ISR Institute for Software Research
University of California, Irvine

Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware

Joshua Garcia
University of California, Irvine
joshua.garcia@uci.edu

Mahmoud Hammad
University of California, Irvine
hammadm@uci.edu

Sam Malek
University of California, Irvine
malek@uci.edu

January 2016
ISR Technical Report # UCI-ISR-16-2

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware

Joshua Garcia, Mahmoud Hammad, and Sam Malek
Institute for Software Research, University of California, Irvine
Department of Informatics, University of California, Irvine
{joshug4, hammadm, malek}@uci.edu

***Abstract*—The number of Android malware apps are increasing very quickly. Simply detecting and removing malware apps is insufficient, since they can damage or alter other files, data, or settings; install additional applications; etc. To determine such behavior, a security engineer can significantly benefit from identifying the specific family to which an Android malware belongs. Additionally, techniques for detecting Android malware, and determining their families, lack the ability to handle certain obfuscations that aim to thwart detection. Moreover, some prior techniques face scalability issues, preventing them from detecting malware in a timely manner.**

To address these challenges, we present a novel machine learning-based Android-malware detection and family-identification approach, RevealDroid, that leverages a small, simple set of selectable features—of which the simplest set of features achieves obfuscation resiliency, efficiency, and accuracy. This result is highly surprising, given that a wide variety of techniques require complex program analyses (e.g., precise data-flow analysis) or large sets of features (e.g., hundreds of thousands of features), leading to scalability problems and lack of resilience to obfuscation. Specifically, our selected machine-learning features leverage categorized Android-API usage, which represent semantics of Android apps. We assess RevealDroid's accuracy and obfuscation resilience on an updated dataset of malware from a diverse set of families, including malware obfuscated using various transformations. We further compare RevealDroid against other state-of-the-research and state-of-the-practice approaches for malware detection and family identification. [1]

I. Introduction

Mobile devices have become ubiquitous, and are still growing quickly. Among such devices, Android has become the dominant platform and is deployed on hundreds of millions of devices around the world. With this widespread usage, an increasing number of malware applications (*apps*) have been found on such devices and the repositories that distribute mobile apps (e.g., Google Play). These malware increasingly resemble their counterparts in Desktop PC environments [5], [1], demonstrating the growing sophistication of mobile malware. Consequently, a significant amount of effort has been expended on producing techniques to detect Android malware.

Existing work on Android malware detection [24], [49], [58], [28], [33], [56], [39], [46], [16], [31], [18] has focused on distinguishing between benign and malicious apps. For example, previous work has demonstrated how large-scale data mining, with some program analysis, can be utilized to assess whether an Android app is benign or malicious [18], [26], [27], [20]. Although accurately making such a distinction is an important step towards fighting the growing prevalence of malware on Android devices, simply declaring an app as malicious and removing it is not enough to address the damage it may have done once deployed [32]. Engineers that assess the impact of a malware app must determine if other apps, files, or settings may have been damaged or altered; whether there are any remaining malicious or problematic services or processes that have been compromised; if any sensitive data has been stolen or leaked; if any unlawful or illegitimate financial charges have been made due to the malware's presence; etc. To make such a determination, a security engineer can significantly benefit from *identifying the specific family to which an Android malware belongs*. The family of a malware app can be coarse-grained (e.g., Trojan, virus, worm, spyware, etc.) or finer-grained, where more specific families (e.g., DroidKungFu [57], DroidDream [57], Oldboot [11], etc.) are identified. Knowledge of the family to which an Android malware belongs can help an engineer determine the specific steps that need to be taken to mitigate or undo damage caused by the malware.

Complicating the detection and family identification of Android malware are transformations that obfuscate apps in order to evade detection and family identification by anti-malware software [9], [17], [37]. For example, *Agent.BH!tr.spy* steals information by sending emails using SMTP with TLS authentication [17], thus hiding the stolen data in a cryptographic protocol. A recent study of Android malware obfuscation has demonstrated that simple transformations can prevent ten popular anti-malware products from detecting any of the transformed malware samples, even though prior to the transformations those products were able to detect those malware samples [37]. Thus, malware detection must be designed to *defeat these evasion techniques*. To achieve this goal, malware detection techniques can utilize program analyses that focus on the key semantics and behavior performed by a malware (i.e., behavior as represented by control flow or data flow of a program), particularly in its interactions with the system APIs and libraries that are external to the app, rather than just on syntactic aspects of its implementation (e.g., identifier name or string constants). However, the extent to which recent Android-malware detection techniques are resilient to modern transformation attacks is not well-understood. Existing studies have largely applied their techniques to malware that do not use any, or very limited, obfuscation [41], [53], [18], [26]. These techniques use features that are not resilient to obfuscations (e.g., features based on control flow [41], [26] or constant strings [53], [18]).

[1] ISR Technical Report UCI-ISR-16-2

To further reduce Android malware propagation and damage, detection or family identification of such malware should be *scalable*. Some state-of-the-art techniques run into scalability issues and can take hours or up to an entire day to analyze even a single app [31], [20]. Cumulatively, this delayed analysis can allow Android apps to propagate undetected for a longer period of time and, thus, cause more damage. Furthermore, it can prevent users from scanning apps directly on their Android devices, which is important given that Android markets have relatively poor vetting processes [58], [23]. Consequently, it is desirable to utilize features that can be extracted efficiently for detection and family identification of Android malware apps, even obfuscated ones.

