DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling

Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao

***Abstract*— Most existing Android malware detection and categorization techniques are static approaches, which suffer from evasion attacks, such as obfuscation. By analyzing program behaviors, dynamic approaches are potentially more resilient against these attacks. Yet existing dynamic approaches mostly rely on characterizing system calls which are subject to system-call obfuscation. This paper presents DroidCat, a novel dynamic app classification technique, to complement existing approaches. By using a *diverse* set of dynamic features based on method calls and inter-component communication (ICC) Intents without involving permission, app resources, or system calls while fully handling reflection, DroidCat achieves superior robustness than static approaches as well as dynamic approaches relying on system calls. The features were distilled from a behavioral characterization study of benign versus malicious apps. Through three complementary evaluation studies with 34 343 apps from various sources and spanning the past nine years, we demonstrated the stability of *DroidCat* in achieving high classification performance and superior accuracy compared with the two state-of-the-art peer techniques that represent both static and dynamic approaches. Overall, *DroidCat* achieved 97% F1-measure accuracy consistently for classifying apps evolving over the nine years, detecting or categorizing malware, 16%–27% higher than any of the two baselines compared. Furthermore, our experiments with obfuscated benchmarks confirmed higher robustness of *DroidCat* over these baseline techniques. We also investigated the effects of various design decisions on DroidCat's effectiveness and the most important features for our dynamic classification. We found that features capturing app execution structure such as the distribution of method calls over user code and libraries are much more important than typical security features such as sensitive flows.**

***Index Terms*— Android, security, malware, dynamic analysis, profiling, detection, categorization, stability, robustness, obfuscation.**

I. Introduction

ANDROID has been the target platform of 97% malicious mobile apps [1], most of which steal personal information, abuse privileged resources, and/or install additional malicious software [2]. With the Android market growing rapidly, it is critically important to differentiate malware from benign apps (i.e., malware detection). Further, for threat assessment and defense planning, it is also crucial to differentiate malware of different families (i.e., malware categorization by family).

Two main classes of approaches to Android malware detection/categorization have been studied: static and dynamic. Static approaches leverage static code analysis to check whether an app contains abnormal information flows or calling structures [3]–[8], matches malicious code patterns [9], [10], requests for excessive permissions [11]–[14], and/or invokes APIs that are frequently used by malware [15]–[18].

Static approaches may have the advantage of being sound and scalable to screening large numbers of apps, yet they cannot always precisely detect malware for three reasons. First, due to the event-driven features of Android, such as lifecycle callbacks and GUI handling, run-time control/data flows are not always statically estimatable; they depend on the run-time environment. This approximation makes static analysis unable to reveal many malware activities. Second, the mere existence of some permissions and/or APIs in code does not always mean that they are actually executed or invoked frequently at runtime to cause an attack. Purely checking for existence of permissions and/or APIs can cause static analysis to wrongly report malware. In particular since API Level 23, Android has added dynamic permission support such that apps can request, acquire, and revoke permissions at runtime [19]. This new run-time permission mechanism implies that static approaches will not be able to discover when an abnormal permission is requested and granted at runtime, and would suffer from more false alarms if users revoke dangerous permissions after app installation. Third, static approaches have limited capabilities in detecting malicious behaviors that are exercised through dynamic code constructs (e.g., calling sensitive APIs via reflection). These and other limits [20] make static analysis vulnerable to widely adopted detection-evading schemes (e.g., code obfuscation [21] and metamorphism [22]). Recently, resource-centric features are also used in static code analysis to overcome the above-mentioned limitations [23]. However, such approaches can still be evaded by malware adopting resource obfuscation [24], [25].

In comparison, dynamic approaches provide a complementary way to detect/categorize malware [26]–[29]. In particular, behavior-based techniques [30] model program *behavioral profiles* [31], [32] with system/API call traces (e.g., [33]–[35]) and/or resource usage [28], [33]. Machine learning has been increasingly incorporated in these

Manuscript received April 12, 2018; revised September 5, 2018; accepted October 26, 2018. Date of publication November 1, 2018; date of current version February 13, 2019. This work was supported in part by an award from Washington State University (NFSG-131074-002). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Lorenzo Cavallaro. *(Corresponding author: Haipeng Cai.)*

H. Cai is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163 USA (e-mail: haipeng.cai@wsu.edu).

N. Meng, B. Ryder, and D. Yao are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24060 USA.

Digital Object Identifier 10.1109/TIFS.2018.2879302

techniques, training classification models from those profiles to distinguish malware from benign apps. However, system-call based malware detectors can still be evaded when an app obfuscates system calls [30], [36]–[38]. Sensitive API usage does not necessarily indicate malicious intentions. Abnormal resource usage does not always correspond to abnormal behaviors, either. Generally, behavior-based approaches relying on system-call sequences and/or dependencies may be easily thwarted by system-call obfuscation techniques (e.g., mimicry attack [37] and illusion [38]). A more comprehensive dynamic app classifier is needed to capture varied behavioral profiles and thus be *robust* to attacks against specific profiles.

Recent dynamic Android malware categorization approaches utilize the histogram [39] or chains (or dependencies) [34] of system calls. A recent dynamic Android malware detection technique [26] differentiates API call counts and strictly matches API signatures to distinguish malware from benign apps. Due to the underlying app features used, both kinds of techniques are subject to replacement attacks [30] (replacing system-call dependencies with semantically equivalent variants) in addition to system-call obfuscation (renaming system-call signatures). Also, the detection approach [26] may not work with apps that adopt the dynamic permission mechanism already introduced in Android [19], due to its reliance on statically retrieved app permissions. Several other malware detectors combine dynamic profiles with static features (e.g., those based on APIs [18] or static permissions [26], [35]), and thus are vulnerable to the same evasion schemes impeding static approaches.

In this paper, we develop a novel app classification technique, *DroidCat*, based on systematic app-level profiling and supervised learning. DroidCat is developed to not only detect but also categorize Android malware effectively (referred to as the *malware detection* and *malware categorization* mode, respectively). Different from existing learning-based dynamic approaches, DroidCat trains its classification model based on a *diverse* behavioral app profile consisting of features that cover run-time app characteristics in complementary perspectives. DroidCat profiles inter-component communication (ICC) calls and invocations of all methods, including those defined by user code, third-party libraries, and the Android framework, instead of monitoring system calls. Also, it fully handles reflective calls while not using features based on app resources or permissions. DroidCat is thus robust to attacks targeting system calls or exploiting reflection. DroidCat is also robust to attacks targeting specific sensitive APIs, because they are not the only target of method invocations.

The features used in DroidCat were decided based on a *dynamic characterization study* of 136 benign and 135 malicious apps. In the study, we traced the execution of each app, defined and evaluated 122 behavioral metrics to thoroughly characterize behavioral differences between the two app groups. All these metrics measure the *relative occurrence percentage* of method invocations or ICCs, which can never be captured by static malware analyzers.

Based on the study, we discovered 70 discriminating metrics with noticeably different values on the two app groups, and included all of them in the feature set. The 70 features are grouped into three feature dimensions: *structure*, *security*, and *ICC*. By training a model with the Random Forest algorithm [40], DroidCat builds a multi-class classifier that predicts whether an app is benign or malicious from a particular malware family. We extensively assessed *DroidCat* in contrast to *DroidSieve* [23], a state-of-the-art *static* app classification technique, and *Afonso* [27], a state-of-the-art *dynamic* peer approach. The evaluation experiments are conducted on 17,365 benign and 16,978 malicious apps that span the past nine years.

Our evaluation results revealed very-high stability of *DroidCat* in providing competitive classification accuracy for apps evolving over the years: it achieved 97.4% and 97.8% F1 accuracy for malware detection and malware categorization, respectively, all with small variations across the datasets of varying years. Our comparative study further demonstrated the substantial advantages of *DroidCat* over both baseline techniques, with 27% and 16% higher F1 accuracy in the detection and categorization mode, respectively. We also assessed the robustness of our approach against a set of malware adopting various sophisticated obfuscation schemes, along with three different sets of benign apps. Our study showed that *DroidCat* worked robustly well on obfuscated malware with 96% to 97% F1 accuracy, significantly (5% to 46%) higher than the F1 accuracy of either baseline approach. Our analysis of the three techniques' performance with respect to varying decision thresholds further corroborated the consistent advantages of our approach. We also conducted in-depth case studies to assess the performance of *DroidCat* on individual malware families and various factors that may impact its performance.

In summary, we made the following contributions:

- We developed DroidCat, a novel Android app classification approach based on a new, diverse set of features that capture app behaviors at runtime through short app-level profiling. The features were discovered from a dynamic characterization study that revealed behavioral differences between benign and malicious apps in terms of method calls and ICCs.
- We evaluated *DroidCat* via three complementary studies versus two state-of-the-art peer approaches as baselines on 34,343 distinct apps spanning year 2009 through year 2017. Our results showed that *DroidCat* largely outperformed the baselines in stability, classification performance, and robustness in both classification modes, with competitive efficiency.
- We conducted in-depth case studies of *DroidCat* concerning its performance on individual malware families and various factors that affect its classification capabilities. Our results confirmed the consistently high overall performance of *DroidCat*, and additionally showed its strong performance on most of the families we examined. We also identified the most effective learning algorithm and dynamic features for *DroidCat*, and demonstrated the low sensitivity of *DroidCat* to the coverage of dynamic inputs.

```
1  // in ad.notify.Settings::getImei(Context context)
2  // m6 returns 'phone'; cls returns 'android.telephony.TelephonyManager'
3  TelephonyManager tm = context.getSystemService(m6(b, b−1, x|76));
4  Class c = Class.forName(mdb.cls(ci));
5  Method m = c.getMethod(mdb.met(mi),null); //met returns 'getDeviceId'
6  return m.invoke(tm, null);
7
8  // in NotificationApplication::onCreate(); cls returns 'ad.notify.Settings'
9  Class c = Class.forName(mdb.cls(ci)); //met returns 'getImei'
10 Method m = c.getMethod(mdb.met(mi), new Class<Context>[1]);
11 adUrl += m.invoke(null, context);
12
13 // in ad.notify.SmsItem::send(String str, String str2)
14 // cls returns 'android.telephony.SmsManager'
15 Class c = Class.forName(mdb.cls(ci)); //met returns 'sendTextMessage'
16 Method m = c.getMethod(mdb.met(mi), new Class<Object>[5]);
17 SmsManager smsManager = SmgManager.getDefault();
18 m.invoke(smsManager, str, null, str2, null, null)
19
20 // in ad.notify.OperaUpdateActivity::sendSms(String str, String str2)
21 Class c = Class.forName(mdb.cls(ci)); // cls returns 'ad.notify.SmsItem'
22 Method m = c.getMethod("send", new Class<String>[2]);
23 Boolean bs = m.invoke(null, str, str2);
24
25 // in ad.notify.OperaUpdateActivity::threadOperationRun(int i, Object o)
26 SmsItem smsItem=getSmsItem(ad.notify.NotifyApplication.smsIndex);
27 Class c = Class.forName("ad.notify.SmsItem");
28 Field f1 = c.getField("number"); int number = f1.get(smsItem);
29 Field f2 = c.getField ("text"); Object text = f2.get(smsItem);
30 sendSms(number, text);
```

Fig. 1. Code excerpts from a *FakeInst* malware sample: the complex and heavy use of reflection can thwart static code-based feature extraction.

