DroidCat: Unified Dynamic Detection of Android Malware

ABSTRACT

Various dynamic approaches have been developed to detect or categorize Android malware. These approaches execute software, collect call traces, and then detect abnormal system calls or sensitive API usage. Consequently, attackers can evade these approaches by intentionally obfuscating those calls under focus. Additionally, existing approaches treat detection and categorization of malware as separate tasks, although intuitively both tasks are relevant and could be performed simultaneously. This paper presents DroidCat, the first *unified* dynamic malware detection approach, which not only detects malware, but also pinpoints the malware family. DroidCat leverages supervised machine learning to train a multi-class classifier using *diverse* behavioral profiles of benign apps and different kinds of malware. Compared with prior heuristics-based machine learning-based approaches, the feature set used in DroidCat is decided purely based on a *systematic dynamic characterization study* of benign and malicious apps. *All* differentiating features that show behavioral differences between benign and malicious apps are included. In this way, DroidCat is robust to existing evasion attacks.

We evaluated DroidCat using leave-one-out cross validation with 136 benign apps and 135 malicious apps. The evaluation shows that DroidCat provided an effective and scalable unified malware detection solution with 81% precision, 82% recall, and 92% accuracy.

1. INTRODUCTION

Android is the target of 97% malicious mobile apps [10], most of which steal personal information, abuse privileged resources, and/or install additional malicious software [7]. Detecting and categorizing malware are crucially important for Android developers and users.

Researchers have mainly taken two types of approaches to Android malware detection: static and dynamic. The static approaches analyze code to check whether an app contains abnormal information flows or calling structures [26, 29, 32, 39, 60], matches malicious code patterns [33, 55], requests for more permissions than necessary [25, 35, 44, 48], and/or invokes APIs that are frequently used by malware [11, 14, 56]. These approaches cannot always precisely detect malware because how control and data dynamically flows is not always statically decidable. The mere existence of some permissions and/or APIs in code does not always mean that they are used or executed at runtime. Furthermore, attackers can use widely adopted techniques such as code obfuscation [21] and metamorphism [38] to evade static pattern matching and deceive the detectors.

In comparison, dynamic approaches provide a complementary way to detect malware [12, 16, 19, 49]. They profile program *behaviors* [15, 52] by collecting system calls [16, 28, 36, 46], permissions [19], and/or resource usage [28, 49]. Based on the profiles, existing dynamic detectors typically use machine learning to train a classifier to decide whether an app is malware. However, these detectors are limited by the specific profiles on which they focus. For example, the system call-based detectors suffer from the evasion attacks, which intentionally obfuscate malicious activities by replacing malicious chains of system calls with some syntactically different, but semantically equivalent system calls [27, 40, 50]. A more advanced dynamic malware detector is needed to capture *varied* behavioral profiles and thus become *robust* to attacks against specific profiles.

Given a malicious app, existing Android malware categorization approaches also rely on system calls [23, 58] to identify the malware family. Similar to many dynamic detectors, these approaches are also subject to the obfuscation attack of system calls [41]. Even worse, current categorization approaches can only be applied to the malicious apps recognized beforehand by humans or other tools. To decide whether a given app belongs to a specific malware family, developers usually take two passes: one pass to differentiate malicious apps from benign ones, and one pass to categorize malicious apps. This two-pass process may not be as efficient as a single pass for both malware detection and classification, thus costing more in time and human effort.

In this paper, we present the design and implementation of a *unified* malware detection technique, *DroidCat*, which detects and categorizes Android malware simultaneously through systematic dynamic profiling and supervised learning. Differently from existing machine-learning based dynamic approaches, DroidCat trains a classification model with a more *diverse* set of behavioral features. It instruments all Inter-Component Communications (ICCs)[1], and all invocations of methods defined by user code, third-party libraries, and the Android framework.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1]ICC is an event-driven model of communication, as further explained in Section 2.

Therefore, DroidCat is robust to attacks targeting system calls, because system calls are not used in our approach. It is also robust to attacks targeting sensitive APIs, because these APIs are not the only information source of method invocations.

The features used in DroidCat manifest the behavioral differences between benign and malicious apps. Compared with prior approaches, this feature set was decided based on a *systematic dynamic characterization study* of 136 benign apps and 135 malicious apps. In the study, we traced the execution of each app, defined and evaluated 122 behavioral metrics to thoroughly characterize any behavioral difference between the two app groups. All these metrics measure the *occurrence frequencies* of certain method invocations or ICCs, which can never be captured by static malware detectors.

Based on the study, we discovered 70 discriminating metrics with noticeably different values, and included all of them into our feature set. By training a model with the Random Forest machine learning algorithm [34], DroidCat builds a multi-class classifier that predicts whether an app is benign or malicious in a particular malware family.

We evaluated DroidCat using leave-one-out cross validation, the strongest form of k-fold cross validation, with our data set of benign and malicious apps. For the unified detection, DroidCat achieved 81% precision, 82% recall, and 92% accuracy. For conventional binary-classification malware detection (only telling benign or not), DroidCat performed even better with 95% precision, 99% recall, and 99% accuracy. As malicious apps behave similarly, *precisely identifying* the malware family is generally more challenging than merely detecting malware. This explains why DroidCat's unified malware detection is not as effective as conventional malware detection.

To understand how sensitive DroidCat is to the selection of behavioral features, we investigated 5 other ways to choose features: 1 full set of 122 metrics and 4 different subsets. Our investigation revealed that DroidCat worked best with the 70 metrics that were noticeably different between benign and malicious apps. To avoid any bias caused by the selection of machine learning algorithms, we also experimented with four other machine learning algorithms: Support Vector Machine [22], Decision Trees [45], k-Nearest Neighbors [13], and Naive Bayes [9]. We found that DroidCat performed best when using Random Forests.

In summary, we have made the following contributions:

- We designed and implemented the *first unified* dynamic Android malware detection approach, *DroidCat*, based on a systematic characterization study of benign and malicious apps. Due to its diverse feature set, DroidCat is robust to evasion attacks targeting specific behavioral profiles. It not only precisely detects malware, but also pinpoints the malware family for most malicious apps.
- We conducted the *first systematic dynamic* characterization study for Android applications with 136 benign apps and 135 malicious apps. To thoroughly characterize program behaviors, we traced ICCs and all methods defined in user code, third-party libraries, and Android SDK. We defined 122 behavioral metrics. No prior work has done such a comprehensive dynamic characterization for Android apps.
- We explored different learning algorithms to train models for unified malware detection, and observed that Random Forest worked best.
- We open sourced both DroidCat and the benchmark suite of 271 Android apps. Our benchmarks will facilitate scientific comparison between dynamic malware detection and categorization techniques.