In this paper, we introduce *RevealDroid*, a lightweight machine learning-based approach for detecting malicious Android apps and identifying their families. RevealDroid leverages a small, simple set of selectable features and machine-learning classifiers—of which the simplest set of features achieves obfuscation resiliency, efficiency of analysis, and accuracy. This result is highly surprising, given that a wide variety of techniques require complex program analyses (e.g., data-flow analysis [20], [53]) or large sets of features (e.g., hundreds of thousands of features [18], [26]), leading to scalability problems and lack of resilience to obfuscation. More specifically, our selected machine-learning features leverage categorized Android-API usage, which represent semantics of both benign and malicious Android apps.

RevealDroid is capable of accurately detecting malicious apps and identifying their families at above 97% for un-transformed apps and above 94% for transformed apps, and can do so, on average, for an app, in under 16 seconds. RevealDroid can maintain this accuracy even for obfuscated apps that it has never seen. We evaluate RevealDroid's detection and family identification accuracy by comparing its ability to correctly identify malware and classify its family on a dataset of over 28,000 benign apps and over 23,000 malware apps from two different malware repositories. We further compare RevealDroid's detection and family-identification accuracy against state-of-the-research approaches: Adagio [26] and MUDFLOW [20], both of which are approaches for malware detection; and Dendroid [41], an approach for malware-family identification. RevealDroid has an overall greater accuracy by about 13%-17% and mislabels 24%-30% fewer benign apps as malicious than MUDFLOW; and RevealDroid achieves up to 25% greater accuracy than Adagio. Additionally, RevealDroid achieves a 14%-60% higher classification rate than Dendroid. We further demonstrate that our learning-based classifiers meet or, in most cases, exceed the detection rate of commercial anti-virus products.

This paper makes the following contributions:

- RevealDroid demonstrates that highly lightweight analyses, combined with standard machine-learning techniques, can achieve high accuracy, scalability, and obfuscation resiliency—all at the same time.
- We construct an updated dataset of over 19,000 malware apps labeled with their 90 malware families and assess RevealDroid's family-identification accuracy on that dataset. We make this updated dataset available for researchers and practitioners [8].
- To evaluate RevealDroid's obfuscation resiliency, we apply several transformations to malware apps in order to obfuscate them and assess our ability to detect and identify families of those transformed apps. Using these transformed apps, we compare RevealDroid's accuracy for detection against Adagio and MUDFLOW, and for family identification against Dendroid.
- We assess the efficiency of RevealDroid's feature extraction and machine-learning classification. We show that RevealDroid's features can be more than 33-85 times faster than information-flow feature extraction—which are features used in a variety of Android malware detection tools [53], [20], [54]—while still exhibiting obfuscation resiliency and accuracy. We further demonstrate that RevealDroid can produce classifiers much more efficiently, as compared to other state-of-the-research tools.

The remainder of this paper is structured as follows. Section II discusses the manner in which we utilize machine learning as a foundation for RevealDroid, and compares the use of machine learning to signature-based methods for malware detection. Section III introduces RevealDroid and its design. Section IV covers our evaluation design, including the research questions we study; Section V discusses the evaluation results for each research question, and RevealDroid's limitations. Section VI covers work related to RevealDroid. Section VII concludes the paper and discusses possible future work.

II. Foundation

Malware detection and family identification can be placed into two categories: signature-based and machine learning-based [53]. For signature-based methods, security engineers must produce (often, manually) specifications that match against key properties of a malware family. For learning-based classification, techniques utilize machine learning to automatically determine whether an app is benign or malicious. Each Android app is an *instance* represented by *features* used to distinguish between apps supplied to learning algorithms (e.g., Android API methods or permissions used). A dataset is a set of instances along with their features.

To classify Android apps as benign, malware, or a specific malware family, we leverage *supervised* learning algorithms. For supervised learning, each instance is given a label; in the case of malware detection, the labels chosen are often simply "benign" or "malicious". The dataset is split into a *training* and *testing* set. A learning algorithm is applied to the training set in order to produce a *classifier*, which can then label apps as "benign" or "malicious". The testing set is passed as input to the classifier to assess its accuracy.

Signature-based methods are highly reliable for detecting known malware, but are often constructed manually and unreliable for detecting variants of known malware or zero-day malware. Learning-based methods require a sizeable dataset and properly selected features to ensure accuracy, but are more likely to generalize in their findings, making them particularly well-suited for identifying variants of known malware or zero-day malware. In this paper, we utilize learning-based methods for our Android malware detection.

III. REVEALDROID

To properly leverage learning-based methods, we must select features that are likely to distinguish both benign apps from malicious ones and different families of malware apps (e.g., DroidDream from DroidKungFu). Android malware detection and family identification can benefit significantly from the utilization of the Android platform itself to represent features of apps. In particular, the types of Android API methods that an Android app accesses often vary significantly between malware families, in order to perform different types of malicious behavior (e.g., sending SMS messages to premium-rate numbers, stealing location and identifier information, acting as a bot, listening for different activation triggers, etc.). We leverage this insight about distinguishing between and identifying Android malware to design an approach for classifying Android malware families. By focusing on Android-API invocations, which in different combinations tend to be malicious or benign, RevealDroid is capable of achieving obfuscation resilience.