- We released for public access DroidCat and our benchmark suites, to facilitate reproduction of our results and the development/evaluation of future malware detection and categorization techniques.

II. Motivating Example

Malware developers increasingly adopt various obfuscation techniques (e.g., code reflection) to evade security checks [41]. Figure 1 shows five code excerpts in a real trojan-horse sample of a dominant [42] malware family *FakeInst* which sends SMS texts to premium-rate numbers. Except for data encryption (by calling *m6* on line 3), this malware heavily uses reflection to invoke methods including Android APIs so as to access privileged resources such as device id (lines 1–11). In addition, to exploit the SMS service (lines 13–18), it retrieves the text and number needed for the malicious messaging via reflection (line 27–29) and then invokes *sendSms* (line 30) which calls *ad.notify.SmsItem::send* via reflection again (lines 20–23).

While simple reflection with string constants (e.g., lines 22,27,28) can be deobfuscated by static analysis [43], [44] (at extra cost), more complex cases may not be (e.g., lines 4,5,15,16 where the class and method names are retrieved from a database object *mdb*). As a result, static code-based features related to APIs and sensitive flows would not be extracted from the app, and techniques based on such features would not detect the security threats. Also, the malicious behaviour in this sample is exhibited in its code only, not reflected in its resource/asset files (e.g., configuration and UI layout); thus approaches bypassing code analysis (e.g., *DroidSieve* [23]) might not succeed either. Further, malware developers can easily obfuscate app resources too [25]. In these situations, we believe that a robust dynamic approach is a necessary complement for defending against such malware samples.

III. Background

A. Android Applications

Programmers develop Android apps primarily using Java, and build them into app package (i.e., APK) files. Each APK file can contain three software layers: user code (*userCode*), Android libraries (*SDK*), and third-party libraries if any (*3rdLib*). An Android app typically comprises four components as follows [45]: *Activities* which deal with UI and handle user interaction to the device screen, *Services* which handle background processing associated with an application, *Broadcast Receivers* which handle communication between Android OS and applications, and *Content Providers* which handle data storage and management (e.g., database) issues.

B. ICC

Components interact with each other through ICC objects—mainly *Intents*. If both the sender and the receiver of an Intent are within the same app, we classify the ICC as *internal*; otherwise, it is *external*. If an Intent has the receiver explicitly specified in its content, we classify the ICC as *explicit*; otherwise, it is *implicit*.

C. Lifecycle Methods and Callbacks

Each app component follows a prescribed lifecycle that defines how this component is created, used, and destroyed. Correspondingly, developers are allowed to overwrite various *lifecycle methods* (e.g., `onCreate()`, `onStart()`, and `onDestroy()`) to define program behaviors when the events happen. Developers can also overwrite other event handlers (e.g., `onClick()`) or define new *callbacks* to implement extra logic when other interesting events occur.

D. Security-Relevant APIs

There are sensitive APIs that acquire personal information of users like locations and contacts. For example, `Location.getLatitude()` and `Location.getLongitude()` retrieve GPS location coordinates. We consider these APIs *sources* of potential sensitive information flows. There are also output APIs that send data out of the current component via network or storage. We consider them *sinks* of potential sensitive information flows. If an app's execution trace has any (control-flow) paths from sources to sinks, the app might be malicious due to potential sensitive data leakage.

IV. Feature Discovery and Computation

At the core of our approach are its features that are computed from app execution traces. Although from these traces we could extract many features, not every feature is a good differentiator of malicious apps from benign ones. Therefore, with a relatively small dataset (136 benign and 135 malicious apps) (Section IV-A), we first conducted a systematic dynamic characterization study by defining and measuring 122 metrics (Section IV-B) as possible features. Based on the comparison between the two groups of apps, we decided which metrics were good differentiation factors, and thus included them into

our feature set (Section IV-D). The central objective of this exploratory study is to discover the features to be used by *DroidCat*.

A. Benchmarks

Our characterization study used a benchmark suite of both benign apps and malicious apps. To collect benign apps, we downloaded the top 3,000 most popular free apps in Google Play at the end of year 2015 as our initial candidate pool. Next, we randomly selected an app from the pool and checked whether it met the following three criteria: (1) the minimum supporting SDK version is 4.4 (API 19) or above, (2) the instrumented APK file runs successfully with inputs by Monkey [46], and (3) navigating the app with Monkey inputs for ten minutes covers at least 50% of user code (we used our characterization toolkit *DroidFax* [47] which includes a statement-coverage measurement tool directly working with APKs, which instruments each statement in user code to track coverage at runtime). If an app met all criteria, we further checked it with VirusTotal [48] to confirm if the app was benign. As such we obtained 136 benign apps. For malicious apps, we started with the MalGenome dataset [49], the most widely used malware collection. We found 135 apps meeting the above criteria, and confirmed them all as malware using VirusTotal. The APK sizes of our benchmarks vary from 2.9MB to 25.6MB. Recall that this characterization study is exploratory with the goal of identifying robust and discriminating dynamic features for app classification, thus we aimed at a relatively small scale (in terms of the benchmark suite size).

B. Metrics Definition

Based on collected execution traces, we characterized app behaviors by defining 122 metrics in three orthogonal dimensions: *structure*, *ICC*, and *security* (Table I). Intuitively, the more *diversely* these metrics capture app execution, the *more completely* they characterize app behaviors. These metrics measure not only the existence of certain method invocations or ICCs, but also their relative occurrence frequencies and distribution. For brevity, we will only discuss a few metrics in the paper; detailed description of all metrics can be found at http://chapering.github.io/droidfax/metrics.htm.

Structure dimension contains 63 metrics on the distributions of method calls, their declaring classes, and caller-callee links. 31 of these metrics describe the distributions of all method calls among three code layers (i.e. user code, third-party libraries, and Android SDK), or among different components. The other 32 metrics describe the distributions of a specific kind of methods—*callbacks* (including lifecycle methods and event handlers). One example *Structure* metric is the percentage of method calls to the SDK layer. Another example is the percentage of `Activity` lifecycle callbacks over all callbacks invoked.

ICC dimension contains 7 metrics to describe ICC distributions. Since there are two ways to classify ICCs, *internal* vs. *external*, and *implicit* vs. *explicit*, enumerating all possible combinations leads to four metrics. The other three metrics are defined based on the type of data contained in the *Intent* object associated with an ICC: the Intent carries data in either its `URI` or `extras` field only, or both. One example *ICC* metric is the percentage of ICCs that carry data through `URI` only. Another example is, out of all ICCs exercised, the percentage that are *implicit* and *external*.

Security dimension contains 52 metrics to describe distributions of sources, sinks, and the reachability between them through method-level control flows. The reachability is used to differentiate from all exercised sources/sinks that are risky. If a source reaches at least one sink, it is considered a *risky source*. Similarly, a *risky sink* is reachable from at least one source. Both of these indicate security vulnerabilities, because sensitive data may be leaked when flowing from sources to sinks. For example, a *Security* metric is the percentage of method calls targeting sources over all method calls. Another example is the percentage of exercised sinks that are risky.

C. Metrics (Feature) Computation

To compute the 122 metrics of an Android app, we first instrumented the program for execution trace collection. Specifically, we used Soot [50] to transform each app's APK along with the SDK library (`android.jar`) into Jimple code (Soot's intermediate representation), and then inserted in the Jimple code probes to run-time monitors for tracing every method call (including those targeting SDK APIs and third-party library functions) and every ICC *Intent*. We also labeled additional information for instrumented classes and methods to facilitate metric computation. For instance, we marked the component type for each instrumented class, the category of each instrumented callback, and the source or sink property of each relevant SDK API. To decide the component type of a class such as `Foo`, we applied Class Hierarchy Analysis (CHA) [51] to identify all the superclasses. If `Foo` extends any of the four known component types such as `Activity`, its component type is labeled accordingly. We used the method-type mapping list in [47] to label the category of callbacks and the source/sink property of APIs. Exception handling and reflection are two widely used Java constructs. Accordingly, our instrumentation fully tracks two special kinds of method and ICC calls: (1) those made via *reflection*, and (2) those due to *exceptional control flows* [52] (e.g., calls from `catch` blocks and `finally` blocks).

Next, we ran the instrumented APK of each app on an Android emulator [53] to collect execution traces, which include all method calls and ICCs exercised. Note that we do not monitor OS-level system calls, because we want DroidCat to be robust to any attacks targeting system calls. Our instrumentation is not limited to sensitive APIs, either. By ensuring that sensitive APIs are not the only target scope of method-call profiling, we make DroidCat more robust against attacks targeting sensitive APIs. Prior work shows that even without invoking malicious system calls or sensitive APIs, some malicious apps still can conduct attacks by manipulating other apps via ICCs [54]–[57]. Thus, we also trace ICCs to further reveal behavioral differences between benign and malicious apps.

TABLE I
METRICS FOR DYNAMIC CHARACTERIZATION AND FEATURE SELECTION

Dimension	# of Metrics	Exemplar Metric	# of Substantially Disparate Metrics	# of Noticeably Different Metrics
Structure	63	The percentage of method calls whose definitions are in user code.	15	32
ICC	7	The percentage of external implicit ICCs.	2	5
Security	52	The percentage of sinks reachable by at least one path from a sensitive source	19	33
Total	**122**		**36**	**70**

To characterize the dynamic behaviors of apps, we need to run each instrumented app for a sufficiently long time using various inputs to cover as many program paths as possible. Manually entering inputs to apps is very inefficient. In order to quickly trigger diverse executions of an app, we used Monkey [46] to randomly generate inputs. To balance between efficiency and code coverage, we set Monkey to feed every app for ten minutes. (*DroidCat* only executes each app for five minutes; we investigated the effect of dynamic coverage on the effectiveness of DroidCat in Section VII-C.) Once the trace for an app is collected via the probed run-time monitors, most of the 122 metrics are computed through straightforward trace statistics. The metrics involving risky sources/sinks are calculated through a dynamic call graph built from the trace, which facilitates reachability computation.

