monkey comes with the Android SDK and is used to randomly generate GUI input events such as tap, input text or toggle WiFi in an attempt to trigger abnormal app behaviors. We used Monkey because the random exploration of Monkey has been found to yield higher statement coverage than tools utilizing advanced exploration techniques [19]. And by complementing Monkey's approach with other strategies (in this case inter-/intra communication), we expect that the coverage could be further improved.

We measure the coverage achieved by this approach. Since code coverage is difficult to measure due to the usage of libraries, we measured component coverage, by measuring the ratio of the components that are executed when performing dynamic analysis and the components that are listed in the Android manifest file. Component coverage is shown in the histogram in Figure 2. While on average component coverage is approximately 43%, a remarkable number of apps reach 100% coverage. This degree of coverage is in line with literature results [19].

3.2 Features Extraction

From the call graphs and the execution traces generated in the previous phase, we extract *sequence* features, *use* features, and *frequency* features at class level and package level. Each type of features forms a distinct dataset. From the ex-

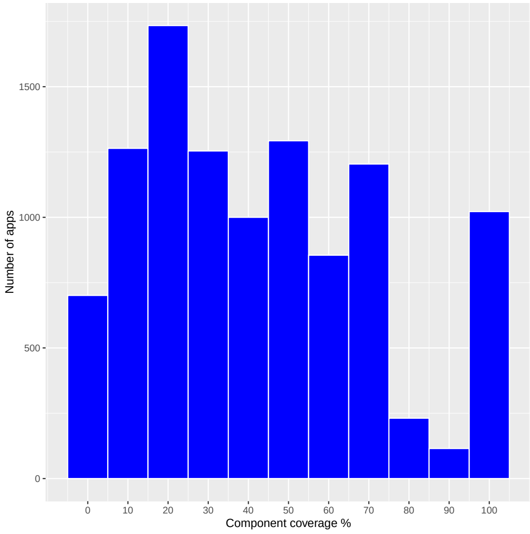


Fig. 2: Histogram of component coverage.

tracted API calls, we identify API calls that require dangerous permissions. We also identify native API calls (e.g., API calls that require system services and access hardware devices). Finally, we identify reflections (i.e., classes that start with $java.lang.reflect$) and mark them as *additional* features. From the Android manifest files, we extract features that represent permission uses (permission requests) and Android component uses as well, which are also considered as distinct datasets.

Note that the API calls that we extract here are abstracted at class level and package level. The rationale for choosing class and package level features instead of method level features is to reduce the amount of features, following the recent state-of-the-art approaches [28,69,47,32]. Method level features would result in millions of features that cost significantly long training time. Those recent approaches have reported that, despite the cost, the classifiers may not achieve a better accuracy since the feature vectors of the samples would be sparse and abstracted API calls features characterize Android malware even better. The abstraction also provides robustness against API changes in Android framework because methods are often subject to changes and deprecation. Figure 3 shows an example of an API at different levels.

Regarding the extraction of *dangerous* features, we implemented an in-house tool that crawls the Android permission documentation website[3] and maps API calls to dangerous permissions. This tool is similar to PScout [11] but PScout only supports up to Andriod 5.11. Our tool supports Android 11 (API 30)[4].

Sequence Features Extraction. We extract sequence of API calls from call graphs and execution traces. Given a call graph, we traverse the graph in a

[3] `https://developer.android.com/guide/topics/permissions/overview`

[4] our crawling tool is available in `https://github.com/Jesper20/msoftx`

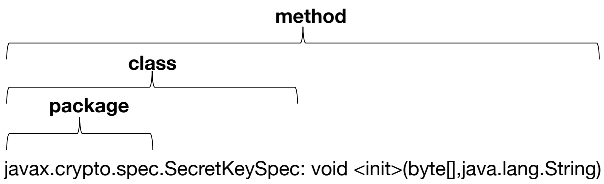


Fig. 3: An example of an API and its package, class, and method

depth first search manner and extract class/package signatures[5] as we traverse (hence, sequence). If there is a loop, the signature is traversed only once. Note that we only extract Android framework classes/packages, Java classes/packages, and standard `org` classes/packages (`org.apache`, `org.xml`, etc.). This is because it is common for malware to be obfuscated to circumvent malware detectors. The obfuscation often involves renaming of custom (user-defined) library and classes/-packages. Hence, a malware detector will not be robust against obfuscation if it is trained on custom library and classes/packages. A study [50] has shown that a simple renaming obfuscation can prevent popular anti-malware products from detecting the transformed malware samples. Hence, we filtered classes/packages that are not from the above-mentioned standard packages. Similarly, we extract classes/packages from the execution traces. However, since execution traces are already sequences, depth first search is not necessary. An excerpt of sequences of API calls extracted from a repackaged malware app *com.test.mygame* is shown in Figure 4.

Next, we discretized the sequence of API calls we extract above so that it can be processed by the classifiers. More precisely, we replace each unique class/package signature with an identifier, resulting in a sequence of numbers. We build a dictionary that maps each class to its identifier. During the testing or deployment phase, we may encounter unknown API calls. To address this, (1) we consider a large dictionary that covers over 160k class signatures and 4605 package signatures from standard libraries and (2) we replace all unknown signatures with a fixed identifier.

The length of the sequences varies from one app to another. The sequence length determines the number of features and to have a fixed number of features, it is necessary to unify the length of the sequences. Since we have two types of API calls sequences — from call graphs and from execution traces — we chose two different uniform sequence lengths. Initially, we extracted the whole sequences. We then took the median length of sequences from call graphs as the uniform sequence size, denoted as L_{cg}, for call graph-based sequence features and took the median length of sequences from execution traces as the uniform sequence, denoted as L_{tr}, for execution traces-based sequence features[6]. If the length of a given sequence is less than L, we pad the sequence with zeros; if the length is longer than L, we trim it to L, from the right. Hence, for each app, we end up with a sequence of numbers which is a feature vector. Each number in the sequence corresponds to the categorical value of a feature. The number of features is the uniform sequence

[5] note that package level features and class level features result in distinct datasets.

[6] L_{cg}=85000, L_{tr}=21000

length L. As a result, we obtain *static-sequence* features from call graphs at class level and package level, denoted as *ssfc* and *ssfp*, respectively. Likewise, we obtain *dynamic-sequence* features from execution traces at class level and package level, denoted as *dsfc* and *dsfp* respectively. As an example, Table 1 shows a sample dataset containing *sequence* features.

```
org.json.JSONObject a(android.telephony.TelephonyManager,
                      android.telephony.SubscriptionManager,int)
java.lang.reflect.AccessibleObject: void setAccessible(boolean)
android.app.Dialog: void dismiss()
android.content.ComponentName: java.lang.String toString()
java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)
android.telephony.SmsMessage$SubmitPdu: java.lang.String toString()
```

Fig. 4: An excerpt of sequence of API calls from a malware sample. It shows the sequence of API calls that require dangerous permissions (Telephony and Sms) and invoke a (potentially malicious) functionality via reflection.

Use Features Extraction. We extract use of API calls at class level and package level from call graphs and execution traces. The extraction process is the same for both call graphs and execution traces. We initially build a database that stores unique classes and packages. Again for obfuscation resiliency, we only consider the Android framework, Java, and standard `org` classes similar to extracting *sequence* features. Given call graphs or execution traces, we scan the files and extract the class signatures and the package signatures (sequence does not matter in this case). Each unique class or package in our database corresponds to a feature (Table 5). The value of a feature is 1 if the corresponding class/package is found in a given call graph or execution trace; otherwise, it is 0. As a result, we obtain *static-use* features from call graphs at class level and package level, denoted as *sufc* and *sufp*, respectively. Likewise, we obtain *dynamic-use* features from execution traces at class level and package level, denoted as *dufc* and *dufp* respectively. Table 2 shows a sample dataset containing *use* features at class level.

Frequency Features Extraction. We extract frequency of API calls from call graphs and execution traces in a similar way to use of API calls features. Except that, for each unique class/package signature, we record the number of its occurrences in the given call graph or execution trace, instead of recording the value 1 to denote the presence of a class/package signature. As a result, we obtain *static-frequency* features from call graphs at class level and package level, denoted as *sffc* and *sffp* respectively. Likewise, we obtain *dynamic-frequency* features from execution traces at class level and package level, denoted as *dffc* and *dffp* respectively. Table 3 shows a sample dataset containing *frequency* features.

Permission and App Component Features Extraction. Android manifest file specifies permissions requested and app components used by the app. Some approaches have used features that characterize permission uses [24,16,9,35] and app component uses [34] to detect Android malware. Therefore, it is important to analyze those features as well. We wrote a Python script to extract those features from Android manifest files. Figure 5 shows a snippet of AndroidManifest file. Line 1 shows the definition of the permission RECEIVE_ BOOT_COMPLETE the app

Table 1: An excerpt of *sequence* features extracted from static call graphs. Sequence length L is fixed at 21,000 for dynamic features and 85,000 for static features, which are the median lengths observed in our datasets.

	seq1	seq2	...	seq$L$	label
benign1	4921	6172	...	84111	0
benign2	29011	4490	...	3923	0
mal1	23712	8122	...	0	1
mal2	213	6311	...	0	1

Table 2: An excerpt of *use* features including *additional* (native calls and reflection) features

	telephony. SmsMessage	app. Dialog	reflect. AccessibleObject	hardware. Camera	label
benign1	1	1	0	0	0
benign2	0	1	1	1	0
mal1	1	1	1	0	1
mal2	0	0	0	1	1

Table 3: An excerpt of *frequency* features including *additional* (native calls and reflection) features

	telephony. SmsMessage	app. Dialog	reflect. AccessibleObject	hardware. Camera	label
benign1	3	9	0	0	0
benign2	0	10	2	3	0
mal1	4	1	2	0	1
mal2	0	0	0	2	1

wishes to be granted to receive system notification when the device completes booting. Line 3 shows the definition of a Broadcast Receiver app component *RestartServiceReceiver* that will handle the system notification for the boot-complete. Table 4 shows a sample dataset containing *permission-use* features.

```
1 <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
2
3 <receiver android:name="org.mysampleapp.RestartServiceReceiver">
4     <intent-filter>
5         <action android:name="android.intent.action.BOOT_COMPLETED"/>
6     </intent-filter>
7 </receiver>
```

Fig. 5: AndroidManifest snippet showing permission and component definition

Table 5 shows a summary of the features (datasets) extracted in this study. There are 14 types of features based on *Type* and *Level* of features and *Analysis* method used.