2. BACKGROUND

To facilitate later discussion, this section introduces main concepts and terminologies relevant to our work.

Android applications. Programmers develop Android apps primarily using Java, and then build them into Android application package (i.e., APK) files. Each APK file can contain three software layers: **user code**, **Android libraries** (i.e., SDK APIs), and **third-party libraries** (if any). An Android application typically comprises four components as follows [1]:

- **Activities**: They dictate the UI and handle the user interaction to the device screen.
- **Services**: They handle background processing associated with an application.
- **Broadcast Receivers**: They handle communication between Android OS and applications.
- **Content Providers**: They handle data storage and management (e.g., database) issues.

ICC. Components interact with each other through ICC objects—mainly *Intents*. We classify ICC in two ways depending on the relationship between Intent senders and receivers, and the content of 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**.

Lifecycle methods and callback. 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*, such as `onCreate()`, `onStart()`, and `onDestroy()`, to define program behaviors when the events happen. Developers can also overwrite other event handlers (such as `onClick()`) or define new *callbacks* to implement extra logic when other interesting events occur.

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 as **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 as **sinks** of potential sensitive information flows. If an app's execution trace has any paths from sources to sinks, the app is considered malicious because of potential sensitive data leakage.

3. UNIFIED MALWARE DETECTION

We designed and implemented DroidCat, a unified malware detection approach leveraging systematic dynamic profiling and supervised learning, to decide whether a given app is benign or belongs to a particular malware family. As shown in Figure 1, there are two phases in our approach: training and testing. In the training phase, DroidCat takes in both benign and malicious apps as input. For each app, it computes behavioral features by instrumenting and executing the program, and by

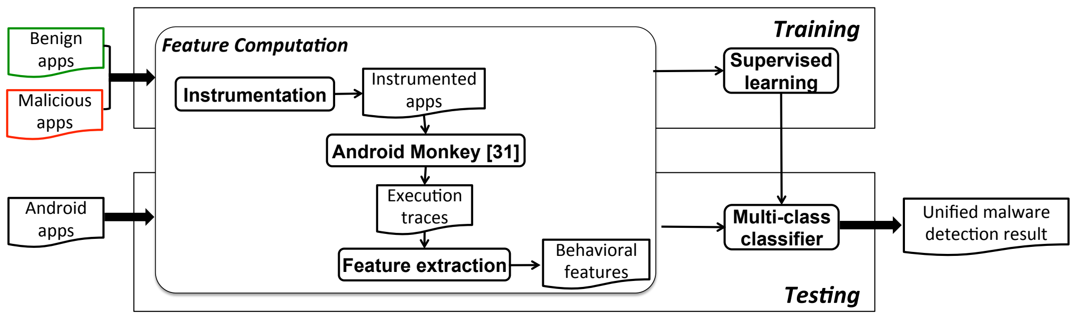

Figure 1: *DroidCat* consists of two phases: training and testing. The first phase takes in both benign and malicious apps to train a multi-class classifier. The second phase takes in Android apps to do unified malware detection with the trained classifier.

collecting a variety of behavioral characteristics. The features are then provided to the supervised machine learning to train a multi-class classifier, which will be used in the second phase. For testing, given an arbitrary app, DroidCat computes its behavioral features in the same way as mentioned above, and then feeds these features to the classifier to decide whether the app is benign or a member of a malware family. In this section, we will discuss how features are computed in Section 3.1, and explain the two phases in Section 3.2 and 3.3, separately.

3.1 Feature Computation

To compute the dynamic features of an Android app, we first instrumented the program for execution trace collection. Specifically, for each app's APK file, we used Soot [37] to decompile the executable file into bytecode, and then inserted bytecode instrumentation to trace every method call, and every ICC together with its *Intent* content. In this process, we also labeled additional information for instrumented classes and methods to facilitate feature extraction. 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 Android API. To decide the component type of a class such as `Foo`, we applied Class Hierarchy Analysis (CHA) [24] 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 a method-type mapping list used in [17] to label the category of callbacks and the source/sink property of APIs.

Next, we ran the instrumented APK of each app on an Android emulator [30] to collect execution traces, which include all method calls and ICCs. Note that we do not instrument OS-level system calls, because we want DroidCat to be robust to any attack 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 robust to any attack targeting sensitive APIs as well. Prior work shows that even without invoking system APIs or sensitive APIs, some malicious apps can still conduct attacks by manipulating other apps via ICCs [42]. Therefore, we also instrument ICCs to reveal any behavioral differences between benign and various malicious apps from a different perspective.

To fully 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 paths as possible. Manually entering inputs to apps is tedious and inefficient. In order to quickly trigger diverse executions of an app, we used Monkey [31] to randomly generate inputs. Since a previous study shows that for an average app, Monkey does not significantly cover more paths after the first 10-minute run [20], we intentionally set Monkey to execute every app for 10 minutes. This allowed us to efficiently collect abundant trace data without sacrificing much dynamic coverage.

Finally, we extracted 70 features from the execution traces. All of these features were defined as the percentage of certain function calls or ICCs with a particular characteristic. For instance, one feature measures the percentage of ICCs carrying URI data only, while another feature describes the percentage of source APIs which have paths reaching at least one sink API. These features were defined based on our systematic dynamic characterization study of Android apps. Both the study and our extracted features will be further discussed in Section 4.

3.2 Training

To train a classifier for unified malware detection, we need behavioral feature data not only for benign apps, but also for malicious apps from different malware families. Therefore, in our training data, each data point represents one app, and has the following format: $<feature\_vector, label>$, where $feature\_vector$ contains 70 feature values, which are computed as mentioned in Section 3.1. *Label* is either *BENIGN* or a malware family's name (e.g., *DroidKungfu*). We use Random Forest [34], a supervised machine learning (ML) algorithm, to train a classifier with the labeled data.

3.3 Testing

In the testing phase, given an unknown app, DroidCat computes its features as mentioned in Section 3.1, and passes the feature vector to the well-trained multi-class classifier. Unlike the training data, each data point in the testing data is unlabeled, because we do not know whether an app is benign. Based on the feature vector, our classifier judges what category label should be assigned to the app.

We implemented DroidCat in Python, and used Scikit-learn [43], a free machine learning toolkit for Python, to train and test the classifier. We provided open source DroidCat at (link withheld for double-blind review.)