D. Metrics (Feature) Selection

To identify any metric that well differentiates between the two app groups, we measured the value of each metric on every benchmark app, and then computed the mean values separately for all benign and malware benchmarks. If a metric had a mean value difference greater than or equal to 5%, we considered the behavioral profile of the two groups ***substantially disparate*** with respect to the metric. If a metric had a difference greater than or equal to 2%, we said the behavioral profile was ***noticeably different*** with respect to the metric. We experimented with various thresholds chosen heuristically, and found these two (5% and 2%) reasonably well represent two major levels of differentiation between our malware and benign samples.

As shown in Table I, by comparing mean metric values across app groups, we found 36 substantially disparate metrics, and 70 noticeably different metrics. We show the top 10 differentiating metrics in Figure 2. There are ten metrics listed on the Y-axis, and the X-axis corresponds to mean metric values, which vary from 0% to 100%. Each metric listed on Y-axis corresponds to: a red bar to show the mean value of all malicious apps, and a green bar to represent the mean of all benign ones. The whisker on each bar represents the standard error of the mean. Empirically, these 10 metrics best demonstrate the behavioral differences between malicious and benign apps.

In the *structure* dimension, malicious apps call fewer methods defined in SDK and more methods defined in user code, and involve more callbacks relevant to UI. This indicates that user operations may trigger excessive or unexpected computation. For instance, on average, SDK APIs account for 80% function calls in the execution of malicious apps, but 91% function calls in benign apps' executions. Concerning caller-callee links, For instance, `SDK->SDK` accounts for 60% of method calls in benign apps, but only accounts for 8% in malware. In comparison, `UserCode->SDK` and `3rdLib->SDK` are the most frequent caller-callee links in malware. In terms of callbacks, malicious apps involve 92% `Activity` lifecycle callbacks and 84% `View` event handlers. Both numbers are significantly larger than their counterparts in benign apps: 73% and 56%. However, malicious apps involve a much smaller portion of `system status` event handlers than benign ones, which is 5% vs. 23%.

> **Implication 1:** *Malware tended to invoke SDK APIs more often from user code or third-party libraries, and define more UI callbacks indicating that user operations on them may trigger excessive/unexpected computation.*

In the *ICC* dimension, malware involves more *external explicit* ICCs with more URI data carried by *Intents*. This means that malware uses explicit ICCs more often to potentially exploit specific external components, or sends more URI data via ICCs to disseminate potentially malicious URIs. Specifically, on average, malware executions contain 42% *external explicit* ICCs, while benign app executions only contain 10%. In reality, to communicate with external components, benign apps usually leverage *implicit* instead of *explicit* ICCs. This is reasonable because developers usually know little about the run-time environment of a user's phone, such as what external components are available to receive *Intents*. Therefore, *external implicit* ICCs can be used to flexibly detect all available potential receivers before any *Intent* is sent. In contrast, the frequently used *external explicit* ICCs by malware seem suspicious. Among data-carrying ICCs, malicious apps have 30% ICCs carrying URI data, while this number is only 10% in benign apps.

> **Implication 2:** *Malware may use more explicit ICCs to potentially attack specific external components, or disseminate potentially malicious URIs more often via ICCs.*

In the *security* dimension, malware invokes more risky source APIs, but fewer logging sink APIs. By executing more risky sources, malware may cause sensitive data leakage. As shown in Figure 2, among all invoked source APIs, malicious apps have 39% risky sources, while benign ones only have 27%. Among all invoked sink APIs, logging sinks accounts for 21% for benign apps, but 9% for malware, which means

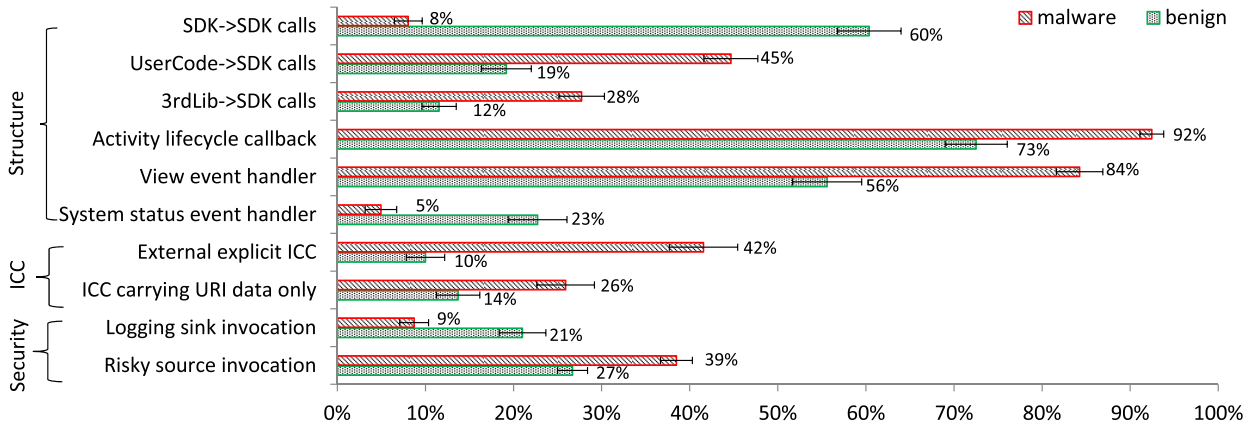

Fig. 2. Top-10 differentiating metrics between malware and benign apps revealed by our exploratory characterization study.

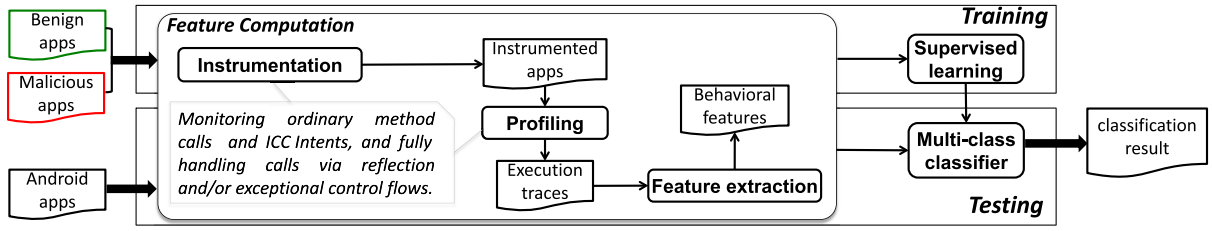

Fig. 3. *DroidCat* overview: it trains a multi-class classifier using benign and malicious apps and then classifies unknown apps.

users may log more frequently in benign apps to secure their sensitive data operations. Besides, we also observe that sink APIs like *Network access* and *Messaging* (SMS/MMS), are invoked more frequently by malware than benign apps.

Implication 3: *Malicious apps exhibit less logging practice than benign ones. They execute more risky sources, which may lead to sensitive data leakage.*

V. The *DroidCat* Approach

Based on our characterization study, we developed DroidCat, an app classification approach leveraging systematic profiling and supervised learning, to decide whether a given app is benign or belongs to a particular malware family. As shown in Figure 3, there are two phases in our approach: training and testing. For training, DroidCat takes both benign and malicious apps as inputs. For each app, it computes the 70 metrics as behavioral features as described above. The features are then provided to supervised machine learning to train a multi-class classifier [58], using the Random Forest algorithm [40]. For testing, given an arbitrary app, DroidCat computes the same set of behavioral features and then feeds these features to the classifier to decide whether the app is benign or a member of a malware family.

We implemented the learning component of DroidCat in Python, using the Scikit-learn toolkit [59], to train and test the classifier. We provided open source the entire DroidCat system (including the feature computation component) and our datasets at https://chapering.github.io/droidcat.

TABLE II
Main Datasets Used in Our Evaluation Studies

Dataset	Period	Benign apps		Malware		
		Source	#Apps	Source	#Apps	#Families
D1617	2016-2017	GP,AZ	5,346	VS,AZ	3,450	153
D1415	2014-2015	GP,AZ	6,545	VS,AZ	3,190	163
D1213	2012-2013	GP,AZ	5,035	VS,AZ,DB,MG	9,084	192
D0911	2009-2011	AZ	439	VS,AZ,DB,MG	1,254	88

VI. Evaluation

For a comprehensive assessment of *DroidCat*'s capabilities in malware detection and categorization, we conducted three complementary evaluation studies. In *Study I*, we aim to gauge the *stability* of *DroidCat* by applying it to four longitudinal datasets (across nine years) to see how well it works for apps in the evolving Android ecosystem. In *Study II*, we compare the prediction performance of *DroidCat* against state-of-the-art peer approaches, including a static and a dynamic approach, by applying the three techniques to two newest datasets among the four. In *Study III*, we measure the robustness against obfuscation of the three techniques using an obfuscation benchmark suite along with varying benign sample sets. We first describe our evaluation datasets in Section VI-A and procedure in Sections VI-B and VI-C, and then present the evaluation results of the three studies in Sections VI-D, VI-E, and VI-F, respectively.

A. Datasets

Table II lists the various datasets (named by ids in the first column) used in our evaluation experiments. Each dataset includes a number (fourth column) of benign apps and a

number (sixth column) of malware in a number (the last column) of families. Apps in each of these four datasets are all from the same period (range of years, in the second column), according to the *age* of each app measured by its *first-seen date* we obtained from VirusTotal [48]. The table gives the sources (third and fifth columns) of the samples. AndroZoo (AZ) [60] is an online Android app collection that archives both benign and malicious apps. Google Play (GP) [61] is the official Android app store. VirusShare (VS) [62] is a database of malware of various kinds including Android malware. The Drebin dataset (DB) is a set of malware shared in [15], and (Malware) Genome (MG) is a malware set shared in [49]. We also used an obfuscation benchmark suite along with benign apps from AZ and GP for Study III (as detailed in Section VI-F).