In addition to leveraging API-based features, we intentionally design the features to be small, in terms of number of features and features types—with three feature types and 219 features in total. This design allows our resulting classifiers to be trained efficiently and quickly identify malware which, in turn, achieves scalability. Even though our machine-learning feature space is small, we carefully select features and provide different learning classifiers, which are capable of determining the combinations of features that exhibit malicious behaviors.

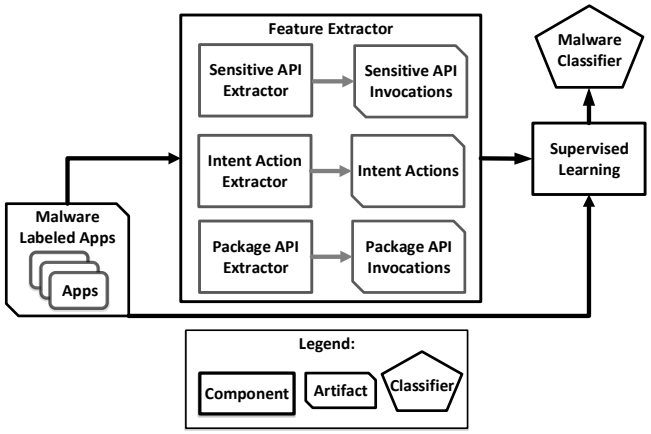

Fig. 1: Overview of RevealDroid's malware classifier production

Figure 1 depicts an overview of RevealDroid, our approach for constructing a malware classifier capable of distinguishing benign apps from malicious ones, and can further determine the family of an Android malware. The *Feature Extractor* component obtains a set of features used to distinguish between apps that are benign or belong to a malware family. To improve run-time efficiency, all three feature types can be extracted in parallel. These features, along with apps labeled with either their malware family or as benign, are passed as input to a supervised-learning algorithm—resulting in the construction of a classifier for identifying malware families.

RevealDroid contains a set of features that involve Android API usage so that they are obfuscation-resilient, represent core semantics of an Android app, and are relevant for determining if an app is malicious or belongs to a particular malware family. None of our selected features require complicated program analyses, making them both efficient and difficult to obfuscate. At the same time, RevealDroid allows its features to be used in different combinations, enabling trade-offs between obfuscation resiliency, efficiency, and accuracy (e.g., higher accuracy for family identification, unless apps are obfuscated). RevealDroid contains the following three types of Android API-based features: (1) Android API usage categorized by whether or not they provide access to security sensitive information or functionality—which is identified by *Sensitive API Extractor* in Figure 1, (2) *actions* of Android messages that an app may listen to—which is identified by *Intent Action Extractor* in Figure 1; and (3) Android API usage categorized by the package to which the API belongs, which is determined by *Package API Extractor* in Figure 1. Surprisingly, despite the simplicity of our features and only having a small number of features, our evaluation results will demonstrate that RevealDroid is capable of achieving high accuracy, scalability, and obfuscation resiliency (see Section V).

For each type of feature, this section explains its importance, and the manner in which the feature type is represented and extracted. The section then covers the labeling of apps and RevealDroid's use of supervised-learning algorithms to produce classifiers for detecting malware and identifying their families.

A. Sensitive API-Usage Extraction

Malware apps must invoke or access Android APIs to perform malicious behaviors (e.g., steal information, send SMS messages to premium-rate numbers to make unlawful financial charges, receive instructions from a remote server, etc.).

To that end, we utilize 30 categories that distinguish the behavior of an API, allowing a supervised-learning algorithm to determine if the particular usage of those categories is either malicious or characteristic of the actions performed by a particular malware family. 28 of these categories represent security-sensitive APIs, one category represents widget-based APIs, and another category represents any APIs not belonging to the other categories. The security-sensitive API categories are determined by SuSi [36], a machine-learning approach for categorizing Android source and sink API methods. For each category, *Sensitive API Extractor* determines the number of invocations per category an app makes to an Android API method, which are used as features for an Android app. Formally, the feature vector $SAPI_a = (s_1, ..., s_i, ..., s_{|C|})$, where C is the set of sensitive API categories, $s_i = |\{m \bullet m \in methods(i)\}|$, m is an invocation of a method in an Android app a, and $methods(i)$ is the set of methods in category $i \in C$.

To illustrate how such features can help distinguish malware families, Table I depicts features for a subset of categories from three Android malware families. For example, in Table I, the Geinimi sample invokes database (DB) APIs 37 times, and SMS APIs only once. The table shows that a supervised learning algorithm can determine that Geinimi samples only access the SMS API once, DroidKungFu1 invokes logging APIs a limited number of times (e.g., 35 times rather than over 220 times), and jSMSHider uses inter-process communication APIs (i.e., sending Android messages) in a very limited manner (e.g., 6 invocations rather than over 130).

TABLE I: Example Sensitive API features from known Android malware families

	DB	IPC	LOG	NET	SMS	Fam
mal4	37	133	246	23	1	Geinimi
mal5	7	139	35	24	0	DroidKungFu1
mal6	4	6	226	10	0	jSMSHider

It is possible to treat each access to particular Android API as a separate feature. However, such a design would result in a large feature space with over 26,500 features, resulting in possible scalability and accuracy issues for a supervised-learning algorithm or the resulting classifier [44], [50], [48].