4. FEATURE EXTRACTION

With the execution traces collected for both benign and malicious apps as mentioned in Section 3.1, we need to extract a variety of behavioral features to characterize different apps, and to leverage machine learning for

Table 1: Metrics for dynamic characterization

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

effective unified malware detection. Although for the same trace we can extract as many features as we like, not every feature is a good differentiator of malicious apps from benign ones. Therefore, with 136 benign apps and 135 malicious apps (Section 4.1), we conducted a systematic dynamic characterization study by defining and measuring 122 metrics (Section 4.2). 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 4.3).

4.1 Benchmarks

Our characterization study needed 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 to check 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 [31], and (3) the 10-minute Monkey run covers at least 50% of user code. If an app met all criteria, we further analyzed it with VirusTotal[2], to check whether the app is benign. If it is, we included it into our data set, which eventually comprised 136 benign apps.

We collected malicious apps with their malware families identified similarly. Instead of starting with the popular free apps in Google Play, we used the 1,433 malicious apps mentioned in prior work [61] as our initial candidate pool. We found 135 apps meeting the above criteria, and analyzed them with VirusTotal for malware confirmation and malware type identification. For instance, we checked a malicious app `rom.jonas.eley` with VirusTotal, which reported that the app is malicious and belongs to the malware family FakeInst. VirusTotal confirmed all 135 malicious apps. Based on its reports, we identified six malware families, as shown in Table 2. The first five malware families were directly reported by VirusTotal; their attack models are explained below:

Table 2: Categories of the 135 malicious apps

Malware Family	# of Apps	Exemplar App
DroidKungfu	5	`com.peter.wuzilianzhu`
ProxyTrojan/NN	27	`android.cat.calendar`
GoldDream	11	`com.craigsrace.headtoheadrcing`
Plankton	8	`cenix.android.vbr`
FakeInst	33	`com.opera.installer`
MALICIOUS	51	`app.batterymonitor`

DroidKungfu can obtain the root access to Android operating system code. It encrypts two known root exploits: a udev exploit and a so-called rageagainstthecage exploit. During execution, it decrypts these two exploits and executes them to launch the attack [6].
ProxyTrojan/NN (short for ProxyTrojan/NotCompatible/NioServ) is a type of Trojan horse designed to use the victim's computer as a proxy server. This gives the attacker the opportunity to do everything from the victim's computer, such as committing credit card frauds and launching attacks against other computers [4].
GoldDream spys on SMS messages received by users and incoming/outgoing phone calls. It then uploads them to a remote server without users' awareness. Moreover, this malware can also fetch commands from a remote server and execute them accordingly [5].
Plankton is installed on phones as a bundle with an application that a user downloaded. It displays unwanted advertisements as notifications. These ads are hard to block unless users completely remove some applications [2].
FakeInst appears to be an installer for other applications. When executed, it sends SMS messages to premium-rate numbers or services, although the content it presents is actually free of charge. Furthermore, it can also intercept, delete, and respond to incoming text messages [3].

In addition to these five families, we synthesized the last family *MALICIOUS* to contain those apps not included in one of the five families. In order to train a multi-class classifier, we need each category to contain a sufficient number of apps. Although VirusTotal identified the malware families for each of these 51 remaining apps, none of those families contained more than 4 apps, and 11 of them contained only 1 app. Therefore, we decided to merge the minor families whose member app counts were less than five. The APK file size of all our benchmark apps varies from 2.9MB to 25.6MB. All benchmarks are available at (link withheld for double-blind review.)

4.2 Dynamic Characterization

Based on collected execution traces, we characterized program behaviors by defining 122 metrics in 3 orthogonal dimensions: structure, ICC, and security (Table 1). Intuitively, the more *diversely* these metrics capture one execution trace, the *more completely* they characterize program behaviors. These metrics measure not only the existence of certain method invocations or ICCs, but also their occurrence frequencies. To simplify explanation, we will only discuss a few metrics in the paper. For a full explanation of all metrics, please refer to our website (link withheld for double-blind review.)

Structure dimension contains 63 metrics to describe the distributions of method calls, their declaring classes, and caller-callee links. Among these metrics, 31 metrics describe the distributions of all method calls among three software 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*.

ICC dimension contains 7 metrics to describe ICC distributions. Since there are two ways to classify ICCs,

[2]VirusTotal is a free online service that analyzes files and URLs to identify viruses, worms, trojans, and other kinds of malicious content. Given a file to analyze, VirusTotal runs multiple antivirus engines and website scanners in parallel, and then aggregates the output from these tools.

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 *Intents*.

Security dimension contains 52 metrics to describe distributions of sources, sinks, and the reachability between them. Specifically, to examine the reachability between any two APIs when given an execution trace, we create a dynamic call graph, and then check for any path in the call graph that links the APIs. If a source has at least one path reaching a sink, it is considered **risky source**. Similarly, a **risky sink** is reachable from at least one source. We defined risky sources and sinks as separate metrics. Both of them indicate security vulnerabilities, because sensitive data may be leaked when flowing from sources to sinks.

4.3 Feature Selection

With 122 metrics defined, we evaluated the metrics for each app in our benchmark suite. If some metrics always showed different profiles between benign apps and malicious apps, we relied on them to characterize the behavioral divergence, and selected them as features to train a unified malware detector.

To identify any metric with different profiles across the two app groups, we measured the value of each metric on every benchmark app, and then computed the mean value for all benign apps and that for all malware. 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*** concerning the metric; if a metric had a difference greater than or equal to 2%, we said the behavioral profile was ***noticeably different*** concerning the metric. As expected, substantially disparate metrics are always a subset of noticeably different metrics. In our setting, 5% and 2% were chosen as heuristics; our later experiments showed that both thresholds work well.

As shown in Table 1, by comparing mean metric values across app groups, we found 36 substantially disparate metrics, and 70 noticeably different metrics. Due to the space limit, we only show the top 10 differentiating metrics in Figure 2. In the figure, 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 two horizontal bars: one red bar to show the mean value of all malicious apps, and one green bar to represent the mean of all benign ones. The whisker on each bar represents the standard error of the mean[3]. These 10 metrics best demonstrate the behavioral differences between malicious and benign apps. We further examine these metrics, and group them by dimensions.

In the *structural* dimension, malicious apps call fewer methods defined in SDK, more methods defined in user code, and involve more callbacks relevant to the UI. This indicates that user operations may trigger excessive or unplanned 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 attack specific external components, or sends more URI data via ICCs to disseminate potentially malicious URIs. 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.

Our metric comparison shows that malicious apps behave noticeably differently from benign ones. Such behavioral differences are reflected in the metrics of the structural, ICC, and security dimensions. By default, we use the 70 noticeably different metrics as the features extracted to train unified malware detection models in DroidCat. Recall that these features show not only the existence of certain method calls or ICCs, but also their dynamic execution rates, which can never be captured by any static malware detector.