Concerning the overhead of dynamic analysis, we randomly chose a subset of samples from each respective source, except for the MG dataset which we used all samples therein given its relatively small size. A few apps were discarded during the benchmark collection, because they could not be unzipped, were missing resource files (e.g., assets), or could not be successfully instrumented, installed, or traced. In particular, for the D1617 and D1415 datasets which we used for a comparative study (Study II), we also discarded samples with which any of the three compared techniques failed in its analysis. We did not apply any of the selection criteria in the characterization study (Section IV-A). The numbers (of samples) listed in the table are those of the remaining samples actually used in our studies. In all, our datasets include 17,365 benign apps and 16,978 malware, for a total of 34,343 samples. The age of these samples ranged across the past nine years (i.e., 2009–2017). We note that there were not exactly the same samples shared by any two of our datasets (although some samples in one dataset might be the evolved versions of samples in another). In cases where the original datasets (e.g., DB and MG) overlap, we removed the common samples from either dataset (e.g., we dismissed MG samples from the original DB dataset). We also ensured that these four datasets did not overlap with the dataset used in our characterization study—we excluded the 136 GP apps and 135 MG malware used in that study when forming the four datasets in Table II. The reason was to avoid relevant biases (e.g., overfitting) since the characterization dataset was used for developing/tuning *DroidCat* (i.e., for discovering/selecting its features).

B. Experimental Setup

For the baseline techniques, we consider both static and dynamic approaches to Android malware prediction. In particular, we compare *DroidCat* to *DroidSieve*, a state-of-the-art *static* malware detection and categorization approach. *DroidSieve* characterizes an app with resource-centric features (e.g., use permissions extracted from the manifest file of an APK) in addition to code (syntactic) features, and then uses these features to train an Extra Trees model that is later used for predicting the label of a given app. We chose *Afonso* as another baseline technique, a state-of-the-art *dynamic* approach for app classification. *Afonso* traces in an app the invocations of Android APIs and system calls in specified lists, and uses the call frequencies to differentiate malware from benign apps based on a Random Forest model.

To enable our comparative studies, we obtained the feature computation code from the *DroidSieve* authors and implemented the learning component. With help of the authors, we were able to reproduce the performance results against part of the datasets used in the original evaluation of this technique hence gained confidence about the correctness of our implementation. We implemented the *Afonso* tool according to the API and system call lists provided by the authors. We developed DroidCat as described earlier. To compute the features and prediction results with the two baselines, we followed the exact settings as described in the respective original papers (and by the authors of *DroidSieve* via emails for which we initially had difficulties getting performance results close to originally reported ones). In particular, to produce the execution traces required by *Afonso*, we ran each app on a Nexus One emulator with API Level 23, 2G RAM, and 1G SD storage for 5 minutes as triggered by Monkey random inputs (same as for *DroidCat* as described in Section IV-C). All of our experiments were performed on a Ubuntu 15.04 workstation with 8G DDR and a 2.6GHz processor.

C. Methodology

We evaluated *DroidCat* in each of its two working modes: (1) *malware detection*, in which it labels a given app as either benign or malicious, and (2) *malware categorization*, in which it labels a given malware with the predicted malware family. To facilitate the assessment of *DroidCat* in these two different modes, we simply treat *DroidCat* as a multi-class classifier, with different number of classes to differentiate in different modes (e.g., two classes in the detection mode, and two or more classes in the categorization mode).

For Study I, we ran four tests of *DroidCat*, each using one of the four datasets (D0911 through D1617). For Study II, we executed *DroidCat* and the two baselines on D1617 and D1415, because these two are the most recent datasets. We used three obfuscation datasets for Study III, in which we ran three tests of the three techniques accordingly.

In each test of these three studies, we sorted apps of *each class* by their age (first-seen date) and split the apps by the date at 70 percentile, and then we held out the 30% newest ones from each class for testing while using the rest for training. We used this hold-out validation in order to avoid overfitting [63]: samples used for fitting a classification model are never used in validating the model. Our evaluation studies did not involve any re-sampling (as in cross validation) either, so as to avoid causing biases in the validation results [64]. This experiment design also makes sure that we never use a model trained on newer samples to test older samples—doing so would not be sensible with respect to the practical use scenarios of a malware detector and the evolution of apps.

The three studies share the same set of metrics for evaluating the performance of the compared techniques in predicting apps of each class. We compute these metrics for each class and

then average the metrics values among all classes to obtain the overall performance of each technique. Specifically, for each class C_i, we assessed a technique's performance with the following three metrics:

Precision (P) measures among all the apps labeled as "C_i" by the technique, how many of them actually belong to that class.

$$P_i = \frac{\text{\# of apps belonging to } C_i}{\text{Total \# of apps labeled as "}C_i''}. \quad (1)$$

Recall (R) measures among all apps belonging to C_i, how many of them are labeled by the technique as "C_i".

$$R_i = \frac{\text{\# of apps labeled as "}C_i''}{\text{Total \# of apps belonging to } C_i}. \quad (2)$$

F1 score (F1) is the harmonic mean of precision and recall. It can be interpreted as a weighted average of the precision and recall.

$$F1_i = \frac{2 * P_i * R_i}{P_i + R_i}. \quad (3)$$

Note that the technique only labels apps with "C_1", "C_2", ..., and never uses any label like "not C_i". To facilitate the metrics computation with respect to a particular class like C_1, we treat all apps with other labels like "C_2", "C_3", ... as "not C_i". For example, suppose there are 10 apps belonging to C_3. The technique labels 11 apps with "C_3", but only 8 of them actually belong to C_3. As a result, $P_3 = 8/11 = 73\%$ because only 8 out of the 11 "C_3"-labeled apps are identified correctly. $R_3 = 8/10 = 80\%$ because only 8 out of the 10 C_3 apps are labeled correctly. $F1_3 = 2*73\%*80\%/(73\%+80\%) = 76\%$.

With the above effectiveness metrics computed for each class, we further evaluated the overall effectiveness of the technique by computing the weighted average among classes. Intuitively, the larger the number of apps in a class, the more weight its effectiveness metrics should have. The malware families vary greatly in size, so we weight each family's contribution to the average by its relative size to the entire testing set. Formally, if we use Γ to represent P or R, and use n_i to represent the number of testing samples in C_i, then the **overall effectiveness** in terms of precision and recall can be computed for N classes with

$$\Gamma_{overall} = \frac{\sum_{i=1}^{N} \Gamma_i * n_i}{\sum_{i=1}^{N} n_i}. \quad (4)$$

Finally, the **overall F1** is computed with:

$$F1_{overall} = \frac{2 * P_{overall} * R_{overall}}{P_{overall} + R_{overall}} \quad (5)$$

To further assess the capabilities of our approach versus the baselines, we compute the receiver operating characteristic (ROC) curve for each technique and relevant dataset it applied to. These curves show how a binary classifier performs with respect to varying decision thresholds, as opposed to one (default) threshold associated with an F1 score. They also depict various tradeoffs between true positive and false positive rates. Thus, the curves complement the three accuracy metrics (P, R, and F1), together constituting a comprehensive measure of the classification performance. In particular, we used the prediction probabilities produced by the classifier to compute these curves. For multi-class classification (i.e., the malware categorization mode), we wrap the classifier with a one-vs-all classifier to compute the curve for each class, and then average all per-class curves to produce an averaged curve. For each ROC curve, we also report the area under curve (AUC) as a summary metric for the ROC.

TABLE III

DroidCat PERFORMANCE FOR MALWARE DETECTION AND CATEGORIZATION

Dataset	Detection			Categorization		
	P	R	F1	P	R	F1
D1617	99.31%	99.27%	99.28%	94.79%	94.74%	94.54%
D1415	97.26%	97.09%	97.16%	97.84%	97.75%	97.70%
D1213	96.38%	96.04%	96.12%	99.73%	99.71%	99.70%
D0911	97.19%	96.96%	97.00%	99.48%	99.43%	99.44%
mean	97.53%	97.34%	97.39%	97.96%	97.91%	97.84%
stdev	1.25%	1.37%	1.34%	2.27%	2.28%	2.38%

D. Study I: Performance Stability

Table III lists the classification performance of *DroidCat* in terms of the three metrics we defined earlier: precision (P), recall (R), and F1-measure accuracy ($F1$). Each row gives the results for the two working modes of *DroidCat* against one of the four datasets used in this study. For instance, on the D1617 dataset (apps in year 2016 through year 2017), *DroidCat* had 99.31% precision and 99.27% recall for detection (binary classification), and 94.54% F1 for categorizing malware into families. The last two rows show the mean performance metrics values over the four datasets of *DroidCat* in each of the two modes, and the associated standard deviation (stdev) of the mean. For instance, for malware detection, *DroidCat* achieved an average F1 accuracy of 97.39% with a standard deviation of 1.34% across all the four datasets spanning the past nine years.

As shown, the performance of *DroidCat* depended on both the dataset it was applied to and the working mode. Specifically, it had the highest accuracy (99.28%) on the newest (D1617) dataset for malware detection, yet performed the best (with 99.70% F1) on the second oldest (D1213) dataset for malware family categorization. Similarly, the worst-case performance also was associated with different mode for different dataset: lowest detection accuracy (96.12%) was for D1213 while lowest categorization accuracy (94.54%) was seen by D1617.

Intuitively, it is more challenging to differentiate more classes. Our results show that *overall DroidCat* performed almost equally well (97.39% versus 97.84%) for detection and categorization modes. While our features were discovered originally from a characterization study that only concerned two classes (malware versus benign apps), this overall contrast suggests that the features could also well differentiate among varied malware families. On the other hand, however, we observed that over the years the categorization performance appeared to decline gradually while the performance for detection did not. What this implies is that the features seem to be less robust against the evolution of malware than for

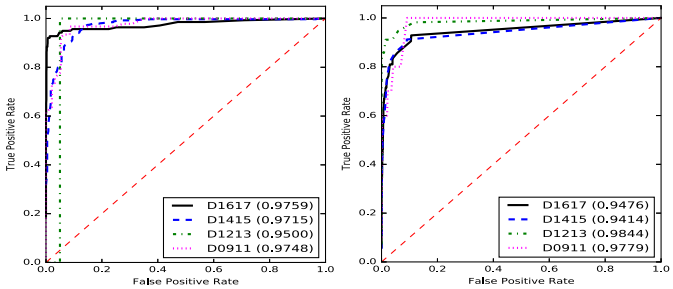

Fig. 4. DroidCat ROC curves with AUCs for malware detection (left) and categorization (right) on four datasets (D0911 through D1617).

differentiating benign apps from all kinds of malware as a whole.