Table 4: An excerpt of *permission-use* features

	CAMERA	CALL_PHONE	READ_SMS	INTERNET	label
benign1	1	0	1	0	0
benign2	0	1	1	0	0
mal1	0	0	1	0	1
mal2	0	0	0	1	1

Table 5: Characteristics of the features (datasets) extracted

#	**Dataset**	**Type**	**Level**	**Analysis**	**#features**
1	dsfc	Sequence	Class	Dynamic	21,000
2	dsfp	Sequence	Package	Dynamic	21,000
3	ssfc	Sequence	Class	Static	85,000
4	ssfp	Sequence	Package	Static	85,000
5	dufc	Use	Class	Dynamic	28,816
6	dufp	Use	Package	Dynamic	1,255
7	sufc	Use	Class	Static	161,240
8	sufp	Use	Package	Static	4,605
9	dffc	Frequency	Class	Dynamic	28,816
10	dffp	Frequency	Package	Dynamic	1,255
11	sffc	Frequency	Class	Static	161,240
12	sffp	Frequency	Package	Static	4,605
13	pu	Use	Permission	Static	4,242
14	cu	Use	App Component	Static	116822

3.3 Classifiers

In the last phase, classifiers are trained and tested on the datasets. The following describes the classifiers used in our evaluations.

3.3.1 Deep Learning (DL) Classifier

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from raw input features. Deep learning classifiers typically comprise an input layer, one or more hidden layers, and an output layer. In our previous work [53], we studied three kinds of DL classifiers — standard deep neural network (DNN), convolutional neural network (CNN), and recurrent neural network (RNN). However, in this work, we decided to use only one DL classifier due to the huge amount of computation required for tuning and evaluating DL classifiers in general. We chose RNN and our rationale is as follows:

The main principles behind CNN are *sparse interaction*, *parameter sharing* and *equivariant representations* to implement filter operators (i.e., kernels), particularly fitting for the image recognition problem. But, in our context, API calls features hardly enjoy these properties. *Recurrent Neural Network (RNN)* is suitable for learning serial events such as language processing or speech recognition [22]. Unlike feed-forward neural networks like standard DNN and CNN, RNN can use their internal memory to process arbitrary sequences of inputs. More specifically, RNN has memory units, which retain the information of previous inputs or the state of hidden layers and its output depends on previous inputs, i.e., what API is used last will impact what API is used next. Hence, by design, RNN is suitable for *sequence*-type features. Furthermore, in our previous work [53], we observed that RNN performs well for *use* features. Therefore, we opted for RNN in our evaluation.

For *use* and *frequency* features, we use the RNN with one input layer, one LSTM layer, one hidden layer, and the output layer with Softmax function. The input layer accepts *use* or *frequency* features as vectors (Section 3.2). Each vector represents an app instance. These vectors are directly fed to the LSTM layer. The LSTM layer is used to avoid the error vanishing problem by fixing weight of hidden layers to avoid error decay and retaining not all information of input but only selected information which is required for future outputs.

Unlike *use* and *frequency* features, *sequence* features are not suitable for directly feeding to the LSTM layer because numerical values for the features will then be treated as *frequency* values by the classifier. As discussed in [33,41], it requires an additional vectorization technique that preserves the sequential patterns. Therefore, for *sequence* features, we add a vectorization step as follows: the RNN input layer accepts sequence features of each app instance (Section 3.2) as a vector. Each class/package identifier in the input vector is transformed into a vector using one-hot encoding [41,63]. The output from this input layer is then fed to the LSTM layer. Alternative to one-hot encoding, embedding techniques such as word2vec [42], apk2vec [44], node2vec [29] and graph2vec [43] can also be applied. However, we leave the problem of evaluating various embedding techniques in Android malware detection context as future work.

3.3.2 Conventional Machine Learning (ML) Classifier

Random Forest (RF) has been proven to be a highly accurate classifier for malware detection [25]. In our previous work [53], RF classifier was evaluated to be the best classifier among ML classifiers. Since we are not comparing the performance among ML classifiers in this extension work, we use only RF classifier as the flagship of ML classifiers[7]. RF is an ensemble of classifiers using many decision tree models [13]. A different subset of training data is selected with a replacement to train each tree. The remaining training data serves to estimate the error and variable importance. We used Scikit-learn [48] to run the RF classifier. Similar to RNN, we applied one-hot encoding for *sequence* features.

3.3.3 Optimizing the Classifiers

We tuned the hyper-parameters of both classifiers to achieve optimal performances as follows.

Tuning the hyper-parameters of RNN. For tuning the parameters, we sampled the data from year 2013 and year 2014 (see Table 8), which is never used as *test data* in our experiments. In total, the data contains about 1000 malware and 1000 benign samples. During the preliminary tuning, we observed that different datasets require different parameter configuration for improved results. In our preliminary phase, it took about 10 days to tune a relatively small dataset (*dufc*). It would take about 30 days each for the larger ones. Since it is intractable to do the tuning for each of the datasets. We decided to do tuning for only *dsfc*, *dufc* and *dffc*

[7] To cross validate the results, we also ran Logistic Regression and Support Vector Machines for one of the datasets. The results are briefly discussed in Section 4.3.

datasets. We then used the same optimal configuration of *dsfc* for other *sequence*-type datasets, i.e., *dsfp*, *ssfc*, and *ssfp*. The same is done for *use* and *frequency* datasets. We used Optuna, a hyper-parameter optimization framework [3], to tune the following hyper-parameters:

- Optimizer (ADAM, SGD, or RMSprop)
- learning rate (lr)
- number of neurons in hidden layer (hidden_sz)
- dropout ratio (p)
- Epoch
- decay weight

Table 6 shows the tuned hyper-parameter values and the F-measure results before and after hyper-parameter optimization.

Table 6: Results of RNN before tuning and after tuning, on the benchmark of apps from year 2013 and year 2014. **F1 (bf.)** represents the results before optimization; **F1 (aft.)** represents the results after optimization; **Optimizer** represents the optimizer used; **lr** represents the learning rate; **hidden_sz** represents the number of neurons used in hidden layer; **p** represents the drop out ratio; **Epoch** defines the number of times that the learning algorithm will work through the training dataset to update the parameters

Dataset	F1 (bf.)	F1 (aft.)	Optimizer	lr	hidden_sz	p	Epoch
dsfc (#1 in Table 5)	0.317	0.556	ADAM	0.0007	120	0.25	30
dufc (#5 in Table 5)	0.86	0.873	ADAM	0.001	30	0.25	30
dffc (#9 in Table 5)	0.748	0.872	ADAM	0.0007	70	0.25	30

Tuning the hyper-parameters of RF. Scikit-learn provides two widely-used tuning libraries — Exhaustive grid search and Randomized parameter optimization — for auto-tuning the hyper-parameters of a given classifier to a given dataset[8]. We combined both tuning methods as follows:

We first apply Randomized parameter optimization, which basically conducts a randomized search over parameters, where each setting is sampled from a distribution over possible parameter values. This gives us a good combination of hyper-parameter values efficiently. We then widen those hyper-parameter values to a reasonable range[9] and use exhaustive grid search to search for the best hypyer-parameter values among the given range. We followed the same process of tuning the RNN classifier. That is, we used the same apps from year 2013 and year 2014 as a basis to tune the RF classifier and we only tuned for *dsfc*, *dufc*, and *dffc* datasets. This results in the optimized hyper-parameters of random forest for Android malware classification as shown in Table 7.

3.4 Data Preprocessing

Imbalanced data causes the learning algorithm to bias towards the dominant classes, resulting in misclassification of minority classes. One effective way to im-

[8] `https://scikit-learn.org/stable/modules/grid_search.html`

[9] Reasonable range is determined according to the time budget of 5 hours.

Table 7: Results of RF before tuning and after tuning, on the benchmark of apps from year 2013 and year 2014. **F1 (bf.)** represents the results before optimization; **F1 (aft.)** represents the results after optimization; **n_estimators** represents the number of trees used; **min_samples_split** represents the minimum samples required for splitting a branch; **max_depth** represents the maximum depth of the tree.

Dataset	F1 (bf.)	F1 (aft.)	n_estimators	min_samples_split	max_depth
dsfc (#1 in Table 5)	0.605	0.657	200	5	90
dufc (#5 in Table 5)	0.817	0.823	94	2	60
dffc (#9 in Table 5)	0.827	0.835	10	5	100

prove the performance of classifiers is the synthetic generation of minority instances during the training phase. In our experiments, we use synthetic minority oversampling technique (smote) [17] to balance the training data.

4 Evaluation

This section presents the experimental comparison results of features, analyses, and classifiers for Android malware detection. Specifically, we investigate the following research questions:

- RQ1: Features. Which types of features perform better?
- RQ2: Classifiers. When optimized, which type of classifiers — conventional machine learning classifier or deep learning classifier — performs better?
- RQ3: Additional features. Does the inclusion of features that characterise reflection, native API calls, and API calls classified as dangerous (dangerous permissions) improve the malware detection accuracy? Does combining static analysis-based and dynamic analysis-based features help?
- RQ4: Robustness. How robust are the malware detectors against evolution in Android framework and malware development?