[3]https://en.wikipedia.org/wiki/Standard_error

5. EVALUATION

In this section, we will first discuss our evaluation methodology in Section 5.1, and then present the evaluation results for DroidCat in Section 5.2 and 5.3. To better understand how sensitive DroidCat is to feature selection and ML algorithm choice, we also investigated different feature sets and ML algorithms, and will report our observations in Section 5.4 and 5.5. Finally, we will describe the analysis cost of our approach in Section 5.6.

5.1 Evaluation Methodology

K-fold cross validation (CV) is a widely adopted method to assess how well a supervised classification generally performs on independent data sets [47]. For our evaluation, we use leave-one-out cross validation (LOOCV), the strongest CV form, to assess DroidCat's unified malware detection capability. In more detail, given n samples from C categories, each time LOOCV uses one sample as testing data, and considers the other $(n-1)$ samples as training data to train and test a multi-class classifier. The experiment is repeated n times until every sample is used exactly once as testing data.

In our settings, $n = 271$ including 136 benign apps and 135 malicious apps, and $C = 7$ including 1 benign category, 5 identified malware families, and 1 synthesized category *MALICIOUS* that contains apps from minor malware families. We chose to use LOOCV because it is recommended as the most efficient way to use limited available data for cross validation [8].

By evaluating DroidCat's prediction capability for each category, we can average the effectiveness metrics among all categories to measure its general performance. Specifically, for each category C_i, we assessed DroidCat's effectiveness with the following four metrics:

Precision (P) measures among all the apps labeled as "C_i" by DroidCat, how many of them actually belong to the category. Formally,

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

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

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

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

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

Accuracy (A) [58] measures among all apps no matter whether or not they belong to C_i, how many of them are labeled correctly as "C_i" or "not C_i". Formally,

$$A_i = \frac{(\text{\#apps correctly labeled as "} C_i'' \text{or "not } C_i''\text{)}}{\text{Total \# of apps}}. \quad (4)$$

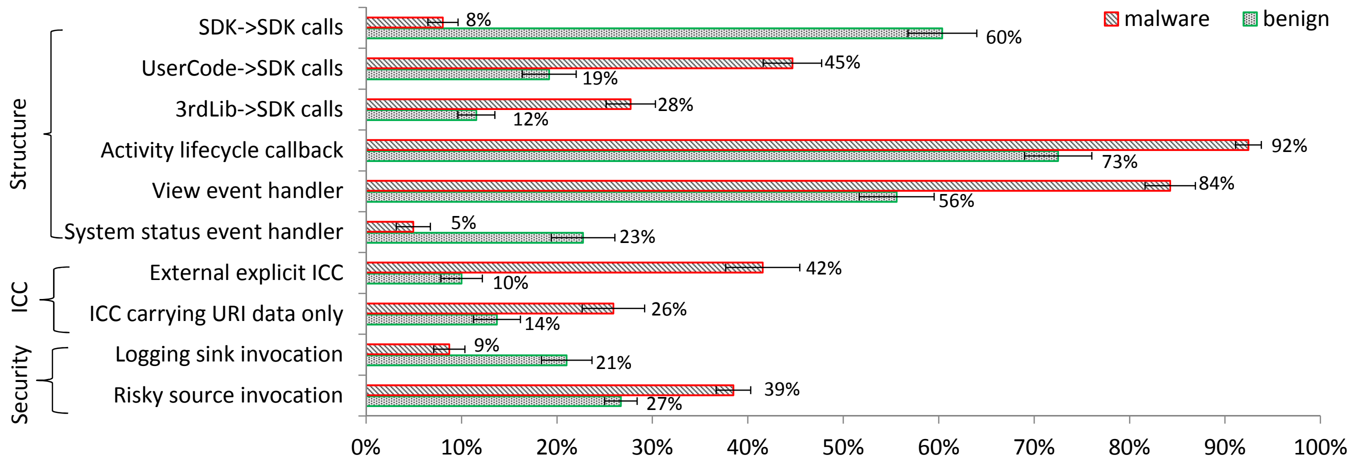

Figure 2: Top-10 differentiating metrics between malware and benign apps.

Note that our multi-class classifier only labels apps with "C_1''", "C_2", ..., or "C_7", and never uses any label like "not C_i". However, to facilitate the accuracy computation with respect to a particular category like C_1, we treat all apps with other labels like "C_2", "C_3", ..., "C_7" as "not C_i".

Suppose among the 271 apps, there are 10 apps belonging to C_3. DroidCat 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. $F1_3 = 2 * 73\% * 80\%/(73\% + 80\%) = 76\%$. Finally, $A_3 = (8 + 258)/271 = 98\%$, because 8 C_3 apps and 258 non-C_3 apps are predicted correctly.

With the above effectiveness metrics computed for each category, we further evaluated the overall effectiveness of DroidCat by computing the weighted average among categories. Intuitively, the larger the number of apps in a category, 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 benchmark suite. Formally, if we use Γ to generally represent P, R, and A, and use n_i to represent the number of samples in C_i, then the **overall effectiveness** in terms of precision, recall, and accuracy can be computed with

$$\Gamma_{overall} = \frac{\sum_{i=1}^{7} \Gamma_i * n_i}{\sum_{i=1}^{7} n_i}. \tag{5}$$

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

$$F1_{overall} = \frac{2 * P_{overall} * R_{overall}}{P_{overall} + R_{overall}} \tag{6}$$

5.2 Unified Malware Detection Results

Table 3 presents our evaluation results, in terms of confusion matrix and the four effectiveness metrics. We also examined the average behavioral profile of each app category to further understand DroidCat's effectiveness.

Confusion Matrix Results.

A confusion matrix (or error matrix) [51] is a specific table layout used to visualize the performance of a supervised learning algorithm. In Table 3, each row shows the number of instances in an actual category, while each column represents the instances in a predicted category. Numbers on the diagonal represent the instances whose categories are predicted correctly. Ideally, if a classifier has all positive numbers on the diagonal and "0" in all other cells, the classifier perfectly labels all instances.

In our confusion matrix, there are seven rows and seven columns, corresponding to the seven categories from C_1 to C_7. The number of apps in each category varies widely. For instance, C_6 contains 135 benign apps and is the biggest category, while C_1 contains 5 *DroidKungfu* malicious apps and is the smallest category. Although it would be better if we had a balanced suite that contained the same number of samples in each category, in reality, this is really difficult to achieve for two reasons. First, some malware families are more prevalent than others and contain more malicious apps. Second, after filtering apps with the criteria mentioned in Section 4.1, we have no control on the resulting app distribution. In future we intend to obtain more malicious apps and create a more balanced benchmark suite.