Figure 4 depicts the ROC curves and associated AUCs (in the parentheses) of *DroidCat* on the four datasets for the two modes. The results show that *DroidCat* worked the best for the newest (D1617) dataset in the detection mode, with an AUC of almost 0.98 (close to the ideal case of 1.0). On the other three datasets, our classifier was also highly accurate with different thresholds. The right chart indicates that it performed the worst for categorizing malware in the D1415 dataset. Nevertheless, even this lowest performance was still highly competitive (0.94 AUC). The categorization accuracy was noticeably higher on other datasets.

Conclusions: Overall, *DroidCat* exhibited highly competitive performance for any mode and dataset, with F1 ranging from 94.54% to 99.73%, and AUC from 0.94 to 0.98. Importantly, *DroidCat* appeared to be quite *stable* in classifying apps seen in the 9-year span we studied, as supported by several observations. First, the standard deviations of performance metrics over the span were generally small, suggesting *DroidCat* worked for both old and new datasets with promising performance. Second, it is noteworthy that the performance of *DroidCat* was not much affected by largely varying dataset sizes or the imbalance between malware and benign samples: nor did there exist a clear correlation between the performance and sample sizes or the imbalances. Third, a larger number of families did not necessarily make *DroidCat* perform worse in malware categorization either. In comparison between its two modes, *DroidCat* tended to be even more stable for malware detection (1.34% standard deviation) than for malware categorization by families (2.38% standard deviation).

Finding 1: *DroidCat achieved mean F1 accuracy of 97.39% and 97.84%, and AUC of 0.95-0.98 and 0.94-0.98, for malware detection and categorization, respectively. It was also stable in classifying apps from different years within 2009–2017, evidenced by small standard deviations in F1 of 1.34-2.38% across the nine years.*

E. Study II: Comparative Classification Performance

In this study, we aim to compare our approach to the two baselines in their classification capabilities. In particular, we conducted comparative analysis in the two classification

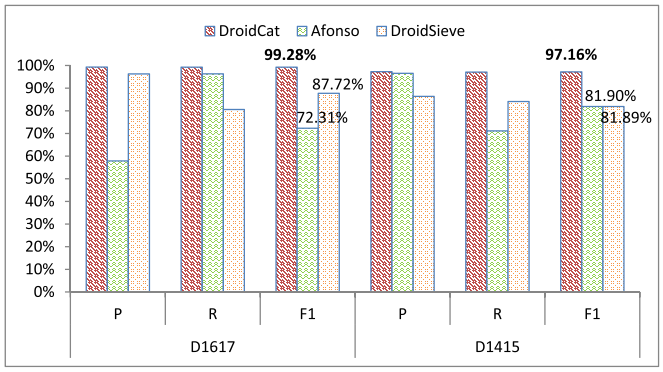

Fig. 5. *DroidCat* versus baselines for malware detection.

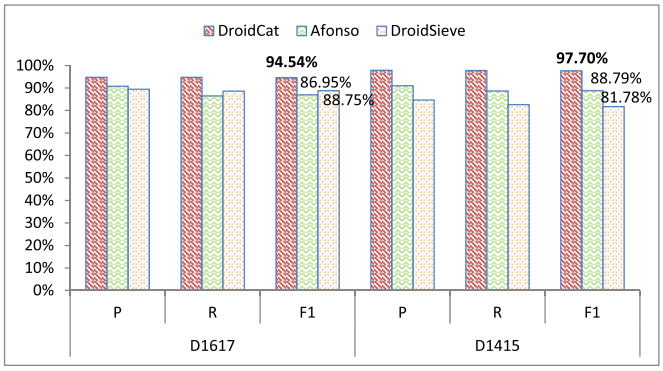

Fig. 6. *DroidCat* versus baselines for malware categorization.

working modes considered. For malware detection, Figure 5 shows the contrast among the three techniques (the three bars in each group) in terms of the three performance metrics (x axis), for the two datasets (D1617 and D14515) used in this study. Our results revealed considerable advantages of *DroidCat* over the two state-of-the-art techniques: on the D1617 dataset, *DroidCat* had 3% higher precision, 19% higher recall, and 11% higher F1 accuracy than the better-performing baseline *DroidSieve*. The advantage of *DroidCat* over the peer dynamic approach *Afonso* was even greater (27% higher F1). On the D1415 dataset, the improvement of *DroidCat* over the better-performing baseline *Afonso* was also substantial (15% higher F1), albeit the gap between the two baselines was quite small (0.1%). Across both datasets, *DroidSieve* was more accurate for malware detection than *Afonso*.

Figure 6 depicts the contrast among the three techniques in their performance in categorizing malware into families. As in the detection mode, *DroidSieve* outperformed *Afonso* for the D1617 dataset, while for the D1415 dataset *Afonso* was more accurate. However, considering the gaps in F1, *Afonso* was overall more competitive than *DroidSieve*, opposite to the contrast in the detection mode. On the other hand, the results clearly show the significant merits of our approach over both baselines. Relative to the better-performing peer approach, *DroidCat* had about 6% higher F1 accuracy on the D1617 dataset and over 9% higher accuracy on the D1415 dataset, although the gaps were lesser than those in the detection mode. Interestingly, while *Afonso* was originally

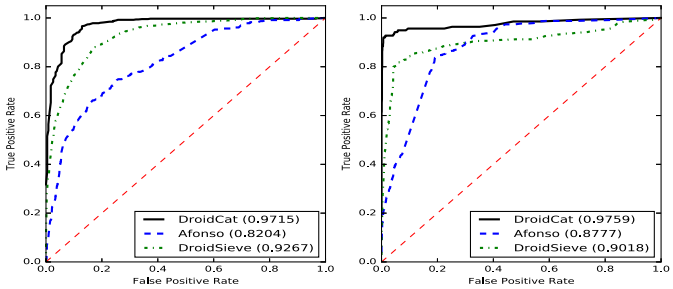

Fig. 7. ROC curves with AUCs of DroidCat versus baselines for malware detection on datasets D1415 (left) and D1617 (right).

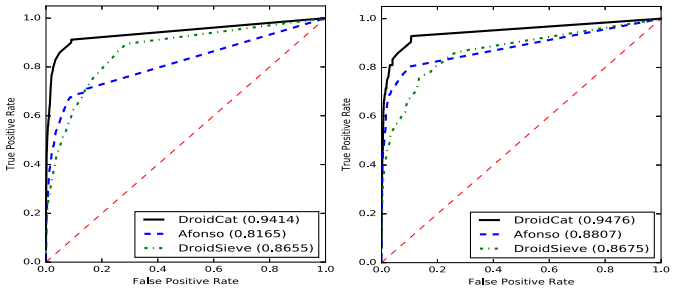

Fig. 8. ROC curves with AUCs of DroidCat versus baselines for malware categorization on datasets D1415 (left) and D1617 (right).

evaluated for malware detection only [27], our results revealed that it performed more accurately for malware categorization. The performance of *DroidSieve* varied very slightly between these two working modes, though.

Across the two datasets, our results also revealed the superiority of *DroidCat* to both baselines in *stability*. *Afonso* achieve noticeably higher performance on the older (D1415) dataset than on the newer (D1617) dataset, regardless of the classification modes (although the gap was smaller for family categorization). *DroidSieve* was similarly unstable to *Afonso*, if not worse, although it performed significantly better on D1617 (than on D1415, also in both working modes). In addition, both baselines had achieved considerably higher performance (e.g., constantly over 90% F1) on other datasets (mostly older than the two we used here) [23], [27], which further suggest their likely instability. On the other hand, we have been able to obtain very similar performance numbers to those originally reported with respect to the originally used datasets. Thus, both baseline techniques seem to be more likely to be overfitted to particular datasets relative to our technique, which again backs up the stability advantage of *DroidCat* over the two baselines.

Figures 7 and 8 depict the ROC curves and corresponding AUCs (shown in parentheses) of *DroidCat* versus the baselines on the two datasets used in this study, for detection and categorization, respectively. The results clearly show the advantages of our approach over the two baselines, regardless of the dataset and classification mode considered, as also evidenced by the constantly higher AUCs of *DroidCat*. In fact, the curves revealed that *DroidCat* was more accurate than the baselines *at any decision threshold*. Between the two baselines, *Afonso* outperformed *DroidSieve* only on one dataset for categorization, while in other cases *DroidSieve* was more accurate for varying thresholds. This contrast was consistent with how these prior approaches compared in terms of F1 accuracy.

TABLE IV
DATASETS USED IN THE STUDY ON ROBUSTNESS (STUDY III)

Dataset	Benign apps				Malware (from *Praguard*)		
	Period	Source	#Apps	obf%	Period	#Apps	obf%
OBF1617	2016-2017	GP,AZ	3,196	57.38%	2010-2012	1,214 (59 families)	100%
OBF1415	2014-2015	AZ	4,462	25.59%			
OBF1213	2012-2013	AZ	4,804	12.57%			

Conclusions: Overall, *DroidCat* surpassed both baseline techniques in each of the two classification working modes, and in most cases the advantages were highly significant. Across the two modes, the performance gap between the baselines and *DroidCat* was even greater in the malware detection mode. Between the two baselines, *DroidSieve* appeared to be more competitive for malware detection, while for malware family categorization *Afonso* was considerably more accurate on one dataset. *Afonso* appeared to be more effective for malware categorization than for malware detection, while *DroidSieve* had similar performance between these two modes. Knowing that *Afonso* classification was based on call frequencies (i.e., sheer counts of calls), the substantial performance merits of *DroidCat* over *Afonso* suggest that relative statistics of calls and call distributions are more effective than sheer counts. In addition, both baselines tended to be much less stable than our approach, as evidenced by the variations of performance across varied datasets.