4.1 Experiment Design

Dataset. Our benchmark consists of 13,772 apps — 7,860 benign samples and 5,912 malware samples. The apps are released in a time-period between 2010 and 2020. Benign samples were collected from Androzoo repository [6]. Malware samples were collected from Androzoo repository [6] and Drebin repository [9]. The labeling of malware samples is confirmed by at least 10 antivirus software via VirusTotal[10]. Zhao et al. [75] highlighted the importance of considering sample duplication. That is, a dataset might contain the same or very similar apps with minor modification which might cause duplication bias. To avoid this bias, we randomized the download process. Initially, we downloaded over 50k samples from the repositories. However, as we evaluate the use of both static and dynamic analysis-based features, we had to filter those samples that can be analyzed by both static and dynamic analysis tools. When we use FlowDroid [10] tool to extract

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

call graphs, some of the apps caused exceptions. But the main bottleneck was dynamic analysis as our intent-fuzzing test generation tool encountered crashes or exceptions for several apps. Therefore, we were not able to extract features for those cases. Note that these are the limitations of the underlying program analysis tools and the objective of this experiment is to compare features and classifiers and not to assess the feature extraction components. We took the intersection of the apps that can be commonly analyzed by static and dynamic analysis tools and ended up with 13,772 apps. Several malware samples from our datasets are obfuscated. This is important to reflect the real world setting because malware authors heavily rely on obfuscation to hide the true behaviors. Table 8 shows the statistics of the datasets according to app release years.

Table 8: Dataset Statistics

Year	**Malware**	**Benign**	**Total**
2010	723	352	1075
2011	1407	683	2090
2012	450	470	920
2013	684	512	1196
2014	365	501	866
2015	639	347	986
2016	170	786	956
2017	866	401	1267
2018	467	3552	4019
2019	130	219	349
2020	11	37	48
Overall Total	5912	7860	13772

In comparison, Table 9 shows the sizes of dataset used by Android malware detection approaches in related work. But note that in comparison with these studies, we evaluate different types of features and both conventional machine learning and deep learning classifiers. Hence, it was intractable for us to use a larger dataset size. Yet, our dataset size is comparable to the sizes used in some recent studies such as [55, 69].

Table 9: Statistics of datasets of some popular malware detection approaches

Reference	#Benign	#Malware
Droid-sec [72]	250	250
DroidSift [73]	13500	2200
Drebin [9]	123453	5560
Narudin et al. [45]	20	1000
Maldozer [33]	37627	33066
RevealDroid [28]	24679	30203
Shen et al. [55]	3899	3899
EnMobile [69]	1717	4897
MaMadroid [47]	8447	35493
DaDiDroid [32]	43262	20431
Marvin [35]	84980	11733
Allix et al. [5]	200000	

Performance measure. We use F-measure (F1) to evaluate the performances, which is a standard measure typically used for evaluating malware detection accuracy [28,47]. F1 score reports an optimal blend (harmonic mean) of precision and recall, instead of a simple average because it punishes extreme values. A classifier with a precision of 1.0 and a recall of 0.0 has a simple average of 0.5 but an F1 score of 0. It can be computed as $F1 = 2*(precision*recall)/(precision+recall)$.

Evaluation Procedure. To avoid temporal bias problem as discussed in [4,49], we split the data based on their release years. We then train the classifier on the data released in a sequence of years and test it on the data released in the subsequent years. To avoid spatial bias problem as discussed in [49], we sample the malware instances from the test dataset so that malware-to-benign ratio is 18%[11]. We note from [49] that malware-to-benign ratio in the wild ranges from 6% to 18% and we did evaluate the features and classifiers with both ratios. But we will discuss the results based on the 18% ratio only.

Our general evaluation procedure to investigate our research questions is as follows: For a given feature (listed in Table 5), we run 21 training and test experiments with the given classifier (RF or RNN) as shown in Table 10.

Table 10: Time- and space-aware train and test procedure used in our experiments

No	**Train Years**	**Test Year**	**Malware-to-Benign Ratio (%) in test dataset**
1	2010-2014	2015	18
2	2010-2014	2016	18
3	2010-2014	2017	18
4	2010-2014	2018	18
5	2010-2014	2019	18
6	2010-2014	2020	18
7	2010-2015	2016	18
8	2010-2015	2017	18
9	2010-2015	2018	18
10	2010-2015	2019	18
11	2010-2015	2020	18
12	2010-2016	2017	18
13	2010-2016	2018	18
14	2010-2016	2019	18
15	2010-2016	2020	18
16	2010-2017	2018	18
17	2010-2017	2019	18
18	2010-2017	2020	18
19	2010-2018	2019	18
20	2010-2018	2020	18
21	2010-2019	2020	18

Hardware used. The experiments were performed on two Linux machines — 1) 40 cores Intel CPU E5-2640 2.40GHz 330GB RAM and 2) 12 cores Intel CPU E5-2603 1.70GHz 204GB RAM. It took about three months to extract call graphs and execution traces from all the 50k plus samples. It took about one month to extract the features from the final benchmark which contains 13,772 samples in total. It took about three months to conduct the machine learning experiments.

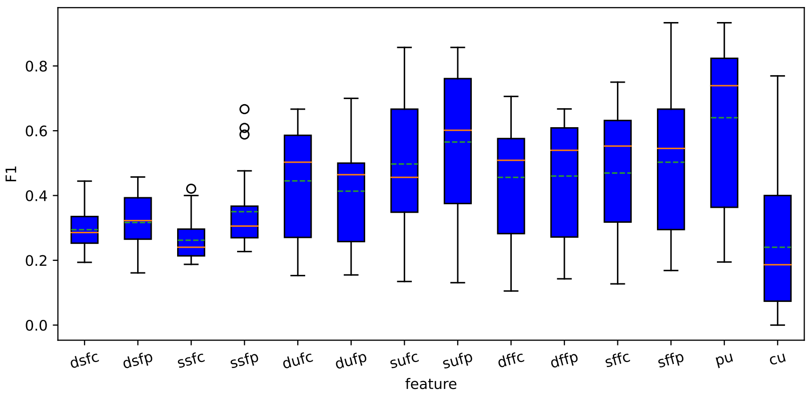
[11] if the size of malware samples does not amount to 18% (which is the case for year 2018 dataset), we use all available malware instances without sampling.

4.2 RQ1: Comparison among features

To investigate this research question, we compare the performance of 14 types of features listed in Table 5. Since we are not comparing the performance of the classifiers in this case, we shall use only Random Forest classifier to evaluate the features. For each feature listed in Table 5, we run 21 training and test experiments with the RF classifier as shown in Table 10.

Figure 6 shows the boxplot, mean, and standard deviation of F1 scores of Random Forest classifier with 14 different types of features based on the 21 train and test evaluations. We apply the Wilcoxon rank-sum test to perform pairwise comparison among features. For each feature, we perform the Wilcoxon rank-sum test against a different feature and test whether its F1 scores are statistically the same as the F1 scores of that feature (null hypothesis). The corresponding p-values are reported in Table 11.

We assume a standard significance level of 95% ($\alpha = 0.05$), i.e., we reject the null hypothesis if *p-value* < 0.05. Table 12 shows the comparison result of each feature against other features based on the p-values reported in Table 11. Each feature, say f, in a given row is compared against the features in the 'columns'. The label $<$ or $>$ in a given cell indicates whether the feature, f, is worse than or better than the feature listed in the corresponding column or not. The label ! denotes that there is no significant difference. For example, in the first row, the feature *dsfc* is compared against other features. It performs worse than *dufc*, *dufp*, *sufc*, *sufp*, *dffc*, *dffp*, *sffc*, *sffp*, and *pu* features; it has no statistical difference with other features.



	dsfc	dsfp	ssfc	ssfp	dufc	dufp	sufc	sufp	dffc	dffp	sffc	sffp	pu	cu
Mean	0.29	0.32	0.26	0.35	0.45	0.41	0.5	0.57	0.46	0.46	0.47	0.5	0.64	0.24
Stdev	0.07	0.08	0.06	0.13	0.18	0.15	0.22	0.23	0.18	0.19	0.21	0.22	0.26	0.23

Fig. 6: Comparison of features based on F1 scores. See Table 5 regarding the feature notations.

Overall, we can observe that *permission-use* feature (see row 'pu') significantly outperformed all other features, except class-level and package-level *static-use* features (*sufc* and *sufp*). It achieved the best F1 mean score at 0.64. The second best

Table 11: P-values of the Wilcoxon rank-sum test between each pair of features

	dsfp	ssfc	ssfp	dufc	dufp	sufc	sufp	dffc	dffp	sffc	sffp	pu	cu
dsfc	0.372	0.097	0.232	0.009	0.009	0.001	0.000	0.005	0.009	0.007	0.002	0.000	0.064
dsfp		0.023	0.831	0.015	0.021	0.007	0.001	0.009	0.014	0.025	0.006	0.000	0.040
ssfc			0.004	0.004	0.001	0.000	0.000	0.001	0.002	0.002	0.000	0.000	0.102
ssfp				0.107	0.155	0.015	0.002	0.076	0.087	0.063	0.031	0.001	0.013
dufc					0.392	0.571	0.054	0.831	0.597	0.580	0.308	0.003	0.002
dufp						0.308	0.021	0.159	0.174	0.314	0.174	0.003	0.003
sufc							0.399	0.642	0.678	0.697	1.000	0.058	0.001
sufp								0.051	0.080	0.170	0.302	0.222	0.000
dffc									0.725	0.529	0.443	0.004	0.002
dffp										0.753	0.505	0.004	0.002
sffc											0.763	0.014	0.002
sffp												0.040	0.000
pu													0.000

Table 12: Comparison of features. The label ! denotes no statistical difference; the labels $<$ and $>$ denote whether the F1 scores of a feature are statistically worse or better than the other feature, respectively.