B. Intent Action Extraction

Different families of malware activate based on different *actions* of *Intents* [57], which are messages sent and received by Android components. An *action* of an Intent specifies the expected behavior to be performed on receipt of the Intent (e.g., opening an editor), or an event that has occurred in the Android system (e.g., an indication that the device has finished booting). Consequently, Intent actions are important information useful for distinguishing between malware families. For example, DroidDream listens for Intents indicating the launch of the Android home screen; BeanBot listens for messages that request the initiation of a phone call.

To identify such actions, *Intent Action Extractor* analyzes an app's *Android Manifest* file and any *Broadcast Receiver* components to determine messages that an app may listen to. The Android Manifest file is an XML file included with every Android app. In that file, a developer can specify the actions of an Intent that the app may process. Broadcast Receivers listen to Intents broadcasted by other apps or the Android system. In particular, *Intent Action Extractor* examines the *onReceive* method of Broadcast Receivers, which are callbacks that process broadcasted Intents. By analyzing both the app's code and Manifest file, *Intent Action Extractor* obtains comprehensive information about actions that may activate different families of malware. For our approach, a total of 108 boolean features represent the actions that an app may process. More formally, the Intent actions feature vector $IA_a = (ia_1, ..., ia_i, ..., ia_{|I|})$, where I is the set of actions for Intents, $ia_i = 1$ if app a listens to action i in a Broadcast Receiver and $ia_i = 0$ otherwise.

TABLE II: Example Intent action features from three known Android malware families

	MAIN	BATT	SYS	PKG	Fam
mal4	1	0	0	0	DroidDream
mal5	0	1	1	0	DroidKungFu1
mal6	0	0	0	1	jSMSHider

Table II shows a simplified version of the Intent action features for three malware families: DroidDream, DroidKungFu1, and jSMSHider. Both DroidDream and DroidKungFu1 are malware families that utilize root exploits and enable remote control. However, they can be distinguished by the Intent actions they listen to: DroidDream listens to Intent actions corresponding to the launch of the Android home screen (MAIN); DroidKungFu1 listens to a variety of system events (SYS) and Intent actions related to battery consumption (BATT). jSMSHider is one of the rare malware families that register to receive Intent actions corresponding to packages (PKG) being installed, replaced, or removed on an Android device.

C. Package API-Usage Extraction

In situations where sensitive API features and Intent actions are insufficient, Android API usage categorized by packages is included as a feature to aid a classifier in detecting malware and distinguishing between malware families. These features have been shown to be useful for distinguishing malware families when manually specifying their signatures [25]. Consequently, we chose to include such features for detecting and identifying families of Android malware using machine learning. To that end, *Package API Extractor* in Figure 1 determines the number of API invocations per Android package. For example, if three methods of classes in the *android.telephony* package are invoked, then the feature corresponding to that package obtains a value of 3. Formally, the feature vector $PAPI_a = (p_1, ..., p_i, ..., p_{|P|})$, where $p_i = |\{m \bullet m \in methodPkgs(i)\}|$, P is the set of Android API packages, $methodPkgs(i)$ are the set of methods in package i, and m is an invocation of a method in an Android app a. By selecting packages to represent API usage, we reduce the feature space, similar to the case for information-flow features, to a total of 81 features, which helps to ensure efficient classifier production.

D. Labeling and Classifier Selection

RevealDroid can detect whether an app is benign or malicious, or determine the family to which a malware belongs. RevealDroid can produce different classifiers to perform these functions. The classifier constructed by RevealDroid depends on the labels used when the classifier is trained. Furthermore, RevealDroid is designed to use different classifiers—some of which may be better for identifying malware families, while others may produce better malware detectors.

To that end, RevealDroid can build multiple n-way classifiers, where n is the number of labels for an Android app. To simply detect whether an app is malware, the training set of Android apps can simply contain $n = 2$ labels: *benign* or *malicious*. For malware family identification, the number of labels correspond to the number of malware families in the training set. For example, Android Malware Genome contains 49 malware families, resulting in $n = 49$ for a malware classifier trained on Malware Genome.

The number of labels for family identification significantly increases the difficulty of correctly labeling an Android app, as compared to the 2-way classification when distinguishing between benign and malicious apps. Nevertheless, as our evaluation results will demonstrate, RevealDroid is capable of achieving high accuracy for identifying malware families of malicious apps.

The supervised-learning algorithm used to construct a classifier can considerably affect its resulting accuracy. Consequently, we (1) allow RevealDroid to utilize different learning algorithms and (2) assess the algorithms best-suited for Android malware detection and family identification in Sections V-E–V-F.

IV. Evaluation Design and Setup

To evaluate RevealDroid, we study its accuracy, efficiency, and resiliency to transformations intended to obfuscate malware. Furthermore, we compare RevealDroid to another state-of-the-research Android malware-family identification approach, Dendroid, and a detection approach, MUDFLOW. Specifically, we answer the following research questions:

- **RQ1**: Which combinations of RevealDroid's features and classifiers accurately distinguish between benign and malicious Android apps?
- **RQ2**: Which combinations of RevealDroid's features and classifiers accurately identify the specific family of a malicious Android app?
- **RQ3**: To what extent is RevealDroid's accuracy affected by transformations that obfuscate malware?
- **RQ4**: What is RevealDroid's run-time efficiency? How does this run-time efficiency compare to other learning-based approaches for malware detection?
- **RQ5**: How does RevealDroid's detection accuracy compare to other detection approaches?
- **RQ6**: How does RevealDroid's family identification capability compare to another state-of-the-research malware-family identification approach?