As shown in the matrix, DroidCat almost always correctly labeled benign apps. Among the 136 benign apps, DroidCat only confused 1 app with *MALICIOUS* apps; however, it did not distinguish well between C_2 and C_7. For instance, 8 of C_2 apps were wrongly labeled as "C_7", while 8 C_7 apps were wrongly labeled as "C_2". We made similar observations between C_5 and C_7. By further checking the C_7 row and C_7 column, it is clear that DroidCat was more likely to confuse C_7 with other categories. This is because C_7 *(MALICIOUS)* consists of malicious apps from a variety of minor malware families, which apps may not share sufficient behavioral characteristics to uniquely differentiate themselves from other kinds of malware or benign apps.

Effectiveness Metrics Values.

Overall, DroidCat achieved 81% precision, 82% recall, 81% F1, and 92% accuracy. In particular, DroidCat achieved as high as 99% accuracy for C_1, C_3, and C_4, but attained the highest F1 score, 93%, for C_4 and C_6. Since accuracy does not always coincide with F1, we have included both metrics to present DroidCat's capability from different perspectives. According to both F1 score and accuracy, we found DroidCat worked less effectively for C_2 and C_7, because it either confuses one of these categories with the other one, or with C_6 *(BENIGN)*.

Hypothetically, the more sample apps a category has,

Table 3: Evaluation results of DroidCat's unified malware detection capability

Confusion Matrix								Effectiveness metrics			
Actual category (# of apps)	Predicted							P	R	F1	A
	C_1	C_2	C_3	C_4	C_5	C_6	C_7				
C_1: DroidKungfu (5)	**4**	0	0	0	0	1	0	100%	80%	90%	99%
C_2: ProxyTrojan/NN (27)	0	**13**	0	0	0	6	8	57%	48%	52%	91%
C_3: GoldDream (11)	0	1	**10**	0	0	0	0	91%	91%	91%	99%
C_4: Plankton (8)	0	1	0	**7**	0	0	0	100%	88%	93%	99%
C_5: FakeInst (33)	0	0	0	0	**26**	2	5	93%	79%	85%	97%
C_6: BENIGN (136)	0	0	0	0	0	**135**	1	87%	99%	93%	92%
C_7: MALICIOUS (51)	0	8	1	0	2	12	**28**	67%	55%	60%	86%
Overall effectiveness								81%	82%	81%	92%

the more data we can use to train the classifier for that category, and more effectively DroidCat should perform. However in Table 3, we have not observed any correlation between the number of apps and the effectiveness. For instance, DroidCat worked best for C_4, although C_4 does not have the most apps. It worked worst for C_2, although C_2 does not have the fewest apps. It may be that some malware families are harder to recognize than others.

A Case Study of DroidCat's Effectiveness.
To understand why DroidCat made mistakes when classifying software, for each cell of Table 3, we gathered the applications and computed their average metric values for the Top-10 differentiating features shown in Figure 2. By comparing the values between the correctly classified applications and those wrongly classified ones, we observed several interesting findings. First, the metric values of different malware had a lot of overlaps, making unified malware detection very challenging.

Second, compared with benign apps, GoldDream malicious apps were almost always observed to have a higher percentage for both external explicit ICCs and ICCs carrying URI data only. The reason is such malware always spies on incoming/outgoing SMS messages and phone calls, and uploads the data to a remote server without users' awareness. When a GoldDream malicious app had no external explicit ICC and no ICC carrying URI only, DroidCat could not classify it correctly.

Third, Plankton malicious apps seemed to usually have a higher percentage for both view event handlers and external explicit ICCs. Such malware displays unwanted advertisements as notifications. With more view event handlers implemented, the malware eagerly monitors for user actions, so that user actions can trigger more advertisement displays and more ICC invocations. However, there was a Plankton malicious app which had very low percentage of view event handlers and almost 0% external explicit ICC invocations. DroidCat wrongly classified it as ProxyTrojan/NN.

Fourth, most FakeInst malicious apps manifested a higher percentage for view event handlers, external explicit ICCs, and ICCs carrying URI data only. This is because such apps send SMS messages to premium-rate numbers of services but present free content. With more view event handlers implemented, the malware eagerly monitors for user actions, triggering more ICCs invoked to send SMS messages to premium-rate numbers and to charge more money for actually free services. However, there were 7 FakeInst malicious apps which had very low percentage of view event handlers and almost 0% external explicit ICCs and ICCs carrying URI only. DroidCat could not correctly identify their category.

In summary, DroidCat captured meaningful behavioral differences between various categories of Android apps. It managed to correctly classify the majority of applications. However, to classify apps more precisely in future, we may need to closely examine the execution data of the apps that DroidCat wrongly classified and define more behavioral metrics.

Finding 1: *DroidCat conducts unified malware detection with 81% precision, 82% recall, 81% F1, and 92% accuracy overall on the benchmarks. In particular, 99% accuracy is achieved for 3 malware families: DroidKungfu, GoldDream, and Plankton.*

5.3 Conventional Malware Detection Results

We also trained and tested DroidCat to simply distinguish between malicious and benign apps without malware categorization. We reused the feature data extracted for the 271 apps, but relabeled the data of malicious apps by replacing various malware category names solely with "*MALWARE*". Then we repeated the LOOCV process to evaluate DroidCat's effectiveness. Our results show that DroidCat effectively detected malware with 95% precision, 99% recall, 96% F1, and 99% accuracy.

Compared with unified malware detection, DroidCat worked more effectively in the conventional malware detection setting. To understand why the effectiveness is different, we further checked the prediction results for individual apps. We observed that unified malware detection commits more mistakes by wrongly classifying malicious apps as benign for some malware families. For example, in the C_2 row of Table 3, 6 apps were wrongly labeled as benign; however, in the conventional setting, only 3 of these apps were classified as benign. One possible reason is that there are much fewer malicious apps in C_2 than in $MALWARE$ (27 vs. 135) so that DroidCat was less trained to separate C_2 apps from benign ones.

This evaluation also indicates that unified malware detection is more challenging than conventional malware detection. As apps from different malware families behave similarly, malicious apps in the same family may not share sufficient dynamic characteristics to distinguish themselves from apps in other families.