	dsfc	dsfp	ssfc	ssfp	dufc	dufp	sufc	sufp	dffc	dffp	sffc	sffp	pu	cu
dsfc	NA	!	!	!	<	<	<	<	<	<	<	<	<	!
dsfp	!	NA	>	!	<	<	<	<	<	<	<	<	<	>
ssfc	!	<	NA	<	<	<	<	<	<	<	<	<	<	!
ssfp	!	!	>	NA	!	!	<	<	!	!	!	<	<	>
dufc	>	>	>	!	NA	!	!	!	!	!	!	!	<	>
dufp	>	>	>	!	!	NA	!	<	!	!	!	!	<	>
sufc	>	>	>	>	!	!	NA	!	!	!	!	!	!	>
sufp	>	>	>	>	!	>	!	NA	!	!	!	!	!	>
dffc	>	>	>	!	!	!	!	!	NA	!	!	!	<	>
dffp	>	>	>	!	!	!	!	!	!	NA	!	!	<	>
sffc	>	>	>	!	!	!	!	!	!	!	NA	!	<	>
sffp	>	>	>	>	!	!	!	!	!	!	!	NA	<	>
pu	>	>	>	>	>	>	!	!	>	>	>	>	NA	>
cu	!	<	!	<	<	<	<	<	<	<	<	<	<	NA

type of features is package-level *static-use* feature (*sufp*) with the F1 mean score of 0.57. *Component-use* feature performed the worst with the F1 mean score at 0.24. In general, we observe that package-level features achieve better or equal F1 scores against their class-level counterparts, e.g., *sufp=0.57* vs *sufc=0.5* and *sffp=0.5* vs *sffc=0.47*, except for the *dynamic-use* case. This result is consistent with the observation made in Onwuzurike et al. [47]. We also observe that static features achieve better F1 scores against their dynamic counterparts, e.g., *sufp=0.57* vs *dufp=0.41* and *sufc=0.5* vs *dufc=0.45*, except for the *dynamic-sequence* case. We also observe that *Sequence* features did not perform well in general as they all achieved less than 0.35 F1 mean score. All these results (of low F1 scores) show that Android malware detection is not actually a solved problem even though majority of the approaches in literature reported near perfect accuracy scores in their experiments. We believe that this is because those approaches did not take into account the biases that we considered in our experiments.

Summary-RQ1: *permission-use* achieved the best F1 mean score at 0.64, followed by another static analysis-based feature (*sufp*). In terms of the abstraction level, package-level features mostly perform better than class-level features. Given that the number of class-level features are much more than the number of package-level features (see Table 5), class-level features are also computationally costly. In terms of the analysis, static analysis-based features mostly perform better than dynamic analysis-based features. Hence, package-level and static features should be preferred.

4.3 RQ2: Optimized DL classifier vs Optimized Conventional ML classifier

In this section, we compare the performance of RF classifier and RNN classifier based on the following 7 types of features: *dsfp*, *ssfp*, *dufp*, *sufp*, *dffp*, *sffp*, and *pu*. Essentially we omitted class-level features and component use features because a) those datasets contain a large number of features and it would be computationally intractable to run all those datasets with deep learning classifier and b) in RQ1, it is already established that those omitted features do not perform as well as the others. To provide a baseline comparison, we also additionally compare our classifiers here against a state-of-the-art approach, MaMaDroid [47]. The train and test procedure is the same as the one applied in RQ1.

Figure 7 shows the boxplot of F1 scores of Random Forest classifier and RNN classifier based on the 7 types of features evaluated with the 21 train and test procedure (Table 10). Assuming a significance level of 95% ($\alpha = 0.05$), we apply Wilcoxon rank-sum test to perform the following pairwise comparisons:

1. RF-dsfp vs RNN-dsfp
2. RF-ssfp vs RNN-ssfp
3. RF-dufp vs RNN-dufp
4. RF-sufp vs RNN-sufp
5. RF-dffp vs RNN-dffp
6. RF-sffp vs RNN-sffp
7. RF-pu vs RNN-pu

Table 13 shows the comparison results between RF classifier and RNN classifier based on the Wilcoxon rank-sum tests. In previous work [53], we observed that un-optimized RNN classifier performs badly compared to ML classifiers. Here, we see that the optimization results in an improved performance for RNN classifier, especially for *sequence* features where RNN performed statistically better than RF in terms of F1 means. On the other hand, RF classifier performed better than RNN on four other features (but not statistically significant), especially for *frequency* and *permission* features. Overall, RNN achieved statistically better performance than RF on 2 out of 7 cases whereas RF performed better for 4 out of 7 cases though statistically not significant.

For the sake of completeness, we also evaluated RNN classifier using word embedding for sequence features (*dsfp* and *ssfp*). It achieved the F1 means of 0.325 and 0.354 for *dsfp* and *ssfp* datasets, respectively. This result is not better than that of RNN classifier with one-hot encoding but is still better than the RF classifier. These results align with the general agreement that RNN is suitable for learning serial events [22], especially since we used LSTM-based RNN that has the ability to effectively capture both long-term and short-term dependencies. On the other hand, we note that word embedding was much more efficient as it produces more compact vectors compared to one-hot encoding [42]. Time taken to train RNN with word embedding is in the order of hours whereas time taken to train RNN with one-hot encoding was in the order of days, for one round of training.

It may be surprising that the DL classifier, the more advanced classifier, does not perform significantly better than the ML classifier, except for sequence-type features. However, recent empirical studies [66,39] also found that DL classifiers are not always the overall winner. Even though those studies are conducted on different application domains (predicting relatedness in stack overflows [66] and generation of commit messages [39]), they also performed similar optimizations of the classifiers as us and used similar experiment designs. Typically, DL classifier needs thorough fine-tuning to the characteristics of the data. Although fine-tuning was done, it is only done on year 2013 and year 2014 data. App characteristics change with the evolution of Android, and this degrades the performance of both types of classifiers. But it seems to affect the DL classifier more. This is discussed in more detail in Section 4.5. Note that fine-tuning to fit all data is intractable, as it is computationally expensive. And it would also bias the results.

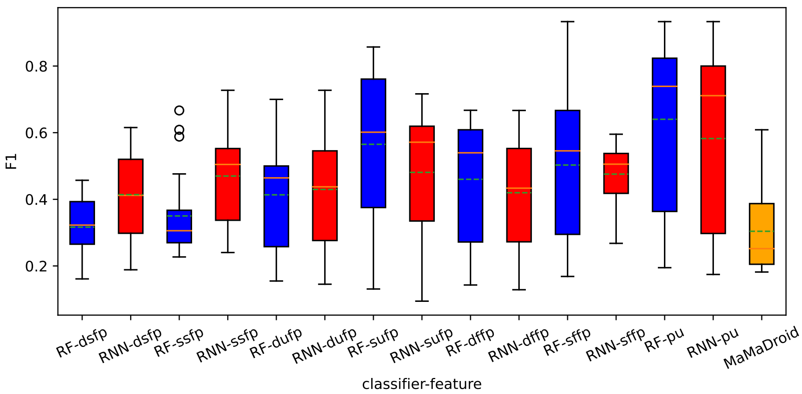


Fig. 7: Comparison between optimized ML classifier and optimized DL classifier based on F1 scores. *RF-dsfp* denotes Random Forest classifier tested with package-level dynamic sequence features; *RNN-dsfp* denotes Recurrent Neural Network classifier tested with package-level dynamic sequence features, and similarly for the rest. The last box plot shows the F1 scores of MaMaDroid [47] which is used as a baseline comparison.

Note that our previous work observed that Random Forest classifier achieved the best performance overall. Hence, we chose Random Forest as the Flagship of conventional ML algorithms for comparing against a DL algorithm. For a sanity check, we also evaluated Logistic Regression and Linear Support Vector Machines on package-level *static-frequency* features using the same training and test procedure. These classifiers achieved the F1 means of 0.48 and 0.41, respectively. In

Table 13: Wilcoxon test of F1 scores for RF and RNN classifiers. At significant level of 0.05, RNN performs statistically better than RF for dsfp and ssfp datasets.

Feature	**RF** F1 mean	**RNN** F1 mean	**p-value**
dsfp	0.317	0.393	**0.020**
ssfp	0.350	0.047	**0.011**
dufp	0.413	0.430	0.763
sufp	0.565	0.481	0.182
dffp	0.460	0.420	0.268
sffp	0.503	0.476	0.538
pu	0.640	0.582	0.466

comparison, RF classifier achieved 0.503. Hence, RF classifier achieved a better result.

To provide a baseline comparison, we also additionally compare our classifiers here against a state-of-the-art malware detector, MaMaDroid [47], which is based on *sequence*-type features. MaMaDroid builds a model from sequences obtained from the call graph of an app as Markov chains. Sequences are extracted at class level, package level, and family level. Four types of classifiers — Random Forest, 1-Nearest Neighbour, 3-Nearest Neighbor (3-NN), and Support Vector Machines are used to learn on the extracted sequence features. As a data preprocessing, Principal Component Analysis is applied. Random Forests achieved the best results in MaMaDroid's experiments. We used MaMaDroid tool[12](used as-is) to extract the sequence features from our benchmark apps. For the sake of consistency, we extracted package-level features[13]. We then used the same configuration of Random Forests classifier stated in MaMaDroid [47]. The last boxplot in Figure 7 shows the F1 scores of MaMaDroid classifier evaluated on our datasets with the same train and test procedure in Table 10. As we can observe in Figure 7, MaMaDroid achieved similar performance to our classifiers with sequence-type features but generally it does not perform as well as other classifier+feature configurations we used here.

Summary-RQ2: When optimized, the DL classifier (RNN) performed better than the ML classifier (RF) on sequence-type features. But DL classifiers do not necessarily always perform better than conventional ML classifiers. DL classifiers may be less useful, especially when the characteristics of test data often change.

4.4 RQ3: Additional Features

In this RQ, we perform two kinds of comparisons: (1) to determine whether additional features, which represent native calls, reflection, and API calls that require dangerous permissions, would improve the performance (2) to determine whether combining the static analysis-based features and the dynamic analysis-based features (hence "hybrid" features) would improve the performance. For both comparisons, we use Random Forest as a classifier.

[12] `https://bitbucket.org/gianluca_students/mamadroid_code/src/master/`

[13] In MaMaDroid's experiments [47], class-level and package-level features produced comparable performance

Regarding the first kind of comparison, we evaluate the RF classifiers trained with additional features based on the datasets: *dsfp*, *ssfp*, *dufp*, *sufp*, *dffp*, and *sffp*. ‘with additional features’ means that a given dataset is concatenated with its corresponding additional features. For example, *dsfp ‘with’* denotes *dynamic-sequence* features concatenated with sequence of native calls features, reflection, and API calls that require dangerous permissions. The train and test procedure is the same as the one applied in RQ1.