We implemented RevealDroid in Java for its feature extraction, malware detection, and malware-family identification. To construct the *Sensitive API Extractor*, *Intent Action Extractor*, and *API Extractor*, we leveraged Soot [43], a static analysis framework, and Dexpler [21], a translator from Android Dalvik Bytecode to Soot's intermediate representation. For machine learning, we selected Weka [29], a widely used machine-learning toolkit for Java.

For conducting our experiments, we leveraged two computing clusters. The first computing cluster (CC1) [3], has 35 compute nodes each with 8-core 2.60GHz CPUs and 64GB RAM, which are the compute nodes we utilized for our experiments. The second computing cluster (CC2) [4] has 108 compute nodes, where each node has between 8 and 64 cores, and 32GB-505GB RAM.

To assess RevealDroid's accuracy, we constructed a dataset of both benign and malicious Android apps. To obtain benign apps, we downloaded approximately 23,000 apps from two sources: *Google Play* [7], Google's official Android app repository, and *F-Droid* [6], an open-source repository of Android apps. Over 21,000 of the apps were obtained from Google Play, while the rest were obtained from F-Droid. Google Play is the official Android market and is vetted for malware. Consequently, the probability of having malware in our samples from Google Play is extremely low, and further alleviated by our use of machine learning. F-Droid apps are overwhelmingly benign apps for two reasons. First, apps uploaded to F-Droid are scanned for malicious behaviors before they are posted. Second, given that all F-Droid apps are open source, they are all open to scrutiny for malicious behaviors.

We obtained malware samples from three Android malware repositories: the Android *Malware Genome* project [57], the Drebin dataset [10], and *VirusShare* [12]. Malware Genome contains over 1,200 Android malware apps from 49 different malware families. We utilized 22,592 Android malware samples from VirusShare. We further leveraged 5,538 samples from the Drebin dataset, which includes the samples from the Android Malware Genome project.

V. Evaluation Results

For each research question, we discuss its importance, the specific experimental setup needed to study it, and our corresponding results. We then discuss the overall findings and limitations of our study.

A. RQ1: Detection Accuracy

To answer RQ1, we assess how accurate RevealDroid's features are for detecting whether an app is benign or malicious. To that end, we developed two approaches based on a *C4.5 decision-tree classifier* [35] and a *1-nearest-neighbor (1NN) classifier* [15] for labeling an app as either benign or malicious.

Table III shows the correct classification rate among the different combinations of three features: sensitive APIs (*SAPI*), Intent Actions (*IA*), and package APIs (*PAPI*). For each combination of features, classifiers, and apps, we performed a 10-fold cross-validation and report the rate of correctly classified apps. For this experiment, we utilized 23,366 benign apps and 28,130 malicious apps.

TABLE III: Detection results for different combinations of RevealDroid's features and classifiers.

Features	C4.5	1NN
SAPI	95.52%	95.00%
SAPI, PAPI	96.25%	96.29%
SAPI, PAPI, IA	97.10%	96.61%
PAPI, IA	96.70%	96.44%
SAPI, IA	96.42%	96.18%
PAPI	95.77%	95.71%
IA	86.63%	81.94%

All combinations of features exhibit a high correct classification rate. Feature combinations with sensitive API or package API features have a correct classification rate between 95% and 97%. Package API features alone or sensitive API features alone exhibit high accuracy—above 95% for both C4.5 and 1NN classifiers. Intent action features by themselves have the lowest classification rate, but are both above 81%. All the feature types together have the highest classification rate at 97%. The C4.5 classifiers had slightly higher accuracy than 1NN classifiers, mostly 1% higher accuracy. The greatest difference in accuracy between C4.5 and 1NN classifiers occurs for Intent action features alone, where there is a 5% difference.

To illustrate the high accuracy for detection of RevealDroid, we showcase additional results of RevealDroid's most accurate classifier, a C4.5 classifier using sensitive API, Intent actions, and package API features. Table IV depicts the 10-fold cross-validation results for that classifier, which includes the following: *Prec*ision indicates the extent to which the classifier produces false positives; *Rec*all shows the extent to which the

classifier produces false negatives; *F-Meas*ure is the weighted harmonic mean of precision and recall; *ROC Area* represents the discriminatory power of our classifier when distinguishing between benign and malicious apps; and the average weighted by the number of apps (*WAvg.*).

TABLE IV: Detection results for combinations of sensitive API, Intent action, and package API features using a C4.5 classifier

	Prec	Rec	F-Meas	ROC Area
Benign	96.90%	96.70%	96.80%	97.50%
Malicious	97.30%	97.40%	97.30%	97.50%
WAvg.	97.10%	97.10%	97.10%	97.50%

The table illustrates that RevealDroid's most accurate detection classifier obtains high accuracy for both benign and malicious apps, with an F-measure value of 97%. In fact, the precision and recall of benign apps alone and malicious apps alone is 97%. RevealDroid also demonstrates a high discriminatory power, as demonstrated by the 97% ROC Area for benign apps, malicious apps, and the weighted average. These consistently high results across multiple measures demonstrates RevealDroid's ability to detect malicious apps with high accuracy.

B. RQ2: Family Identification

Identifying an Android app as malware is insufficient for dealing with the damage it may cause. Once a malicious app is deployed, it may install other apps, steal information, modify settings, etc. Thus, determining the family to which an app belongs can aid engineers and end users in determining how to deal with the malicious app, besides simply removing it.