Finding 2: *DroidCat conducts conventional malware detection with 95% precision, 99% recall, 96% F1, and 99% accuracy, working more effectively than in the unified malware detection setting. It indicates that unified malware detection is more challenging.*

5.4 Sensitivity to Feature Selection

To understand how sensitive DroidCat is to different selections of features, we investigated different ways to select features. In addition to the default configuration of 70 features (D*) used in DroidCat, we also explored the

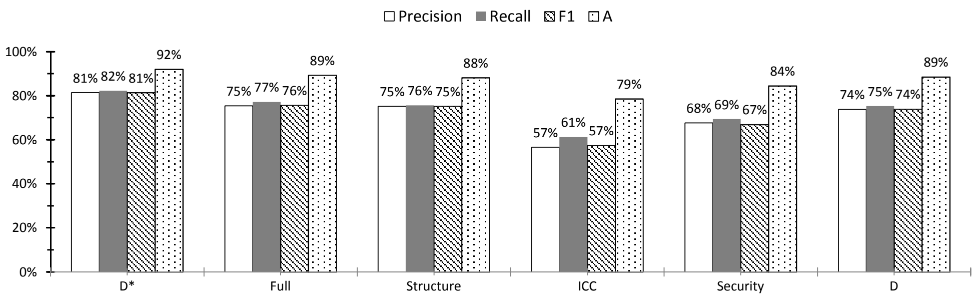

Figure 3: DroidCat's effectiveness with six alternative feature sets

following five feature sets.

- ***Full***: All 122 behavioral metrics defined for our characterization study were used as features. Hypothetically, this feature set should produce an equally good or even better classifier than D*, since in addition to the 70 features, the full set also includes 52 metrics with indistinguishable differences between benign and malicious apps. Even if these metrics cannot help improve the classification result, machine learning should be able to assign lower weights to them to remain effective.

- ***Structure***: Only the 32 noticeably different metrics in the structure dimension were selected as features. Although these metrics are already included in the default 70 features, we do not know how important these 32 metrics are compared with the other 38 metrics in D*. Hypothetically, this feature set should produce a worse classification than D*, because more than half of the features used in D* are not included.

- ***ICC***: Only the 5 noticeably different metrics in the ICC dimension were selected as features. Since the feature set is so small, it is quite possible that this small set will work much more poorly than D*.

- ***Security***: Only the 33 noticeably different metrics in the security dimension were selected as features. Along with the *Structure* and *ICC* feature sets, this feature set was examined to understand how features selected in different dimensions can affect the overall effectiveness of DroidCat.

- ***D***: All of the 36 substantially disparate metrics of Table 1 were selected as features. Similar to the above three subsets, *D* was also investigated to understand which feature subset plays an important role in affecting DroidCat's effectiveness. If any of these four subsets performs equally well with *D**, it means that we can further refine the default feature set to improve DroidCat's efficiency.

We compared the five new feature sets with D* by separately feeding them to the Random Forest algorithm for unified malware detection. The evaluation results are shown in Figure 3.

Interestingly, *D** worked best, even better than *Full* for all metrics. The reason is that the additional 52 nondiscriminatory metrics may provide noisy information and mislead the classifier to perform poorly. Therefore, *D** is more desirable, because it achieves better effectiveness with fewer features. This experiment also demonstrates that more features do not necessarily improve classification capability. Instead, they can worsen the capability especially if they seem to be nondiscriminatory.

The *D* set had similar effectiveness to *Full*, albeit worse than *D**. The 36 substantially disparate metrics can capture the behavioral differences among apps in different categories. Therefore, in cases when computation resources are limited, and fast model training and testing approaches are required, the *D* set can be used.

The *Structure* set had comparable effectiveness with *D*, but was significantly better than *ICC* and *Security*. This indicates that the behavioral differences between categories are more effectively reflected by the chosen metrics in structure dimension. Unsurprisingly, *ICC* performs worst because its feature set is too small.

Finding 3: *D*, the default feature set of DroidCat, exhibits the best effectiveness compared with other five alternative feature sets, because it balances well the diversity and relevance of features.*

5.5 Sensitivity to Learning Algorithm Choice

To understand how sensitive DroidCat is to the selection of machine learning (ML) algorithm, in addition to Random Forest, we also experimented with seven other supervised learning algorithms.

- ***Random Forest (RF)*** [34], or random decision forest, is an ensemble learning method for classification, regression, and other tasks. It operates by constructing a multitude of decision trees at training time and outputting the class on which the majority agrees (classification), or the mean prediction of all decision trees (regression). We used 100 decision trees to perform classification separately, and then output the class for which the majority trees vote.

- ***Support Vector Machine (SVM)*** [22] SMV is a discriminative classifier that produces a separating hyperplane to categorize samples when given labeled training data. By configuring SVM with 2 different kernel functions we obtained SVM-linear and SVM-rbf.

- ***Decision Tree (DT)*** [45] is a non-parametric learning method that creates a model to predict the value of a target variable by learning simple decision rules inferred from data features. We used the default random number generator (*random_state=None*).

- ***k-Nearest Neighbors (kNN)*** [13] is a non-parametric learning method whose input consists of the k closest training examples in the feature space and output is a class membership. Given a specific object, kNN classifies it according to the class of the majority of its k nearest neighbors. We set k to 5.

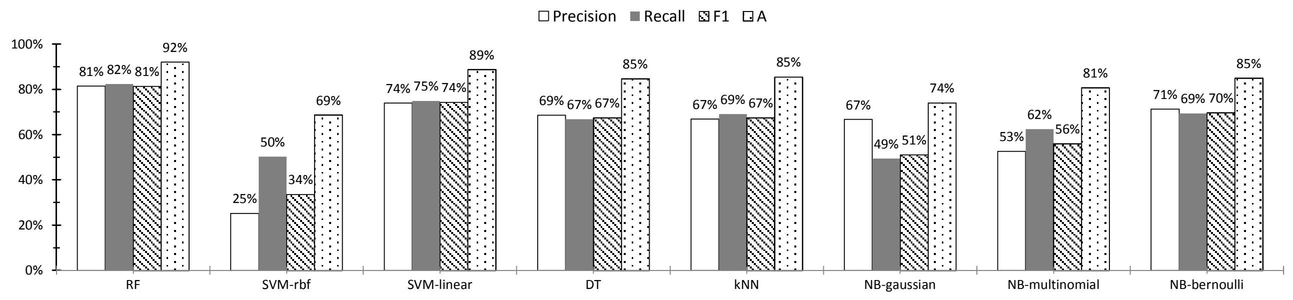

Figure 4: DroidCat's effectiveness with eight alternative ML algorithms.

- ***Naive Bayes (NB)*** [9] is a probabilistic classifier which applies Bayes' theorem with strong independence assumptions between features. To estimate the parameters for each feature's distribution, users usually assume that their features have a Gaussian, Multinomial or Bernoulli distribution. We investigated three NB algorithms: NB-gaussian, NB-multinomial, and NB-bernoulli, corresponding to those assumed distributions.