Figure 8 shows the box plots of the F1 scores for ‘without’ and ‘with’ additional features. Similar to RQ2, we apply Wilcoxon rank-sum test to perform pairwise comparisons and Table 16 reports the F1 means and the statistical test results. We observe that the performance significantly improved for the *dynamic-sequence* features when additional features are included. The F1 mean also increases for *static-sequence* and *dynamic-use* features but the improvements are not statistically significant. The F1 mean actually decreases for other types of features.

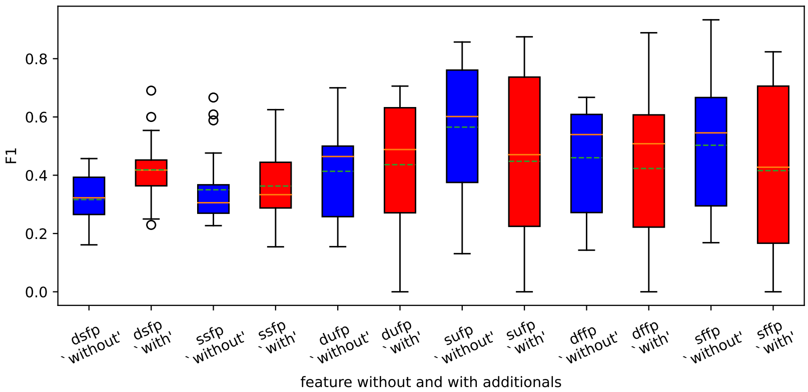


Fig. 8: Comparison of “without” and “with” additional features. *dsfp ‘with’* denotes that *dynamic-sequence* features are concatenated with sequence of native calls features, reflection, and API calls that require dangerous permissions; likewise for the others.

Table 14: Wilcoxon test of F1 scores for “without” and “with” additional features. “without” and “with” columns show the F1 means. Only *dynamic-sequence* feature shows statistical improvement when incorporated with additional features.

Feature	**without**	**with**	**p-value**
dsfp	0.317	0.419	**0.004**
ssfp	0.350	0.363	0.633
dufp	0.413	0.436	0.385
sufp	0.565	0.448	0.195
dffp	0.460	0.423	0.642
sffp	0.503	0.416	0.268

To explain this behavior, we performed principal component analysis of the *static-use* datasets containing only the additional features, i.e., *use* of native API calls, reflection, and dangerous permissions. Figure 11 shows the PCA plot of six most significant features from year 2015 to year 2020 datasets. As shown in the figure, the data points of malware samples largely overlaps with those of benign samples. Therefore, there is no difference between malware samples and benign samples in terms of the use of additional features.

This can be explained by the fact that it is legitimate for mobile apps to use those features to implement their services. That is, mobile apps do need to request dangerous permissions to access camera, microphone, heart rate (body sensor), etc. It is also common to use native calls to use system services like reading and writing to files, and use reflection to dynamically load new functionalities. For example, Figure 9 shows an excerpt of API calls extracted from a benign app $biart.com.flashlight$ that we sampled from our dataset. It contains the use of native API calls for accessing system services and dangerous permissions to use camera device.

We note that both benign and malware apps use API call features as well. And yet API call features can still discriminate malware. It is likely because each set of additional features look at a specific aspect of app behaviors, e.g., whether an app uses dangerous permission or not, whereas API call features cover the complete app behaviors based on call graphs or execution traces and thus, specific behaviors covered by additional features may have already been implicitly covered by API call features. Hence, we believe that API call features better profile the app behaviors and additional features do not further discriminate malware.

```
java.lang.System: long currentTimeMillis()
android.hardware.Camera: void startPreview()
java.lang.Thread: java.lang.Thread currentThread()
android.media.MediaPlayer: int getVideoHeight()
```

Fig. 9: An excerpt of API calls found in a benign app sample.

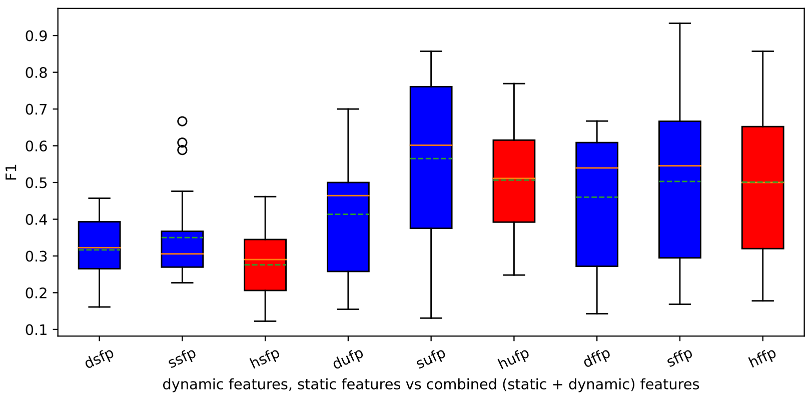
Regarding the second kind of comparison, we combine static analysis-based features and dynamic analysis-based features to determine whether the hybrid features would improve the performance. We concatenate *static-sequence* features and *dynamic-sequence* features, let us denote as $hsfp = ssfp \parallel dsfp$. Table 15 shows an example of *hsfp*. Likewise, we concatenate *static-use* features and *dynamic-use* features, and concatenate *static-frequency* features and *dynamic-frequency* features, denoted as *hufp* and *hffp*, respectively. We then perform the 21 training and test evaluations on those 3 new types of features using Random Forest as classifier. Note that we simply concatenate the two types of features without any data processing.

Figure 10 shows the F1 scores for "without" and "with" combining the static analysis-based features and the dynamic analysis-based features. Table 16 shows the Wilcoxon test results. As we can observe, the F1 mean actually decreases when the two types of features are combined, although there is no statistical difference according to Wilcoxon tests. This is likely due to overlapped features from the

Table 15: An excerpt of *hybrid-sequence* features.

	hsfp								
	ssfp				*dsfp*				
	s-seq1	s-seq2	...	s-seq$L$	d-seq1	d-seq2	...	d-seq$L$	label
benign1	4921	6172	...	84111	74921	567	...	84111	0
benign2	29011	4490	...	3923	12901	4490	...	3923	0
mal1	23712	8122	...	0	23712	6812	...	0	1
mal2	213	6311	...	0	23	63011	...	0	1

two analyses since both analyses extract features from the same app. For example, both analyses extract the package *android.net* as a feature. Assuming *use* features, static analysis will report the value 1 for this feature if it detects the presence of this package in the call graph. But dynamic analysis will report a value 0 for the same feature if it does not observe the execution of this package at runtime. On the other hand, static analysis will report the value 0 for *android.net* feature if it does not detect the presence of this package in the call graph; but dynamic analysis will report the value 1 for *android.net* if the app invokes this package using dynamic code loading, which is not presented in the static call graph. Hence, the conflicting values in the overlapped features may be confusing to the classifier, resulting in worse performance. Dealing with such overlapped features deserves a separate, thorough investigation as it requires to investigate how to leverage different types of information conveyed by static and dynamic analyses and extract the semantic meaning provided by these analyses together, rather than simply concatenating the two types of features.



	dsfp	ssfp	hsfp	dufp	sufp	hufp	dffp	sffp	hffp
Mean	0.317	0.35	0.276	0.413	0.565	0.507	0.46	0.503	0.501
Stdev	0.084	0.129	0.084	0.151	0.226	0.149	0.186	0.217	0.210

Fig. 10: F1 scores for "without" and "with" combining features

Table 16: Wilcoxon test of F1 scores for "without" and "with" combining features. No statistical difference was observed at a significance level of 0.05.

Comparison	**p-value**
hsfp vs dsfp	0.195
hsfp vs ssfp	0.074
hufp vs dufp	0.068
hufp vs sufp	0.308
hffp vs dffp	0.385
hffp vs sffp	0.860

Summary-RQ3: Including features that characterize reflection, native API calls, and dangerous permissions on top of API-call features does not further discriminate Android malware from benign apps because benign apps often use those features to implement their services. Combining the two types of analyses requires a means to deal with overlapped features because simply concatenating the two types of features results in worse performance compared to its static or dynamic counterparts.

4.5 RQ4: Robustness Against Android Evolution

In this research question, we investigate which combination of classifiers and features is most robust against Android evolution over time. Figure 12 shows the F1 score of different classifier-feature combinations against time. In Figure 12, we observe that most of the classifier-feature combinations show similar patterns in terms of F1 score over time, which means that those features are all sensitive to changes in Android permissions and API calls, and malware construction. For example, in late 2015, Google released Android 6 that introduced a redesigned app permission model. As in the previous version, apps are no longer automatically granted all the permissions they request at install-time. Users are required to grant or deny the specified permissions when an application needs to use it for the first time. The user can also revoke these permission at anytime. This caused a shift in the characteristics of benign apps in terms of permission and API usage. Furthermore, malware authors are also constantly advancing their malware so as to bypass the detection mechanisms, for example, by using obfuscation or applying adversarial learning [52]. Adversarial learning [31] is a technique that generates samples (e.g., malware variants) which are carefully crafted/perturbed to evade detection. Clearly, such changes in Android permissions and API calls, and malware construction affect malware detection performances.