Original Android Malware Genome. To determine RevealDroid's ability to classify Android malware apps into families, we assessed RQ2 by utilizing the Android Malware Genome (AMG) [57], which contains over 1,250 apps and 49 malware families. To that end, we used RevealDroid to construct classifiers with 49 different labels, one for each family in AMG. We determined the combinations of classifiers and features that provided the most accurate classification of AMG.

Table V depicts the classification rate for the most accurate classifiers among the different combinations of our three feature types. For this experiment, we conducted a 10-fold cross-validation to assess the accuracy of our various classifiers and features. As in the prior experiment, the numbers of apps (No. Apps) in Table V vary due to the types of features used.

TABLE V: RevealDroid's classification rate for family identification utilizing different features and classifiers on AMG

Features	**C4.5**	**1NN**
SAPI	87.69%	87.29%
PAPI	88.48%	88.64%
SAPI, IA	91.51%	91.75%
IA, PAPI	90.58%	92.42%
SAPI, PAPI	91.18%	89.28%
SAPI, IA, PAPI	93.62%	92.98%

Overall, the accuracy of RevealDroid's malware-family classifiers is between 87% and 94% for all combinations of features and classifiers. These results showcase RevealDroid's ability to identify a malicious app with high accuracy. This outcome indicates that our API-based features are well-chosen for discriminating between malware families.

Combinations of our three feature types significantly increased the accuracy for family identification, which is difficult to do given the already high classification rate of either package or sensitive API features alone. Although the overall increase in correct classification rate is 4%-7%, these features significantly improved accuracy for specific families. For example, Intent action features raised the accuracy of samples from the GoldDream family, consisting of 47 samples, to 100% from 85% for sensitive API features. As another example, package API and Intent action features increased the accuracy for the DroidDream family, consisting of 16 samples, from 48% to 94% when combined with sensitive API features.

Expanded Android Malware Genome. To further assess our classifier and determine if more samples for particular families would improve our results, we significantly expanded the samples that exist in AMG. To that end, we utilized a set of Android malware samples from VirusShare [12], which contains over 24,000 unlabeled malware samples ranging from May 2013 through March 2014, whereas the original AMG samples are from August 2010 through October 2011. To identify the families of those samples, we leveraged VirusTotal [13], a service that contains metadata about malware. We constructed a client to obtain possible families identified by over 50 commercial antivirus products. For each Android malware sample in VirusShare, we recorded the malware family that appears most among the 50 products. From the VirusShare samples, we identified 857 samples from families that are part of the AMG project and extracted their features using RevealDroid. We combined those 857 samples with the original AMG samples to produce an expanded AMG (EAMG). As a result, we increased the number of samples by 68% of its original size. The overwhelming majority (76%) of the new samples belong to GingerMaster (305), Plankton (242), and KMin (107). This increase in samples is particularly stark for the GingerMaster family, which originally contained only 4 samples—a relatively low number for training a classifier.

TABLE VI: RevealDroid's classification rate for family identification utilizing different features and classifiers on EAMG

Features	**C4.5**	**1NN**
SAPI	84.38%	86.11%
PAPI	86.05%	87.40%
SAPI, IA	89.93%	91.09%
IA, PAPI	90.15%	91.79%
SAPI, PAPI	87.78%	88.65%
SAPI, IA, PAPI	90.54%	91.74%

To assess RevealDroid on EAMG, we performed a 10-fold cross-validation on EAMG using a C4.5 and 1NN classifier with the same combinations of features, similar to the previous experiment for malware-family identification. Table VI shows our results for EAMG.

Similar to our previous results, RevealDroid correctly classifies 84%-92% of the malware samples in EAMG. This consistently high accuracy, despite a significant increase in the dataset size, demonstrates the effectiveness of RevealDroid

for family identification. Furthermore, the trends regarding increases for specific families remain for EAMG as it did for AMG. For example, adding both Intent action features and package API features to sensitive API features improved the accuracy for the GoldDream family—consisting of 60 samples—from 85% to 95% and for the Zitmo family—consisting of 248 samples—from 77% to 88%. Lastly, whether the GingerMaster family could be reliably classified was unclear because there were only 4 samples in AMG. However, in EAMG, with an additional 305 GingerMaster samples, combinations involving sensitive API features obtained up to 91% accuracy.

The results for AMG and EAMG indicate that combinations of sensitive API features are highly accurate for identifying manually-labeled malware families.

VirusShare and Malware Genome Families. To assess RevealDroid's classifiers' effectiveness on more recent Android malware families, we evaluated those classifiers on a much larger set of malware samples from VirusShare. To produce a ground truth of families for malicious apps beyond those found in AMG, we again leveraged VirusTotal. We uploaded every sample in VirusShare to VirusTotal, which returned a label for each app. Each label was, in turn, parsed to obtain a family name. For example, a malicious app labeled as *Android/Fam.B* would be parsed to obtain "Fam" and highly generic terms (e.g., "Android"), single letter terms (e.g., "B"), and non-alphabetic terms were removed. After this parsing, if five or more antivirus products agreed on the family label for the app, we assigned that label to the app. The label that appears the most is accepted as the final family label, where ties are broken arbitrarily. To further improve the fidelity of the family labeling, we removed samples with low reputation scores that are not necessarily malware (e.g., some adware) or highly generic labels (e.g., some apps are literally labeled "generic") were also eliminated. We further combined this sample set with EAMG, resulting in a dataset of 90 families and 19,321 malware samples. Table VII depicts the family labels with 200 or more samples from that dataset, totaling 16,756 samples.