Finding 2: *DroidCat outperformed the state-of-the-art techniques compared, with up to 27% and 16% higher F1, and 0.15 and 0.13 greater AUC, for malware detection and categorization, respectively. DroidCat also appeared to be noticeably more stable than the two baseline techniques over time when achieving competitive performance.*

F. Study III: Robustness

For this study, we used three obfuscation datasets as described in Table IV. We intended to only use benign apps from AZ, but had to include GP apps of year 2017 because of the shortage of AZ apps of that year. For malware, we focused on those from Praguard [24], an obfuscation benchmark suite including the 1260 MG apps and 237 malware from the Contagio dataset [65] all transformed using multiple obfuscation schemes combined. Praguard has several subsets each using a different scheme combination. We used the most obfuscated subset while removing those corresponding to the 135 MG malware used in our characterization study (to avoid possible overfitting biases). From the remaining 1362 apps, we were able to compute features of 1214 apps for both *DroidCat* and the two baselines. The table also lists the percentage of apps that are obfuscated (*obf%*). All the apps' age was again

TABLE V
ROBUSTNESS OF *DroidCat* VERSUS BASELINES

Technique	Perf.	Detection				Cate.
		OBF1617	OBF1415	OBF1213	Average	
DroidCat	P	97.46%	96.86%	96.40%	96.85%	97.26%
	R	97.34%	96.71%	96.19%	96.69%	97.07%
	F1	**97.33%**	**96.66%**	**96.13%**	**96.64%**	**97.06%**
Afonso	P	98.43%	71.07%	85.37%	83.91%	54.55%
	R	52.07%	86.73%	93.81%	79.89%	56.47%
	F1	**68.11%**	**78.13%**	**89.39%**	**79.59%**	**51.03%**
DroidSieve	P	86.44%	87.98%	85.08%	86.48%	91.81%
	R	83.34%	85.94%	81.77%	83.67%	93.49%
	F1	**80.51%**	**82.67%**	**76.09%**	**79.62%**	**92.27%**

determined by their first-seen dates. Note that even benign apps were increasingly obfuscated, which further justifies the importance of app classifiers being robust against obfuscation. We applied the methodology as described in Section VI-C. In addition, we confirmed that in each data split (based on 70-percentile first-seen dates) there are non-trivial portions of obfuscated samples of each class in both training and testing sets. As for Studies I and II, we ensured that the three benign sets did not overlap.

Table V lists classification performance (*Perf.*) numbers (P, R, and F1) of *DroidCat* versus *Afonso* and *DroidSieve* in the detection and categorization (*Cate.*) modes on the three datasets (second row) used in this study. The F1 numbers are highlighted in boldface. The numbers in the sixth column are the averages over the three datasets, weighted by the dataset sizes. For instance, on the OBF1617 dataset, *DroidCat* had a 97.33% F1 accuracy, versus 68.11% by *Afonso* and 80.51% by *DroidSieve*, all in the *detection* mode. In the *categorization* mode, the three techniques had the same performance across the three datasets since they all use the same malware set (i.e., the 1214 Praguard malware).

Our results show that *DroidCat* largely surpassed the baseline techniques in any performance metric on any of the three datasets for malware detection. In the malware categorization mode, *DroidSieve* achieved an F1 (92%), the closest to that of *DroidCat* (97%) among all our comparative experiments. We note that *DroidSieve* achieved 99% F1 for categorizing the MG subset of our malware set here, and over 99% F1 for detecting the MG malware from benign-app sets different from ours [23]. The considerable drop in accuracy, when 237 more malware and many different benign apps were trained and tested, suggested the potential overfitting of this technique to particular datasets. On the other hand, compared to our results in Study II, *DroidSieve* performance did not change much due to obfuscation. Thus, the technique tended to be obfuscation-resilient indeed, and its performance variation with respect to the original evaluation results in [23] seems to be mainly attributed to its instability.

Afonso appeared to be resilient against obfuscation too, but only for malware detection. The substantial performance drop (by over 30% in F1) because of obfuscation indicates its weak obfuscation resiliency for categorizing malware. Meanwhile, its considerable performance variations for malware detection across the three datasets corroborate the instability of this technique.

In contrast, *DroidCat* tended to be both robust and stable. The robustness was evidenced by the small difference in performance metrics between this study and Study II. Its performance variations across the three datasets were also quite small, showing its stability even in the presence of complicated obfuscation.

We also computed the ROC curves and AUCs of the three techniques on each of the three datasets. The contrasts between *DroidCat* and the two baselines was similar to those seen in Study II. The AUC numbers (0.97–0.99 for *DroidCat*) show considerable advantages of our approach as well (0.05 and 0.09 greater AUC than any baseline for detection and categorization, respectively). In all, the ROC results confirmed that *DroidCat* is robust to various obfuscation schemes, with respect to varying decision thresholds, more than the two baselines.

Conclusions: On overall average, *DroidCat* achieved a 96.64% F1, compared to 79.59% by *Afonso* and 79.62% by *DroidSieve* in the detection mode. In the categorization mode, *DroidCat* also significantly outperformed the two baseline techniques, with 5–46% higher F1. ROC results corroborated the robustness merits of our approach, compared to the baselines. In absolute terms, the accuracy and AUC numbers revealed that *DroidCat* can work highly effectively with obfuscated apps.

Finding 3: *DroidCat exhibited superior robustness to both state-of-the-art techniques compared, by achieving 96% to 97% F1 accuracy on malware that adopted sophisticated obfuscation schemes along with varying sets of benign apps, significantly higher than the two baselines.*

VII. IN-DEPTH CASE STUDIES

We have conducted in-depth case studies on a subset of our datasets to access the capabilities of our approach in classifying apps with respect to individual malware families. Through the case studies, we also investigated the effects of various design factors on the performance of *DroidCat*. We summarize our methodology and findings below. Further details can be found in our technical report on *DroidCat* [66].

A. Setup and Methodology

We started with the characterization study dataset and added into it malware samples from years 2016 and 2017 in the wild, resulting in 287 benign apps and 388 malware. The majority of these apps adopted various obfuscation strategies (e.g., reflection, data encryption, and class/method renaming). The malware samples were in 15 popular families, including *DroidDream*, *BaseBridge*, and *DroidKungFu* which were among the most evasive families according to a prior study [24], and *FakeInst* and *OpFake* which are known to combine multiple obfuscation schemes (renaming, string encryption, and native payload). We applied the same hold-out validation procedure, and the same three accuracy metrics (P, R, F1) as used in the evaluation experiments (Section VI-C).

B. Results

For malware categorization, *DroidCat* performed perfectly (with 100% F1) for the majority (11) of the (15) studied families. In particular, these 11 families include the three that were previously considered highly evasive: *DroidDream*, *BaseBridge*, and *DroidKungFu*. Previous tools studied [24] achieved no more than 54% detection rate (i.e, the recall metric in our study) on these three families. By weighted average, over all the 15 classes, *DroidCat* achieved 97.3% precision, 96.8% recall, and 97.0% F1. In the malware detection mode, *DroidCat* worked even more effectively, with 97.1% precision, 99.4% recall, and 98.2% F1. These results are largely consistent with what we obtained from the extensive evaluation studies (Studies I through III).

C. Effects of Design Factors

1) Feature Set Choice: We investigated several alternative feature sets, including the full set of 122 metrics, the set of metrics in each of the three dimensions, and the set of 36 substantially disparate metrics (see Table I). We found that *D** (the default set of 70 features used by *DroidCat*) worked the best, suggesting that adding more features does not necessarily improve classification performance. The *Structure* features had significantly better effectiveness than *ICC* and *Security* features.

2) Most Important Dynamic Features: To see which specific features are the most important to our technique, we computed the importance ranking [23], [67] of the 70 features used by DroidCat. We found that *Structure* features consistently dominated the top list, especially when there were a greater number of classes that our classifier had to differentiate. In particular, two subcategories of *Structure* features contributed the most: (1) distribution of method/class invocation over the three code layers, and (2) callback invocation for lifecycle management. The *Security* features were generally less important, with the *ICC* features being the least important. Among all *Security* features, those capturing risky control flows and accesses to sensitive data/operations of particular kinds (e.g., sinks for SMS_MMS) exhibited the greatest significance. The very few *ICC* features included in these top rankings contributed more to identifying benign apps from malware than to distinguishing malware families.

3) Learning Algorithm Choice: In addition to the Random Forest algorithm (*RF*, with 128 trees) used by default, we experimented *DroidCat* with seven other learning algorithms. Our results show that *RF* performed significantly better than all the alternatives. Support Vector Machine [68] (*SVM*) with linear kernel had the second best effectiveness, while *SVM* with rbf kernel performed the worst. Naive Bayes [69] with Bernoulli distribution had the third best effectiveness, while with Gaussian distribution it had the second worst effectiveness. Neither Decision Trees [70] nor k-Nearest Neighbors [71] worked as well as the best setting of the above three.

4) Input Coverage: We have repeated our case studies on a new dataset obtained by applying the same coverage filter used in our characterization study. Only apps for which 10-minute Monkey inputs covered at least 50% of user code were selected, resulting in 136 benign and 145 malicious apps. The user-code coverage for these 281 apps ranged from 50% to 100% (mean 66%, standard deviation 12%). The higher-coverage dataset contained 10 app categories: *BENIGN* and 9 malware families. We applied the same held-out validation as used in other experiments. For malware detection and (9-class) malware categorization, *DroidCat* gained consistent increases in each of the three performance metrics (P, R, F1) on the higher-coverage datasets compared to our results without the coverage filter. Yet, the increases were all quite small (at most 1.5%). These small differences indicate that the performance of DroidCat did not appear to be very sensitive to the user-code coverage of run-time inputs.

VIII. Efficiency

The primary source of analysis overhead of all the three techniques compared is the cost for feature extraction. For dynamic approaches like *DroidCat* and *Afonso*, this cost includes the time for tracing each app, which is five minutes in both techniques. Specifically, *DroidCat* took 353.9 seconds for feature computation and 0.01 seconds for testing, on average per app. In contrast, *Afonso* took 521.74 seconds for feature computation and 0.015 seconds for testing per app. As expected, the tracing time dominated the total feature computation cost in *DroidCat* and *Afonso*. Also, in these two techniques, the testing time is almost negligible, mainly because their feature vectors are both relatively small (at most 122 features per app in *DroidCat*, and 163 features per app in *Afonso*). As a static approach, *DroidSieve* does not incur tracing cost. Its average feature computation cost was 74.19 seconds per app. However, *DroidSieve* uses very-large feature vectors (over 20,000 features per app), causing its substantial cost for the testing phase (on average 3.52 seconds per app). Concerning the storage cost, *DroidCat* and *Afonso* took 21KB and 32KB per app, respectively, mainly for storing the traces. *DroidSieve* does not incur trace storage cost, and it took 0.4KB per app for storing feature files.

In all, *DroidCat* appeared to be reasonably efficient as a dynamic malware analysis approach, and was lighter-weight than the peer dynamic approach *Afonso*. *DroidSieve* was the most efficient among the three techniques, due to its lack of tracing overheads. However, given the substantially superior performance of *DroidCat* over *DroidSieve*, the additional cost incurred by *DroidCat* can be seen to be justified.