Based on Figure 12, among the classifier+feature combinations, the RF classifier with *permission-use* (*RF-pu*), followed by the RNN classifier with *permission-use* (*RNN-pu*) could be considered most robust. When trained on year 2010-2014 dataset (Figure 12a), all other combinations did not achieve more than 0.65 F1 score on the datasets from subsequent test years whereas *RF-pu* and *RNN-pu* maintained above 0.65 F1 score, except for test year 2017 and 2018. We also observe that the RF classifier with *static-use* (*RF-sufp*) is an interesting combination. When trained on year 2010-2014 dataset, it did not perform well; but when trained with more data, i.e., year 2010-2015 dataset and subsequent ones, it produced a performance similar to *RF-pu* and *RNN-pu*. But its classifier counterpart

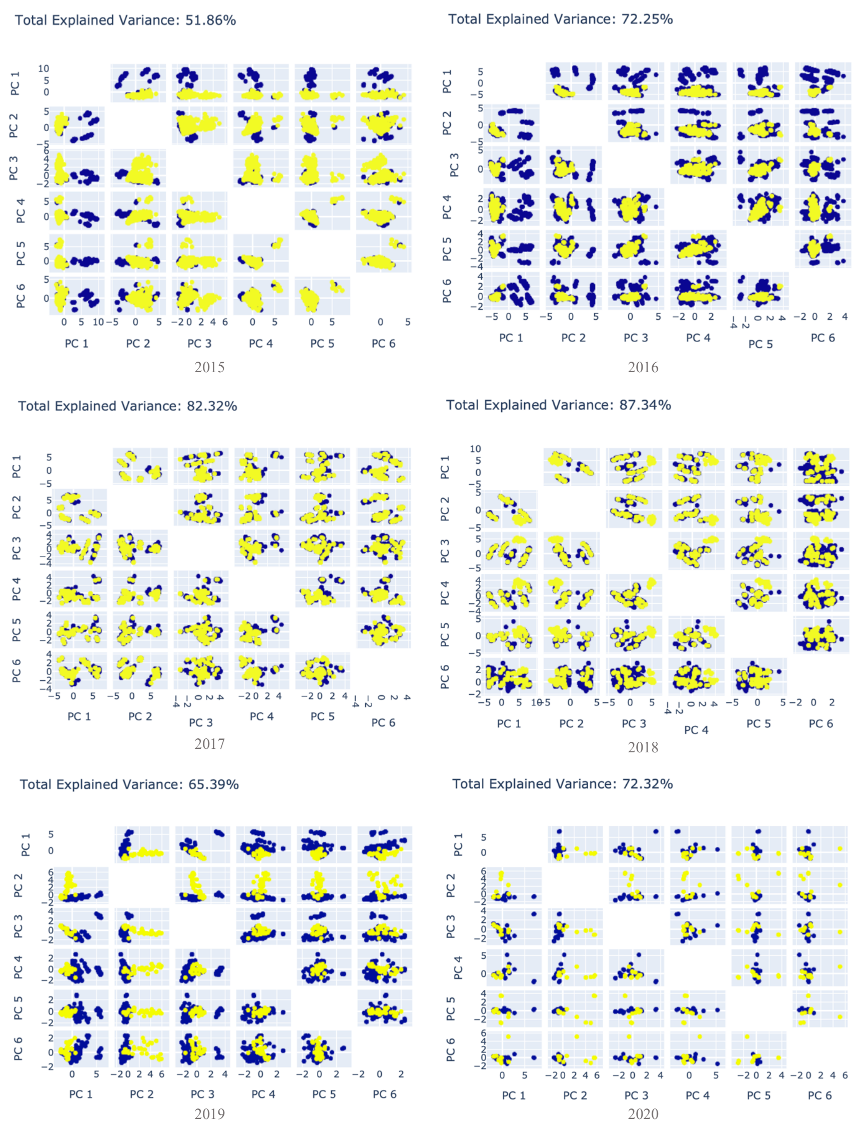


Fig. 11: Principal component analysis (6 components) of *additional* features used in malware and benign apps. Yellow color indicates malware and blue color indicates benign apps.

RNN-sufp did not perform quite as well and it is likely that RNN needs further fine tuning in this case. When there is sufficient training data, *RF-sufp* may be considered another robust classifier+feature combination.

We expected that the performances of classifiers+features will generally decrease over time. As observed in Figure 12a, this is the case from year 2015 to year 2018. But we observe that the performances actually improve in year 2019 and

2020, especially for *RF-pu* and *RNN-pu*. To understand this behavior, we did the PCA analysis of *permission-use* features in malware apps from years 2010-2014 versus malware apps from year 2019 and the PCA analysis of *permission-use* features in malware apps from years 2010-2014 versus malware apps from year 2020. The goal is to analyze the difference in characteristics between malware from those different released years. The result is shown in Figure 13. We observe that malware characteristics in terms of the use of permissions are similar. To further investigate the behavior shown in Figure 12(a), we extracted the most informative *permission-use* features for Random Forest for making classification decisions[14]. We found that most informative features from years 2010-2014 and from year 2019 and year 2020 commonly include READ_PHONE_STATE, SEND_SMS, READ_SMS, and GET_TASKS. Therefore, it is likely that those common features improved the detection performance for year 2019 dataset and year 2020 dataset.

Other commonly informative permission-use features across years (i.e., 2015, 2016, 2017, 2018) include ACCESS_WIFI_STATE, CHANGE_WIFI_STATE, INSTALL_SHORTCUT, INTERNET, and WRITE_EXTERNAL_STORAGE. Likewise, we analyzed the most informative *static-use* features across years; they include org.apache.http.conn, org.apache.http.client, java.security.cert, java.lang.annotation, android.net.wifi, android.transition, android.support.v4.accessibility service, android.media.session, javax.net, android.telephony, com.google.ads.mediation, and com.google.android.gms.ads. The functionality of these APIs range from network connection and telephony services to media and advertisement services. Hence, these APIs can be considered as good predictors of malware.

To evaluate whether time-aware and space-aware evaluation setting is important, we also ran 10-fold cross validation on RF classifier, with all the datasets combined (from year 2010 to year 2020). Table 17 compares the results. As shown in Table 17, the cross validated results are clearly better than the results of time-aware and space-aware evaluation setting (Table 10). That is, time and space biases unfairly report improved results. Allix et al. [5] reported that the F1 scores of Android malware classifier were lower than 0.7 in a time-aware scenario. Similarly, our best classifier achieved 0.64 F1 mean score. Fu and Cai [27] also reported that the F1 score dropped from about 90% to below 30% with a span of one year. Our results not only corroborate with the results of previous studies [5, 27] but also confirm that the biased improvement occurs regardless of features used. From this observation, we can conclude that timeline is an important aspect in malware detection. That is, malware detector should be re-trained whenever possible.

Table 17: Comparison of F1 mean scores between ten fold cross validation and time- and space-aware classification settings)

Feature	**10-fold CV**	**time- and space-aware settings**
dsfp	0.695	0.317
ssfp	0.670	0.35
dufp	0.797	0.413
sufp	0.824	0.565
dffp	0.795	0.46
sffp	0.796	0.503
pu	0.673	0.64

[14] using feature importance library in Scikit-learn

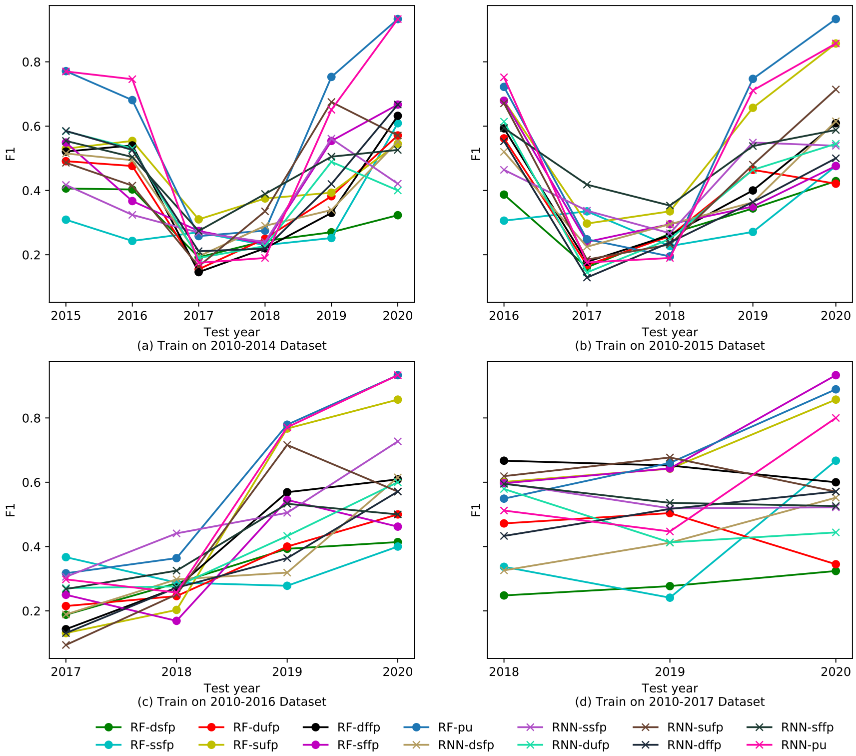


Fig. 12: Performance vs Time

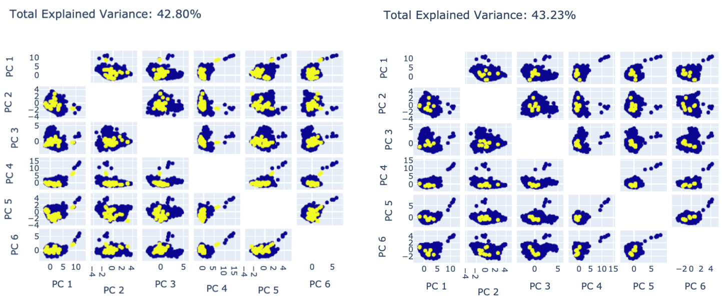
Summary-RQ4: Malware detectors are sensitive to Android evolution. That is, changes in app characteristics — benign or malware — result in fluctuation in the malware detector's performance regardless of the features and the classifiers used. Therefore, we recommend that malware detector should be re-trained with most relevant training samples whenever possible. Among the classifier-feature combinations that we investigated, the Random Forest with *permission-use* feature can be considered as the most robust.

4.6 Threats to Validity

Here we discuss the main threats to the validity of our findings.

Threats to the *conclusion validity* are concerned with issues that affect the ability to draw the correct conclusion. To limit this threat, we applied a statistical test (i.e., Wilcoxon rank-sum test) that is non-parametric, thus it does not assume experimental data to be normally distributed. Additionally, to increase heterogeneity of samples in the data set, we considered apps from multiple markets (Androzoo and Drebin) and released over multiple years (from 2010 to 2020).