TABLE VII: Family labels with 200 or more samples after combining VirusShare and EAMG families

Family	No. of Samples
UMpay	216
Plankton	249
root exploit	252
Airpush	269
Opfake	302
Utchi	306
GingerMaster	353
Admogo	362
Youmi	518
Adwo	523
DroidKungFu	538
SMSreg	558
Agent	856
Fakeinst	1655
Dowgin	2063
SMSsend	7736

Table VIII shows the correct classification rate for RevealDroid's most accurate classifiers. RevealDroid's classifiers all achieve 85%-88% correct classification rates. This result is remarkable given the number of samples and the possible family labels per sample: A random classifier would obtain only about 1.1% correct classification rates; and a naive classifier that simply marks every app with the most frequent family label (SMSsend) would only obtain a 40% classification rate. Consequently, for every 10 apps that RevealDroid classifiers label, only 1 app would need to be manually corrected.

TABLE VIII: RevealDroid's classification rate for family identification utilizing different features and classifiers on VirusShare and EAMG families

Features	C4.5	1NN
SAPI	85.52%	86.42%
SAPI, IA	87.83%	87.06%
SAPI, IA, PAPI	88.24%	87.25%
PAPI	85.75%	85.06%
SAPI, PAPI	86.39%	85.96%
IA, PAPI	87.64%	86.84%

A handful of malware families make a significant difference to the results of this particular RevealDroid classifier. GingerMaster samples and Dowgin samples tend to be classified as each other, due to similar API usage. However, this issue can be alleviated using the EAMG classifier, which obtains 91% accuracy for the GingerMaster family. The Root exploit label tends to be classified as SMSsend, GingerMaster, or Gingerbreak. Given that Gingerbreak is a root exploit, and GingerMaster utilizes that particular exploit, these classifications are relatively close. Nevertheless, RevealDroid is not designed to detect malware at the root level, since it is an application-level analysis: A more specific analysis is likely needed to accurately detect root exploits for Android. The last major family affecting RevealDroid's classification rate is the relatively generic malicious *Agent* family label, which denotes mainly Trojans, Viruses, and Worms. This misclassification likely occurs since the label itself is not discriminative enough—targeting three different types of higher-level families.

C. RQ3: Obfuscation Resiliency

Malware can avoid detection by using evasive techniques that obfuscate malicious behaviors. Previous work has shown that 10 commercial antivirus products are unable to detect Android malware after simple transformations are applied to obfuscate such malware [37], [38]. To assess RevealDroid's resiliency to obfuscations, we transformed existing malware using *DroidChameleon* [37], [38], a tool suite capable of automatically transforming malware in a variety of ways. DroidChameleon transformations have previously been shown to prevent 10 commercial antivirus products from detecting the resulting transformed apps [38]. Another alternative obfuscation tool for Android we considered is ADAM [55]. However, DroidChameleon provides a wider variety of obfuscations, has composite transformations, and has demonstrated the ability to completely evade anti-malware detection. We selected apps from the original AMG to assess RevealDroid's obfuscation resiliency. Using AMG allows us to assess both the malware detection and family identification abilities of RevealDroid for obfuscation resiliency. Specifically, we evaluated different subsets of RevealDroid's three types of features for their obfuscation resiliency.

Table IX depicts the *sets of transformations* we applied: *call indirection*, where a method invocation is moved into a

TABLE IX: Sets of transformations attempted or applied

Trans. Set	Call Indirection	Rename Classes	Encrypt Arrays	Encrypt Strings
ts0	X	X	X	X
ts1	X	X	X	
ts2	X	X		
ts3	X			

new method which, in turn, is invoked in place of the original method; *renaming of classes*, where the identifier of classes is changed, which may prevent detection or family identification that searches for specific class names; and *encrypting arrays* and *strings* if they are used by an app. We selected these transformations because they have been shown to evade anti-virus products [38], can be combined to produce stronger obfuscations, and actually result in apps that are still usable. We manually tested several malicious apps, after applying transformations, to verify that the obfuscations resulted in runnable, usable apps.

We attempted to utilize another set of transformations from DroidChameleon: reordering code, which reorders instructions in the methods of a program, and a reflection transformation, which converts direct method invocations to ones that leverage the Java reflection API. We found these transformations to result in apps that crash or have errors that Soot catches, both of which never occurred before the transformations were applied. We also discovered that the code reordering often transforms a method such that it skips much of its original functionality. Consequently, we could not include these transformations in our analysis.

For each app, we first attempted to obfuscate it using all transformations. Each time a set of transformations could not be applied by DroidChameleon, we removed one transformation from the set. Previous work on Android obfuscation has demonstrated that some transformations cannot be applied together to the same app [30]. Therefore, we needed to apply transformations in different combinations to assess if they are feasible. Consequently, we attempted to transform apps in the following sequence $(ts0, ts1, ts2, ts3)$. Using this scheme for our selected set of malware apps from AMG, we successfully applied transformation set $ts0$ to 969 apps, transformation set $ts1$ to a single app, and transformation set $ts3$ to 231 apps. No apps could be successfully transformed using transformation set $ts2$. In total, these transformations resulted in 1,201 obfuscated malware samples for this experiment.