IX. Limitations and Threats to Validity

The difficulty and overhead of tracing a large number of apps present challenges to dynamic analysis, which constrained the scale of our studies. While reasonably large for a dynamic analysis of Android apps, our datasets may still be relatively small in size compared to those used by many static approaches. In particular, considering our datasets split by ranges of years, our samples from each period may not be representative of the app population of that period. For this reason, our results are potentially subject to overfitting. To mitigate this limitation, we have considered benchmarks from diverse sources. Recall the goal of DroidCat is to complement

static approaches in scenarios where they are inapplicable (Section II). In all, our experimental results and conclusions are best interpreted with respect to the datasets used in our studies.

Prior studies have shown that learning-based malware detectors are subject to class imbalances in training datasets [72], [73]. Our results also suffer from this subjection as our datasets contain imbalanced benign and malware samples, as well as imbalanced malware families. There were two causes for these imbalances: (1) our data sources do not provide balanced sample sets, and (2) for fair evaluation we needed to use exactly the same samples for evaluating *DroidCat* against the two baselines, thus we had to discard some samples for which the features for any technique cannot be computed (Section VI-A), which further perplexed our control of data balance. On the other hand, however, the imbalances enabled us to additionally assess the stability of our approach against the baselines: for instance, in Study I, our results revealed that the performance of DroidCat was not much affected by the imbalance of both kinds (more benign apps, in D1617 and D1415, or more malware, in D1213 and D0911). We also note that all the datasets against which we compared *DroidCat* to the baselines contained much less malware than benign samples. This kind of imbalance resembles real-world situations in which we do have much fewer malware than benign apps.

Intuitively, the more app code covered by the dynamic inputs, the more app behaviors can be captured and utilized by our approach. We thus conducted a dedicated study in this regard. Our results confirmed that with higher-coverage inputs DroidCat improved in effectiveness. However, the effectiveness differences were small (<2%) between two experiments involving datasets that had large differences (20%) in code coverage. Nevertheless, these results may not be generalizable; more conclusive results would need more extensive studies on the effect of input coverage. Also, although *DroidCat* relies on capturing app behavioral patterns in execution composition and structure (instead of modeling explicit malicious behaviors through suspicious permission access and/or data flows), reasonable coverage is still required for producing usable traces to enable the feature computation.

We aimed to leverage a *diverse* set of behavioral features (in three orthogonal dimensions) to make DroidCat robust to various evasion attacks that target specific kinds of dynamic profiles. To empirically examine the robustness, we purposely used an obfuscation benchmark suite in the evaluation. Further, in the case studies, we used datasets in which the majority of apps adopted a variety of evasion techniques, including complex/heavy reflection, data encryption, and class/method renaming. However, other types of evasion (especially anti-dynamic-analysis) attacks [74] have not been explicitly covered in our experiments. For instance, some malware might detect and then evade particular kinds of run-time environments (e.g., emulator). In our evaluation, the dynamic features were all extracted from traces gathered on an Android emulator. The high classification performance we achieved suggests that our approach seems robust against emulator-evasion attacks. On the other hand, after our features are revealed, attackers could take adversarial approaches to impede the computation of our dynamic features or pollute the code to make our features less discriminatory.

DroidCat works at app level without any modification of the Android framework and/or OS as in [18] and [76]. This design makes DroidCat easier to use and more adaptable to rapid evolution of the Android ecosystem, but it does not handle dynamically loaded code or native code yet. Meanwhile, *DroidCat* requires app instrumentation, which may constitute an impediment for its use by end users. A more common deployment setting would be to use *DroidCat* for batch screening by an app vetting service (e.g., as part of an app store), where the instrumentation, tracing, learning, and prediction can be packed in one holistic automated process of the service. Finally, our technique follows a learning-based approach using features that can be contrived, thus it may be vulnerable to sophisticated attacks such as mimicry and poisoning [76].

X. Related Work

A. Dynamic Characterization for Android Apps

There have been only a few studies broadly characterizing run-time behaviors of Android apps. Zhou *et al.* manually analyzed 1,200 samples to understand malware installation methods, activation mechanisms, and the nature of carried malicious payloads [49]. Cai *et al.* instrumented 114 benign apps for tracing method calls and ICCs, and investigated the dynamic behaviors of benign apps [81]. These studies either focus on malicious apps or benign ones. Canfora *et al.* profiled Android apps to characterize their resource usage and leveraged the profiles to detect malware [82]. We profiled method and ICC invocations in our characterization study as in [81] yet with both benign and malicious samples. Also, our study aimed at not only behavior understanding [49], [81]. We further utilized the understanding for app classification like [82] yet with different behavioral profiles and not only for malware detection (but also for categorizing malware by families).

B. Android Malware Detection

Most previous detection techniques utilized static app features based on API calls [15]–[17], [35], [77]–[79] and/or permissions [15], [18], [23], [26], [35]. ICCDetector [55] modeled ICC patterns to identify malware that exhibits different ICC characteristics from benign apps. Besides static features, a few works enhanced their capability by exploiting dynamic features (i.e., *hybrid* approaches) such as messaging traffic [26], file/network operations [18], and system/API calls [35]. However, approaches relying on static code analysis are generally vulnerable to reflection and other code obfuscation schemes [20], which are widely adopted in Android apps (especially in malware) [41]. Suarez-Tangil *et al.* [23] mined non-code (e.g., resources/assets) features for more robust detection. Static-analysis challenges have motivated dynamic approaches, of which ours is not the first. Afonso *et al.* [27] built dynamic features on system/API call frequencies for malware detection, similar to [29] where occurrences of

TABLE VI

COMPARISON OF RECENT WORKS ON ANDROID MALWARE CLASSIFICATION IN CAPABILITY AND ROBUSTNESS. DET: DETECTION, CAT: FAMILY CATEGORIZATION, SYSC: SYSTEM CALL, RT_PERM: RUN-TIME PERMISSION, RES: RESOURCE, OBF: OBFUSCATION

Technique	Year	Approach	Classification Capability		Robustness against Analysis Challenges			
			DET	CAT	Reflection	SYSC_OBF	RT_PERM	RES_OBF
DroidMiner [78]	2014	Static	✓	✓	✗	✓	✓	✓
DroidSIFT [79]	2014	Static	✓	✓	✗	✓	✗	✓
Drebin [15]	2014	Static	✓	N/A	✗	✓	✗	✗
MudFlow [80]	2015	Static	✓	N/A	✗	✓	✓	✓
Afonso et al. [27]	2015	Dynamic	✓	N/A	unknown	✗	✓	✓
Marvin [18]	2015	Hybrid	✓	N/A	✗	✓	✗	✗
Madam [35]	2016	Hybrid	✓	N/A	unknown	✗	✗	✓
ICCDetector [55]	2016	Static	✓	N/A	✗	✓	✓	✗
DroidScribe [34]	2016	Dynamic	N/A	✓	✓	✗	✓	✓
StormDroid [26]	2016	Hybrid	✓	N/A	✗	✓	✗	✓
MamaDroid [81]	2017	Static	✓	N/A	✗	✓	✓	✓
DroidSieve [23]	2017	Static	✓	✓	✓	✓	✗	✗
DroidCat	this work	Dynamic	✓	✓	✓	✓	✓	✓

unique callsites were used as features. A recent static technique MamaDroid [80] and its dynamic variant [83] model app behaviors based on the transition probabilities between abstracted API calls in the form of Markov chains.

C. Android Malware Categorization

Approaches have been proposed to categorize malware into known families. Xu *et al.* traced system calls, investigated three alternative ways to graphically represent the traces, and then leveraged the graphs to categorize malware [39]. Dash *et al.* generated features at different levels, including pure system calls and higher-level behavioral patterns like file system access which conflate sequences of related system calls [34]. Some of the malware detection techniques have been applied to family categorization as well [23], [77], [78].

D. Discussion

Table VI compares our approach to representative recent peer works in terms of classification capability with respect to the three possible settings and robustness against various analysis challenges. For the settings that a tool was not designed to work in, the capability was not applicable (hence noted as *N/A*).

Almost all the static approaches compared are vulnerable to reflection as they use features based on APIs. Marvin [18] as a hybrid technique also suffers from this vulnerability as it relies on a number of static API-based features. Techniques using features on static permissions, such as DroidSIFT [78], Drebin [15], and StormDroid [26], face challenges due to run-time permissions [19], [84] which are increasingly adopted by (over one third already of) Android apps [85]. The use of features based on system calls comprises the resiliency of DroidScribe [34], Madam [35], and Afonso [27] against obfuscation schemes targeting system calls [30], [36], [38]. DroidSieve [23] gains high accuracy with resilience against reflection by reducing code analysis and using resource-centered features, but may not detect malware that expresses malicious behaviors only in code while with benign resources/assets. Our comparative study results presented in this paper have supported this hypothesis. In addition, like a few other works that extract features (other than permission) from resource files [15], [18], [55], it may not work with malware with resources obfuscated [24], [25], [41].

In contrast, DroidCat adopts a purely dynamic approach that resolves reflective calls at runtime, thus it is fully resilient against even complex cases of reflection. It relies on no features from resource files or based on system calls, thus it is robust against obfuscation targeting those features. While it remains to be studied if it well adapts to Android ecosystem evolution, DroidCat would not be much affected by run-time permissions as it does not use related features. Also, compared to prior approaches typically focusing on API calls, *DroidCat* characterizes the invocations of all methods and ICCs.

We omitted in the table the effectiveness numbers (e.g., detection rate and accuracy) for these compared works because they are not comparable: the numbers all came from varied evaluation datasets. In this paper, we have extensively studied two of the listed approached versus ours on the same datasets. Nonetheless, in terms of any of the effectiveness metrics we considered, DroidCat appeared to have very promising performance relative to the state-of-the-art peer approaches.

XI. CONCLUSION

We presented DroidCat, a dynamic app classification technique that detects and categorizes Android malware with high accuracy. Features that capture the *structure* of app executions are at the core of our approach, in addition to those based on *ICC* and *security* sensitive accesses. We empirically showed that this *diverse*, novel set of dynamic features enabled the superior *robustness* of DroidCat against analysis challenges such as heavy and complex use of reflection, resource obfuscation, system-call obfuscation, use of run-time permissions, and other evasion schemes. These challenges impede most existing peer approaches, a real concern since the app traits leading to the challenges are increasingly prevalent in modern Android ecosystem.