Threats to *internal validity* concern the subjective factors that might have affected the results. To limit this threat, apps have been randomly selected and



(a) Characteristics of malware from year 2010-2014 (blue) vs malware from year 2019 (yellow)

(b) Characteristics of malware from year 2010-2014 (blue) vs malware from year 2020 (yellow)

Fig. 13: Principal Component Analysis of *permission use* features from malware apps. Yellow color indicates malware and blue color indicates benign apps.

downloaded from markets among those that satisfy our experimental settings (year 2010 to 2020) and experimental constraints (that they work with FlowDroid static analysis tool and with Monkey testing tool).

The threats to *construct validity* concern the data collection and analysis procedures. Labeling case studies as benign/malware is based on a standard approach, that is (i) relying on VirusTotal classification available as metadata information for apps from Androzoo; and (ii) manually recognized malicious behavior for apps from Drebin. Empirical results are based on F-measure, which is a standard performance measure. Moreover, to limit bias, we split the training data and the test data based on their release years and a realistic malware-to-benign distribution. A threat regarding the analysis procedure is the code coverage. As we explained in Section 3.1, we used a combination of GUI fuzzer and Intent fuzzer so as to cover both GUI events and inter-component communications which are typical and essential behaviors in mobile apps. However, like any other test generation-based approach, the code coverage of our test generator is also limited. Although we apply genetic algorithm, a state-of-the-art technique for Intent generation developed in our previous work [21], it was not able to generate test cases (Intents) for some of the paths in the call graphs. This could result in missing information in dynamic features and we acknowledge that this may explain the reason why static features perform better than dynamic features.

Regarding the analysis of sequence features, we trimmed the call sequences that are too long, taking into account the variances in sequence length among apps (see Section 3.2). One may argue that this may result in missing information in sequence features. However, our rationale is that using a longer sequence length result in many zero-features for most of the apps, resulting in several redundant features. We did some preliminary experiments using a longer sequence length and observed that the performance actually decreases. Another analysis-related threat is regarding the extraction of API-permission mapping (to extract *dangerous* permission features). We looked at the official Android documentation, which includes the mappings for public APIs only. The mappings for hidden and private APIs (which can be invoked through reflection) were not included. Thus, we acknowledge that such APIs, which may be in the dangerous permission category, would be missed by our approach. However, our argument here is that undocumented APIs change frequently and it is intractable for us to document them comprehensively, especially since we are dealing with versions across 11 years. Also from malware detection point of view, we believe that relying on a more consistent (official) list of APIs to build malware detector is more robust.

Threats to *external validity* concern the generalization of our findings due to the relatively smaller size of our dataset compared to the literature (Table 9). This is due to our consideration of several features and types of analyses (static and dynamic). By contrast, existing work that uses larger dataset size tends to focus on static analysis. However, as both static analysis- and dynamic analysis-based features are relevant and useful for malware detection, we decided to evaluate them in this work. Despite our best efforts, we were able to analyse only 13,772 apps due to the time taken and the computation complexity of our analyses. Especially our test generation tool took a long time to complete. It also encountered compatibility issues due to changes in different versions of the Android platform and we had to adapt our tool. On the other hand, to mitigate the issue, we considered apps from multiple app stores and released over 11 years.

5 Insights

For Antivirus vendors. In RQ1, we found that features at permission level or package level produce the best performances, while they are also computationally more efficient compared to more fine-grained features at class level. Deep learning algorithms have recently been used in the context of Android malware detection. They have the ability to learn hierarchical features and complex sequential features. But this usually comes at the cost of careful fine-tuning the hyper-parameters, which may take some time. On the other hand, conventional machine learning classifiers have been shown to be effective at Android malware detection. Especially, ensemble classifiers like Random Forest aggregates multiple classifiers to learn complex patterns. It achieves good classification results without much hyper-parameter tuning. In our experiments, we tuned both types of classifiers. But in RQ2, we observed that tuning Random Forest takes much less time and effort compared to RNN, the deep learning classifier. Yet the results are comparable, except for sequence features. Hence, our recommendation to antivirus vendors is that it is more cost-effective to use conventional machine learning classifiers for Android malware detection when using other types of features. In RQ4, we learnt that malware detectors' performance is sensitive to changes in Android framework and malware construction. Our recommendation to antivirus vendors is to take these findings into consideration when building and evaluating malware detectors and update them often.

For research community. In RQ1, we observed that dynamic features do not perform as well as static features in general. We discussed in Section 4.6, this could be due to code coverage issue by our test generator. Essentially, the test generator fails to generate test inputs when the target path requires satisfying certain conditions in the application logic or if the path involves user interaction (e.g., a click action). Researchers could improve on this aspect by combining dynamic test generation with static constraint solving techniques such as Thome et al. [61] for more effective test generation. In RQ3, we learnt that features that characterize reflection, native API calls, and dangerous permissions on top of API calls features do not further discriminate Android malware from benign apps. In Android, all the features, including native API calls, reflection and dangerous permissions are designed to be used, to serve their various functional purposes. However, malicious apps often abuse this to conduct malicious activities like accessing sensitive information. Hence, the empirical study conducted in this work is not complete. Distinct apps might have very different functionalities. What is considered legitimate of a particular set of apps (e.g., sharing contacts for a messaging app) can be considered a malicious behavior for other apps (e.g., a piece of malware that steals contacts, to be later used by spammers). A more accurate ML model should also take into consideration the main functionalities that are declared by an app, such as the ones proposed in [70, 20]. Hence, the future study should investigate the use of clustering to group apps with similar functionalities and evaluate based on clusters of those similar apps. In another note, we found that combining static-based features and dynamic-based features does not result in better performance. But in this case, we simply concatenated the two types of features without any data preprocessing to filter overlapped or redundant features. Future studies could consider applying an appropriate feature reduction technique, such as Principal Component Analysis, t-distributed Stochastic Neighbor Embedding, Multidimensional Scaling, Isometric mapping, etc., to deal with overlapped features.

In RQ4, we learnt that cross validation, which is typically used in Android malware detection approaches, allow malware "from the future" to be part of the training sets and thus, produce biased results. Allix et al. [5] observed that such a biased construction of training datasets has a positive impact on the performance of the classifiers and thus, the results are unreliable. In addition, Pendlebury et al. [49] also reported an issue with spatial bias where the evaluation does not consider the realistic distribution between malware and benign samples. Our studies also produced similar findings, despite different types of features we used. Therefore, researchers from Android malware detection community should validate their proposed state-of-the-art approaches again, taking into consideration the *temporal* and *spatial* biases.

6 Conclusion

In this work, we evaluated various techniques commonly used for building Android malware detectors. More specifically, we evaluated 14 types of features. We applied both static and dynamic analyses to extract those features. We evaluated two types of classifiers (conventional machine learning classifier and deep learning classifier). We also evaluated additional features (native API calls, reflection, and APIs that require dangerous permissions) and combined (static+dynamic) features. We investigated which types of features perform better; evaluated which types of classifiers perform better when optimized; evaluated whether additional features can improve the performance; and evaluated which combination of features and classifiers are more robust against the evolution of Android. We conducted the experiments in a time- and space-aware setting. We conducted all the experiments on a common benchmark containing 7,860 benign samples and 5,912 malware samples, collected over a period of 11 years (from year 2010 to 2020). We observed that *permission-use* features performed the best among features, followed by *static-use* package-level features; package-level features represent a good abstraction level as they perform well and are computationally efficient; static features perform better than dynamic features. We also observed that even when optimized, deep learning algorithm does not always perform better than conventional machine learning algorithm. Due to the tendency of benign apps to use reflection, native API calls, and APIs that require dangerous permissions, inclusion of those features does not further improve the accuracy of malware classification. Lastly, we found that malware classifier needs to be updated whenever applicable, regardless of features and classifiers used, as they are sensitive to changes in Android APIs and malware construction. In future work, we intend to further investigate other deep learning classifiers, given that we only evaluated one deep learning classifier in this work due to the time and resource required for optimization and evaluation. We also intend to investigate the effect of clustering the apps based on their functional similarities and performing the training and testing according to the clusters of apps.

Funding and/or Conflicts of interests/Competing interests

The work of Lwin Khin Shar, Yan Naing Tun, Lingxiao Jiang, and David Lo is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme, National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore. The work of Mariano Ceccato is partially supported by project MIUR 2018-2022 "Dipartimenti di Eccellenza". The authors declare that they have no conflict of interest.