We fed the default feature set $D*$ to each of the above ML algorithms, and thus trained eight classifiers. Our evaluation results of their unified malware detection effectiveness are shown in Figure 4. Based on our experiments, *RF* performs significantly better than all other ML algorithms, perhaps because it gathers diverse independent classifiers to predict separately, and only reports the prediction that the most classifiers agree on.

SVM-linear has the second best effectiveness, while *SVM-rbf* performs the worst among all explored algorithms. With the same *SVM* algorithm applied, DroidCat's effectiveness is significantly affected by the selection of kernel functions (i.e., configuration) in *SVM*.

We observed similar differences between the three variant algorithms of *NB*. *NB-bernoulli* has the third best effectiveness, while *NB-gaussian* gets the second worst effectiveness. One possible reason is that the Bernoulli-distribution assumption aligns better with the actual distribution of all feature values of the benchmarks.

Neither *DT* nor *kNN* works as well as the best setting of other kinds of ML algorithms, including *RF*, *SVM*, and *NB*. This may indicate that these algorithms are not a good fit for the classification problem. For instance, when features have complicated interferences with each other, a decision tree is not capable of modeling the relationship. For small categories, such as C_1 with five members, *kNN* cannot work well. The reason is that for any app whose actual category is C_1, there may be not sufficient support among the five nearest neighbors in training data to correctly label the app.

Finding 4: *RF outperforms all other investigated algorithms. The effectiveness of DroidCat depends both on which ML algorithm is used and how that algorithm is configured.*

5.6 Analysis Overhead

There are five sources of analysis overhead of DroidCat: APK instrumentation, trace collection, feature computation, model training, and model testing.

For each benchmark app, the instrumentation overhead varied from 20 seconds to 80 seconds, with an average of 50 seconds. To collect traces, DroidCat invokes Monkey to run every app for 10 minutes. Compared with the original uninstrumented version, the run-time slowdown of the instrumented version for trace collection was at most 3%. The feature computation took 24 seconds on average. As these three types of overhead comprise per-app time cost, the overall analysis cost of instrumenting and tracing apps and computing features is proportional to the number of apps under the unified malware detection.

With training and testing data ready, the average training time of Random Forest was 5 seconds, while the testing time was 2 seconds.

Finding 5: *DroidCat has very low analysis overhead, which makes it applicable to large app sets for unified malware detection.*

6. THREATS TO VALIDITY

In addition to the benign apps, our benchmark suite only includes five malware families, each of which covered at least five apps, and one synthesized family to include miscellaneous malicious apps. This small number of categories may be not sufficient to evaluate DroidCat's capability for unified malware detection. In the future, we will include additional malware in our existing families and new malware families in a revised benchmark suite, for more balanced training data.

We took the same approach as prior work [53] to identify benign apps and malicious apps with VirusTotal. We created the oracle category label for each included benchmark app based on VirusTotal's output. The validity of DroidCat depends on the correctness and completeness of VirusTotal in identifying malware.

The effectiveness of DroidCat is affected by the dynamic coverage of execution traces. After running each app with Monkey for 10 minutes, we included the app as a benchmark as long as the execution trace covered more than 50% code. However, with more manual inspection, we realized that the collected traces contain a lot of repetitively executed paths, because Monkey generated many similar or identical inputs again and again. In future, we will implement filters for Monkey-generated inputs to condense duplicated executions, and to diversify the collected execution paths. In this way, we will achieve better dynamic coverage in the 10-minute execution trace, lower the bar to include more apps as benchmarks, and better characterize the dynamic behaviors of each app. Alternatively, we may obtain the same dynamic coverage by running each app for fewer minutes.

We designed 122 behavioral metrics in three dimensions to comprehensively capture any difference between benign apps and different malware families. We aimed to leverage such diverse behavioral profiles to make DroidCat robust to the obfuscation attacks which target specific dynamic profiles. However, we have not implemented any such attack to actually verify DroidCat's robustness. In future, we will implement various attack models to evaluate the resilience of DroidCat.

There are various malware detection and categorization techniques proposed by other researchers. It would be great if we could run those tools against the same benchmark suite and conduct a fair comparison. However, most relevant tools are not publicly available, making such tool-comparison experiments impractical. Therefore, we have open sourced DroidCat and published our benchmark suite to facilitate any future comparison between malware detectors and classifiers.

7. RELATED WORK

This section describes related work on (1) dynamic Android characterization studies, (2) malware detection, and (3) malware categorization.

Dynamic Characterization for Android Apps. There are very few empirical studies characterizing the runtime behaviors of Android apps [17, 54, 61]. For instance, Zhou et al. manually analyzed 1,200 malware samples to understand malware installation methods, activation mechanisms, and the nature of carried malicious payloads [61]. Cai et al. instrumented 114 benign apps for method calls and ICCs, and investigated the dynamic behaviors of benign apps [17]. These two studies either focus on malicious apps or benign ones. However, our systematic dynamic study characterizes both app groups to better understand their common behavioral profiles and divergent ones. Canfora et al. profiled Android apps to characterize their usage of resources like CPU, memory, storage, and network, and then leveraged the profiles to detect malware [18]. Similar to their research methodology, we have also first profiled Android executions and then leveraged some of the profiles to do unified malware detection; however, our profiles focus on method invocations and ICCs.

Android Malware Detection. Researchers have mainly taken two kinds of approaches to detect malware: static and dynamic.

The static approaches analyze source code and/or manifest files to detect any abnormal control flow, data flow, call graph, API call, ICC usage, or permission request [11, 26, 32, 56, 57, 59]. For instance, Droidmat extracted static information about permissions, deployment of components, Intent message passing, and API calls to train a machine learning model for malware detection [56]. Apposcopy built inter-component call graphs to detect any subgraph matching the signature of known malware families like DroidKungfu [26]. ICCDetector first performed systematic characterization of ICC patterns in benign apps and malicious ones, and then used the ICC patterns to train a model to distinguish malicious apps from benign ones [57]. However, these approaches cannot always precisely detect malware, because how a program behaves dynamically is not always statically inferable. Furthermore, they are subject to the well-adopted attacks that intentionally obfuscate code and make malicious apps to appear benign.

In comparison, the dynamic approaches provide a complementary way to detect malware [12, 16, 19, 49]. They observed behavioral differences between benign and malicious apps, and then leveraged machine learning to train a model based on the differences. For example, Crowdroid collected system calls and counted the occurrence of each unique call to create feature vectors [16]. Andromaly monitored the consumption of CPU, memory, and battery to extract features [49]. StormDroid used both the static feature of permission requests in manifest files, and dynamic features like sensitive APIs, call sequences, and invocation counts to detect malware [19]. In comparison, DroidCat does not instrument system calls, because various wide-adopted attacks leverage code polymorphism to obfuscate system calls [27, 40, 50]. It does not monitor resource consumption either, because CPU or battery usage does not reflect fine-grained program behaviors. DroidCat traces method calls and ICCs, and uses the relative percentage of each call or ICC to statistically characterize dynamic behaviors.