Through extensive evaluation and in-depth case studies, we have shown the superior stability of our approach in achieving high classification performance, compared to two state-of-the-art peer approaches, one static and one dynamic. Meanwhile, in absolute terms, DroidCat achieved significantly higher accuracy than the peer approaches studied for both

malware detection and family categorization. Thus, DroidCat constitutes a promising solution complementary to existing alternatives.

REFERENCES

[1] (2015). *Android Malware Accounts for 97% of Malicious Mobile Apps*. [Online]. Available: http://www.scmagazineuk.com/updated-97-of-malicious-mobile-malware-targets-android/article/422783/

[2] *The Ultimate Android Malware Guide: What It Does, Where It Came From, and How to Protect Your Phone or Tablet*. Accessed: Apr. 5, 2018. [Online]. Available: http://www.digitaltrends.com/android/the-ultimate-android-malware-guide-what-it-does-where-it-came-from-and-how-to-protect-your-phone-or-tablet/

[3] K. Lu *et al.*, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. NDSS*, 2015, pp. 1–15.

[4] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. ICSE*, 2015, pp. 303–313.

[5] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. FSE*, 2014, pp. 576–587.

[6] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, 2013, pp. 45–54.

[7] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. MobiSys*, 2012, pp. 281–294.

[8] H. Cai and J. Jenkins, "Leveraging historical versions of Android apps for efficient and precise taint analysis," in *Proc. MSR*, 2018, pp. 265–269.

[9] B. Wolfe, K. Elish, and D. Yao, "High precision screening for Android malware with dimensionality reduction," in *Proc. 13th Int. Conf. Mach. Learn. Appl.*, Dec. 2014, pp. 21–28.

[10] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh, "Automatic generation of string signatures for malware detection," in *Proc. RAID*, 2009, pp. 101–120.

[11] H. Kang, J.-W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying Android malware using static analysis along with creator information," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 6, p. 479174, 2015.

[12] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. CCS*, 2012, pp. 241–252.

[13] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proc. 17th ACM Symp. Access Control Models Technol.*, 2012, pp. 13–22.

[14] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. CCS*, 2009, pp. 235–245.

[15] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 23–26.

[16] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. SecureComm*, 2013, pp. 86–103.

[17] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.

[18] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. COMPSAC*, vol. 2, Jul. 2015, pp. 422–433.

[19] (2015). *Requesting Permission at Run Time*. [Online]. Available: https://developer.android.com/training/permissions/requesting.html

[20] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. ACSAC*, Dec. 2007, pp. 421–430.

[21] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. IEEE Symp. Secur. Privacy*, May 2005, pp. 32–46.

[22] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in *Proc. SAC*, 2010, pp. 1970–1977.

[23] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: Fast and accurate classification of obfuscated Android malware," in *Proc. CODASPY*, 2017, pp. 309–320.

[24] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on Android malware," *Comput. Secur.*, vol. 51, pp. 16–31, Jun. 2015.

[25] G. Square. (2017). *Dexguard*. [Online]. Available: https://www.guardsquare.com/en/dexguard

[26] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "StormDroid: A streaminglized machine learning-based system for detecting Android malware," in *Proc. ASIA CCS*, 2016, pp. 377–388.

[27] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *J. Comput. Virology Hacking Techn.*, vol. 11, no. 1, pp. 9–17, 2015.

[28] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012.

[29] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.

[30] J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao, "Impeding behavior-based malware analysis via replacement attacks to malware specifications," *J. Comput. Virology Hacking Techn.*, vol. 13, no. 3, pp. 193–207, 2017.

[31] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. NDSS*, 2009, pp. 8–11.

[32] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of Android malware behaviors," in *Proc. NDSS*, 2015, pp. 1–15.

[33] H. S. Galal, Y. B. Mahdy, and M. A. Atiea, "Behavior-based features model for malware detection," *J. Comput. Virol. Hacking Techn.*, vol. 12, no. 2, pp. 59–67, 2016.

[34] S. K. Dash *et al.*, "DroidScribe: Classifying Android malware based on runtime behavior," in *Proc. IEEE Secur. Privacy Workshops*, May 2016, pp. 252–261.

[35] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018.

[36] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: Automatically evading system-call-behavior based malware detection," *J. Comput. Virol.*, vol. 8, nos. 1–2, pp. 1–13, 2012.

[37] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *Proc. ACSAC*, Dec. 2008, pp. 418–430.

[38] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti, "Operating system interface obfuscation and the revealing of hidden operations," in *Proc. DIMVA*, 2011, pp. 214–233.

[39] L. Xu, D. Zhang, M. A. Alvarez, J. A. Morales, X. Ma, and J. Cavazos, "Dynamic Android malware classification using graph-based representations," in *Proc. IEEE 3rd Int. Conf. Cyber Secur. Cloud Comput. (CSCloud)*, Jun. 2016, pp. 220–231.

[40] T. K. Ho, "Random decision forests," in *Proc. 3rd Int. Conf. Document Anal. Recognit.*, 1995, p. 278.

[41] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, 2017, Art. no. 76.

[42] (2012). *Over 60 Percent of Android Malware Comes from One Malware Family: Fakeinstaller*. [Online]. Available: http://tech.firstpost.com/news-analysis/over-60-percent-of-android-malware-comes-from-one-malware-family-mcafee-48109.html

[43] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. ICSE*, May 2015, pp. 77–88.

[44] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proc. CCS*, 2016, pp. 343–355.

[45] *Android App Components*. Accessed: Apr. 5, 2018. [Online]. Available: http://www.tutorialspoint.com/android/android_application_components.htm

[46] Google. (2015). *Android Monkey*. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[47] H. Cai and B. G. Ryder, "DroidFax: A toolkit for systematic characterization of Android applications," in *Proc. ICSME*, Sep. 2017, pp. 643–647.

[48] *VirusTotal*. [Online]. Available: https://www.virustotal.com/

[49] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.

[50] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot—A Java bytecode optimization framework," in *Proc. Cetus Users Compiler Infrastruct. Workshop*, 2011, pp. 1–11.
[51] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. ECOOP*, 1995, pp. 77–101.
[52] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proc. ASE*, 2014, pp. 343–348.
[53] Google. (2015). *Android Emulator*. [Online]. Available: http://developer.android.com/tools/help/emulator.html
[54] D. Octeau *et al.*, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. USENIX Secur. Symp.*, 2013, pp. 543–558.
[55] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based malware detection on Android," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 6, pp. 1252–1264, Jun. 2016.
[56] J. Jenkins and H. Cai, "Dissecting Android inter-component communications via interactive visual explorations," in *Proc. ICSME*, Sep. 2017, pp. 519–523.
[57] J. Jenkins and H. Cai, "ICC-inspect: Supporting runtime inspection of Android inter-component communications," in *Proc. MobileSoft*, 2018, pp. 80–83.
[58] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proc. Conf. Emerg. Artif. Intell. Appl. Comput. Eng.*, 2007, pp. 3–24.
[59] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[60] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. MSR*, May 2016, pp. 468–471.
[61] (2018). *Google Play Store*. [Online]. Available: https://play.google.com/store
[62] (2018). *VirusShare*. [Online]. Available: https://virusshare.com/
[63] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 326–327, Sep. 1995.
[64] R. B. Rao, G. Fung, and R. Rosales, "On the dangers of cross-validation. An experimental evaluation," in *Proc. SIAM Int. Conf. Data Mining*, 2008, pp. 588–596.
[65] *Contagio Dataset*. Accessed: Apr. 5, 2018. [Online]. Available: https://contagiodump.blogspot.com
[66] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Unified dynamic detection of Android malware," Comput. Sci. Dept., Virginia Tech, Blacksburg, VA, USA, Tech. Rep. TR-17-01, Jan. 2017. [Online]. Available: http://hdl.handle.net/10919/77523
[67] D. Cournapeau. (2016). *Machine Learning in Python*. [Online]. Available: http://scikit-learn.org/stable/supervised_learning.html#supervised-learning
[68] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.
[69] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Malaysia, Kuala Lumpur: Pearson, 2016.
[70] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.
[71] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *Amer. Statist.*, vol. 46, no. 3, pp. 175–185, 1992.
[72] S. Roy *et al.*, "Experimental study with real-world data for Android app security analysis using machine learning," in *Proc. ACSAC*, 2015, pp. 81–90.
[73] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, "Empirical assessment of machine learning-based malware detectors for Android," *Empirical Softw. Eng.*, vol. 21, no. 1, pp. 183–211, 2016.
[74] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of Android execution environments," in *Proc. ICSE*, May 2017, pp. 300–311.
[75] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "ANDRUBIS–1,000,000 apps later: A view on current Android malware behaviors," in *Proc. 3rd Int. Workshop Building Anal. Datasets Gathering Exper. Returns Secur.*, Sep. 2014, pp. 3–17.
[76] S. Venkataraman, A. Blum, and D. Song, "Limits of learning-based signature generation with adversaries," in *Proc. NDSS*, 2008, pp. 1–16.
[77] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2014, pp. 163–182.
[78] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual api dependency graphs," in *Proc. CCS*, 2014, pp. 1105–1116.
[79] V. Avdiienko *et al.*, "Mining apps for abnormal usage of sensitive data," in *Proc. ICSE*, May 2015, pp. 426–436.
[80] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting Android malware by building Markov chains of behavioral models," in *Proc. NDSS*, 2017, pp. 1–12.
[81] H. Cai and B. G. Ryder, "Understanding Android application programming and security: A dynamic study," in *Proc. ICSME*, Sep. 2017, pp. 364–375.
[82] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Acquiring and analyzing app metrics for effective mobile malware detection," in *Proc. ACM Int. Workshop Secur. Privacy Anal.*, 2016, pp. 50–57.
[83] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro. (2018). "A family of droids—Android malware detection via behavioral modeling: Static vs dynamic analysis." [Online]. Available: https://arxiv.org/abs/1803.03448
[84] M. Dilhara, H. Cai, and J. Jenkins, "Automated detection and repair of incompatible uses of runtime permissions in Android apps," in *Proc. MobileSoft*, 2018, pp. 67–71.
[85] Google. *Android Developer Dashboard*. Accessed: Sep. 20, 2016. [Online]. Available: http://developer.android.com/about/dashboards/index.html

Authors' photographs and biographies not available at the time of publication.