References

1. Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: Proceedings of the International Conference on Security and Privacy in Communication Systems, pp. 86–103. Springer (2013)
2. Afonso, V.M., de Amorim, M.F., Grégio, A.R.A., Junquera, G.B., de Geus, P.L.: Identifying android malware using dynamically obtained features. Journal of Computer Virology and Hacking Techniques **11**(1), 9–17 (2015)
3. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2623–2631 (2019)
4. Allix, K., Bissyandé, T.F., Jérome, Q., Klein, J., Le Traon, Y., et al.: Empirical assessment of machine learning-based malware detectors for android. Empirical Software Engineering **21**(1), 183–211 (2016)
5. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Are your training datasets yet relevant? In: Proceedings of the International Symposium on Engineering Secure Software and Systems, pp. 51–67. Springer (2015)
6. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 468–471. ACM (2016)
7. Alshahrani, H., Mansourt, H., Thorn, S., Alshehri, A., Alzahrani, A., Fu, H.: Ddefender: Android application threat detection using static and dynamic analysis. In: Proceedings

of the IEEE International Conference on Consumer Electronics (ICCE), pp. 1–6. IEEE (2018)

8. Android: UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey` (2019)
9. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Proceedings of the Network and Distributed System Security Symposium, vol. 14, pp. 23–26 (2014)
10. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 259–269. ACM (2014)
11. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the Android permission specification. In: Proceedings of the 2012 ACM conference on Computer and Communications Security, pp. 217–228. ACM (2012)
12. Bai, Y., Xing, Z., Li, X., Feng, Z., Ma, D.: Unsuccessful story about few shot malware family classification and siamese network to the rescue. In: Proceedings of the 42nd International Conference on Software Engineering, pp. 1560–1571. IEEE/ACM (2020)
13. Barandiaran, I.: The random subspace method for constructing decision forests. IEEE Trans. Pattern Anal. Mach. Intell **20**(8), 1–22 (1998)
14. Bläsing, T., Batyuk, L., Schmidt, A.D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In: Proceedings of the 5th International Conference on Malicious and Unwanted Software, pp. 55–62. IEEE (2010)
15. Cai, H.: Assessing and improving malware detection sustainability through app evolution studies. ACM Transactions on Software Engineering and Methodology (TOSEM) **29**(2), 1–28 (2020)
16. Chan, P.P., Song, W.K.: Static detection of android malware by using permissions and api calls. In: Proceedings of the International Conference on Machine Learning and Cybernetics, vol. 1, pp. 82–87. IEEE (2014)
17. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)
18. Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: 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 (2016)
19. Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for android: Are we there yet?(e). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 429–440. IEEE (2015)
20. Demissie, B.F., Ceccato, M., Shar, L.K.: Anflo: Detecting anomalous sensitive information flows in android apps. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 24–34. IEEE/ACM (2018)
21. Demissie, B.F., Ceccato, M., Shar, L.K.: Security analysis of permission re-delegation vulnerabilities in android apps. Empirical Software Engineering **25**(6), 5084–5136 (2020)
22. Deng, L., Yu, D., et al.: Deep learning: methods and applications. Foundations and Trends® in Signal Processing **7**(3–4), 197–387 (2014)
23. Dini, G., Martinelli, F., Saracino, A., Sgandurra, D.: Madam: a multi-level anomaly detector for android malware. In: Proceedings of the International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, pp. 240–253. Springer (2012)
24. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on Computer and Communications Security, pp. 235–245. ACM (2009)
25. Eskandari, M., Hashemi, S.: A graph mining approach for detecting unknown malwares. Journal of Visual Languages & Computing **23**(3), 154–162 (2012)
26. Fan, M., Liu, J., Luo, X., Chen, K., Chen, T., Tian, Z., Zhang, X., Zheng, Q., Liu, T.: Frequent subgraph based familial classification of android malware. In: Proceedings of the 27th International Symposium on Software Reliability Engineering, pp. 24–35. IEEE (2016)
27. Fu, X., Cai, H.: On the deterioration of learning-based malware detectors for android. In: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 272–273. IEEE (2019)
28. Garcia, J., Hammad, M., Malek, S.: Lightweight, obfuscation-resilient detection and family identification of android malware. ACM Transactions on Software Engineering and Methodology (TOSEM) **26**(3), 11 (2018)

29. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864 (2016)
30. Huang, C.Y., Tsai, Y.T., Hsu, C.H.: Performance evaluation on permission-based detection for android malware. In: Advances in Intelligent Systems and Applications-Volume 2, pp. 111–120. Springer (2013)
31. Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D.: Adversarial machine learning. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, pp. 43–58 (2011)
32. Ikram, M., Beaume, P., Kaafar, M.A.: Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling. arXiv preprint arXiv:1905.09136 (2019)
33. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: Maldozer: Automatic framework for android malware detection using deep learning. Digital Investigation **24**, S48–S59 (2018)
34. Kim, T., Kang, B., Rho, M., Sezer, S., Im, E.G.: A multimodal deep learning method for android malware detection using various features. IEEE Transactions on Information Forensics and Security **14**(3), 773–788 (2018)
35. Lindorfer, M., Neugschwandtner, M., Platzer, C.: Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In: Proceedings of the 39th Annual Computer Software and Applications Conference, vol. 2, pp. 422–433. IEEE (2015)
36. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C.: Andrubis–1,000,000 apps later: A view on current android malware behaviors. In: Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), pp. 3–17. IEEE (2014)
37. Liu, X., Liu, J.: A two-layered permission-based android malware detection scheme. In: Proceedings of the 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, pp. 142–148. IEEE (2014)
38. Liu, Y., Tantithamthavorn, C., Li, L., Liu, Y.: Deep learning for android malware defenses: a systematic literature review. ACM Journal of the ACM (JACM) (2022)
39. Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z., Wang, X.: Neural-machine-translation-based commit message generation: how far are we? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 373–384 (2018)
40. Ma, Z., Ge, H., Liu, Y., Zhao, M., Ma, J.: A combination method for android malware detection based on control flow graphs and machine learning algorithms. IEEE access **7**, 21235–21245 (2019)
41. McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., et al.: Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 301–308. ACM (2017)
42. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems **26** (2013)
43. Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S.: graph2vec: Learning distributed representations of graphs (2017)
44. Narayanan, A., Soh, C., Chen, L., Liu, Y., Wang, L.: apk2vec: Semi-supervised multi-view representation learning for profiling android applications. In: Proceedings of the IEEE International Conference on Data Mining (ICDM), pp. 357–366. IEEE (2018)
45. Narudin, F.A., Feizollah, A., Anuar, N.B., Gani, A.: Evaluation of machine learning classifiers for mobile malware detection. Soft Computing **20**(1), 343–357 (2016)
46. Naway, A., LI, Y.: A review on the use of deep learning in android malware detection (2018)
47. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). ACM Transactions on Privacy and Security (TOPS) **22**(2), 14 (2019)
48. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
49. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., et al.: Tesseract: Eliminating experimental bias in malware classification across space and time. In: Proceedings of the 28th USENIX Security Symposium, pp. 729–746. USENIX Association (2019)

50. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 329–334 (2013)
51. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: Puma: Permission usage to detect malware in android. In: Proceedings of the International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions, pp. 289–298. Springer (2013)
52. Shahpasand, M., Hamey, L., Vatsalan, D., Xue, M.: Adversarial attacks on mobile malware detection. In: Proceedings of the IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile), pp. 17–20. IEEE (2019)
53. Shar, L.K., Demissie, B.F., Ceccato, M., Minn, W.: Experimental comparison of features and classifiers for android malware detection. In: Proceedings of the 7th International Conference on Mobile Software Engineering and Systems, pp. 50–60. IEEE/ACM (2020)
54. Sharma, A., Dash, S.K.: Mining api calls and permissions for android malware detection. In: Proceedings of the International Conference on Cryptology and Network Security, pp. 191–205. Springer (2014)
55. Shen, F., Del Vecchio, J., Mohaisen, A., Ko, S.Y., Ziarek, L.: Android malware detection using complex-flows. IEEE Transactions on Mobile Computing **18**(6), 1231–1245 (2018)
56. Shi, L., Ming, J., Fu, J., Peng, G., Xu, D., Gao, K., Pan, X.: Vahunt: Warding off new repackaged android malware in app-virtualization's clothing. In: Proceedings of the Conference on Computer and Communications Security, pp. 535–549. ACM (2020)
57. Soot: Soot - a java optimization framework, https://github.com/sable/soot (2018). URL `https://sable.github.io/soot/`
58. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1808–1815 (2013)
59. Suarez-Tangil, G., Dash, S.K., Ahmadi, M., Kinder, J., Giacinto, G., Cavallaro, L.: Droidsieve: Fast and accurate classification of obfuscated android malware. In: Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy, pp. 309–320 (2017)
60. Symantec: Internet Security Threat Report. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf` (2019)
61. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.: Search-driven string constraint solving for vulnerability detection. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering, pp. 198–208. IEEE (2017)
62. Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., Yagi, T.: Malware detection with deep neural network using process behavior. In: Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 577–582. IEEE (2016)
63. Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., Yagi, T.: Malware detection with deep neural network using process behavior. In: Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 577–582. IEEE (2016)
64. Wu, B., Chen, S., Gao, C., Fan, L., Liu, Y., Wen, W., Lyu, M.R.: Why an android app is classified as malware: Toward malware classification interpretation. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(2), 1–29 (2021)
65. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: Droidmat: Android malware detection through manifest and api calls tracing. In: Proceedings of the 7th Asia Joint Conference on Information Security, pp. 62–69. IEEE (2012)
66. Xu, B., Shirani, A., Lo, D., Alipour, M.A.: Prediction of relatedness in stack overflow: deep learning vs. svm: a reproducibility study. In: Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement, pp. 1–10. ACM/IEEE (2018)
67. Xu, K., Li, Y., Deng, R., Chen, K., Xu, J.: Droidevolver: Self-evolving android malware detection system. In: Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P), pp. 47–62 (2019)
68. Xu, K., Li, Y., Deng, R.H., Chen, K.: Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 473–487. IEEE (2018)
69. Yang, W., Prasad, M., Xie, T.: Enmobile: Entity-based characterization and analysis of mobile malware. In: Proceedings of the IEEE/ACM 40th International Conference on Software Engineering, pp. 384–394. IEEE (2018)
70. Yang, X., Lo, D., Li, L., Xia, X., Bissyandé, T.F., Klein, J.: Characterizing malicious android apps by mining topic-specific data flow signatures. Information and Software Technology **90**, 27–39 (2017)

71. Yerima, S.Y., Sezer, S., Muttik, I.: High accuracy android malware detection using ensemble learning. IET Information Security **9**(6), 313–320 (2015)
72. Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-sec: deep learning in android malware detection. In: ACM SIGCOMM Computer Communication Review, vol. 44, pp. 371–372. ACM (2014)
73. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 1105–1116 (2014)
74. Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M.: Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In: Proceedings of the Conference on Computer and Communications Security, pp. 757–770. ACM (2020)
75. Zhao, Y., Li, L., Wang, H., Cai, H., Bissyandé, T.F., Klein, J., Grundy, J.: On the impact of sample duplication in machine-learning-based android malware detection. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(3), 1–38 (2021)
76. Zou, D., Wu, Y., Yang, S., Chauhan, A., Yang, W., Zhong, J., Dou, S., Jin, H.: Intdroid: Android malware detection based on api intimacy analysis. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(3), 1–32 (2021)