Android Malware Categorization. Two approaches have been proposed to categorize malware [23, 58]. Xu et al. traced system calls, investigated three alternative ways to graphically represent the traces, and then leveraged the graphs to categorize malware [58]. Dash et al. generated features at different levels, including pure system calls, decoded Binder communication, and higher-level behavioral patterns like file system access which conflate sequences of related system calls. Both approaches require users to provide a set of malicious apps, instead of any random unknown apps, to identify the malware families. This requirement may cause extra effort of using malware detection tools.

8. CONCLUSION

We presented DroidCat, the first dynamic *unified* malware detection approach. DroidCat not only detects malware, but also identifies the malware family. Specifically, DroidCat instruments Android apps to trace ICCs and all invocations of methods defined in user code, third-party libraries, and the Android framework. Based on the traces, 70 behavioral features are extracted to characterize each app. Random Forest is used to predict whether a given app is benign or belongs to a certain malware family. Different from static malware detectors, the 70 features in DroidCat measure the dynamic occurrence frequencies of method calls and ICCs, which are impossible to measure by static approaches. Different from many learning-based dynamic malware detectors, DroidCat defines a *diverse* set of features according to a systematic characterization study on the behavioral differences between benign apps and malicious ones.

Using 136 benign apps and 135 malicious apps, our evaluation with LOOCV reveals that DroidCat conducts unified malware detection with 81% precision, 82% recall, 81% F1, and 92% accuracy. With a deeper study on DroidCat's effectiveness results, we observed that DroidCat's feature set captured meaning behavioral differences between malware families and benign apps, although some tricky cases still require far more sophisticated behavioral features in future. DroidCat can achieve even better effectiveness when simply distinguishing malware from benign apps, which is 95% precision, 99% recall, 96% F1, and 99% accuracy. Our evaluation reveals that unified malware detection is more challenging than conventional malware detection, because malicious apps of different malware families behave similarly, while the apps from the same family do not always share sufficient characteristics to uniquely indicate their category. By investigating different feature sets, we have found that the selected 70-feature set embodies a good trace-off between diversity and relevance: all these features show noticeable differences between the two app groups from different perspectives. Our exploration of different ML algorithms also demonstrates that Random Forest works best.

9. REFERENCES

[1] Android - application components. http://www.tutorialspoint.com/android/android_application_components.htm.

[2] android/plankton. http://www.avgthreatlabs.com/us-en/virus-and-malware-information/info/android-plankton/.

[3] First ever android sms trojan targeting u.s. users. https://blog.kaspersky.com/fakeinst-targets-us-users/4601/.

[4] Proxy trojan. http://www.webopedia.com/TERM/P/Proxy_Trojan.html.

[5] Security alert: New android malware – golddream – found in alternative app markets. https://www.csc2.ncsu.edu/faculty/xjiang4/GoldDream/.

[6] Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets. https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html.

[7] The ultimate android malware guide: What it does, where it came from, and how to protect your phone or tablet. 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/.

[8] Why every statistician should know about cross-validation. http://robjhyndman.com/hyndsight/crossvalidation/.

[9] *Russell, Stuart J. and Norvig, Peter*. Artificial Intelligence: A Modern Approach, 2003.

[10] Android malware accounts for 97% of all malicious mobile apps. http://www.scmagazineuk.com/updated-97-of-malicious-mobile-malware-targets-android/article/422783/, 2015.

[11] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, 2013.

[12] 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. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, 2015.

[13] N. S. Altman. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3):175–185, 1992.

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

[15] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[17] H. Cai and B. Ryder. Understanding application behaviours for android security: A systematic characterization. Technical Report TR-16-05, Virginia Tech, May 2016. http://hdl.handle.net/10919/71678.

[18] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, 2016.

[19] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 377–388, 2016.

[20] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015.

[21] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[22] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 1995.

[23] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying Android malware based on runtime behavior. *Mobile Security Technologies (MOST)*, 2016.

[24] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995.

[25] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[26] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *FSE*, 2014.

[27] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 418–430. IEEE, 2008.

[28] H. S. Galal, Y. B. Mahdy, and M. A. Atiea. Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, pages 1–9, 2015.

[29] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, 2013.

[30] Google. Android emulator. http://developer.android.com/tools/help/emulator.html, 2015.

[31] Google. Android Monkey. http://developer.android.com/tools/help/monkey.html, 2015.

[32] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[33] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[34] T. K. Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, 1995.

[35] 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. Sen. Netw., 2015.

[36] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, pages 351–366, 2009.

[37] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. Soot - a Java bytecode optimization framework. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[38] J. Lee, K. Jeong, and H. Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1970–1977. ACM, 2010.

[39] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*, 2015.

[40] 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.*, 2012.

[41] J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao. Impeding behavior-based malware analysis via replacement attacks to malware specifications. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2016.

[42] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of USENIX Security Symposium*, 2013.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[44] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[45] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1986.

[46] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[47] G. Santafe, I. Inza, and J. A. Lozano. Dealing with the evaluation of supervised classification algorithms. *Artificial Intelligence Review*, 44(4):467–508, 2015.

[48] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, 2012.

[49] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. “Andromaly”: A behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[50] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 214–233. Springer, 2011.

[51] S. V. Stehman. Selecting and Interpreting Measures of Thematic Classification Accuracy. *Remote Sensing of Environment*, 62(1):77–89, 1997.

[52] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[53] K. Tian, D. Yao, B. G. Ryder, and G. Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. *2016 IEEE Security and Privacy Workshops (SPW)*, 2016.

[54] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[55] B. Wolfe, K. Elish, and D. Yao. High precision screening for android malware with dimensionality reduction. In *Proceedings of the 2014 13th International Conference on Machine Learning and Applications*, 2014.

[56] 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 *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[57] K. Xu, Y. Li, and R. H. Deng. ICCDetector: ICC-Based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[58] L. Xu, D. Zhang, M. A. Alvarez, J. A. Morales, X. Ma, and J. Cavazos. Dynamic android malware classification using graph-based representations. In *Cyber Security and Cloud Computing (CSCloud), 2016 IEEE 3rd International Conference on*, pages 220–231, 2016.

[59] 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 *Computer Security-ESORICS 2014*, pages 163–182. Springer, 2014.

[60] 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. of the International Conference on Software Engineering (ICSE)*, 2015.

[61] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.