We split our app dataset using two different training strategies to assess RevealDroid for obfuscation resiliency. For the first strategy, we trained classifiers on a dataset that contains the original apps and then tested those classifiers on the obfuscated versions of those apps. This strategy has been leveraged by previous work [53]. For the second strategy, we refrain from training classifiers on any apps that we transformed (i.e., original, malicious apps before obfuscation); however, we test on the transformed apps. The second strategy raises the standard of obfuscation resiliency compared to the standard used in previous work. For the second strategy, the classifier must be able to detect and identify the family of malicious apps that are (1) obfuscated and (2) never seen before by a RevealDroid classifier. For example, this kind of strategy simulates the case

TABLE X: Detection rates for obfuscated, malicious apps

		Detection		Family Identification	
Trained on Original	Features	C4.5	1NN	C4.5	1NN
Yes	SAPI	97.42%	99.00%	92.84%	99.50%
	SAPI, IA	97.98%	93.11%	74.79%	72.52%
	SAPI, PAPI	99.50%	100.00%	96.67%	99.67%
	PAPI	98.92%	100.00%	97.17%	99.67%
No	SAPI	93.42%	92.01%	83.22%	86.44%
	SAPI, IA	92.35%	87.40%	75.17%	68.66%
	SAPI, PAPI	92.01%	94.09%	88.81%	87.63%
	PAPI	92.76%	92.51%	87.12%	87.63%

where a malicious app is previously packaged as a game—but is later packaged instead as an app for downloading wallpaper, given a few new malicious functionalities, and is finally obfuscated. Note that an overwhelming majority of apps in AMG are repackaged, making the previous example particularly relevant. Consequently, a classifier must truly learn the malicious behavior in such a situation, and ignore irrelevant features (e.g., features that are obfuscated). Overall, the second strategy gives us a clearer idea about RevealDroid's ability to generalize its detection and family identification while still maintaining accuracy in the face of obfuscation.

Table X showcases the *Detection* and *Family Identification* rates for the two most accurate classifiers (*C4.5* and *1NN*) produced by RevealDroid for a combination of *Features*. The top section of the table depicts the results for classifiers *Trained on* the *Original* malicious apps that are transformed for obfuscation, i.e., the first training strategy explained above. The bottom section of the table reports the results for classifiers that are *No*t trained on the original apps that are transformed for obfuscation, i.e., the second training strategy explained above. Gray cells indicate the highest detection or family-identification rate for a combination of features and a training strategy.

Detection. For the first training strategy, the detection rate is very high for all combinations of features and classifiers—ranging from 93% to 100% (top-left region of Table X). The combination of sensitive API and package API features, and package API features alone, when utilized with a 1NN RevealDroid classifier obtains a perfect detection rate. Two other combinations of features and classifiers obtain a near perfection (99%) detection rate: sensitive API features used with a 1NN classifier, or sensitive API features when used in tandem with package API features and a C4.5 classifier.

For the second training strategy, the detection rate remains high for all combinations of features and classifiers, ranging from 87% to 94% (bottom-left region of Table X). Thus, the lack of training on the original apps, prior to our addition of automated transformations, reduced the overall accuracy only very slightly. Similar to the first strategy, the C4.5 classifier with sensitive API and package API features obtain the highest detection rate. Unlike the first strategy, C4.5 classifiers tend to slightly outperform 1NN classifiers.

As can be observed in Table X, Intent action features slightly reduce the detection rate for obfuscated apps, unlike for non-

obfuscated detection (see Section V-A). This result indicates that Intent action features are sensitive to obfuscations. For that reason, we only show one combination of Intent action features for our obfuscation-resilience evaluation. We will examine the affect of obfuscations on Intent action features more below in the case of family identification.

Family Identification. For family identification using the first training strategy, RevealDroid's classifiers achieve 73%-99% correct classification rates (top-right region of Table X). In particular, combinations of sensitive API and package API features achieve the greatest obfuscation resiliency—from 93%-99%. Combinations involving package API features and 1NN classifiers achieve the highest accuracy at 99%.

In the case of family identification, the effect of obfuscation on different features varies widely. Intent action features—which have already shown slight evidence for a lack of obfuscation resilience for detection—have a generally negative effect on the correct classification rate for both training strategies. We took a closer look at the results and determined that Intent action features are not necessarily referenced using the Android API directly, but instead are hard-coded as strings. As a result, the encrypt strings transformations can obfuscate Intent action features.

Classification rates for the second training strategy range from 69%-89% (bottom-right region of Table X). Without utilizing Intent action features, the classification rates for that training strategy range from 83%-89%. The C4.5 classifier with sensitive and package API features achieves the highest classification rate in this scenario at 89%. Similar to detection rates, the second training strategy results in very slightly lower classification rates.

Summary. Sensitive API combined with package features exhibit high obfuscation resiliency for both training strategies—with rates between 88% and 100%. By extension, these combinations of features exhibit the most obfuscation resiliency for both detection and family identification. Nevertheless, either sensitive API or package API features individually achieve very high results. Overall, these results indicate that, with lightweight features, both detection and family identification of Android malware can be accurately performed, even for malware obfuscated with a variety of transformations.

D. RQ4: Run-Time Efficiency

The number of both benign and malicious Android apps is growing very quickly [14] making it is increasingly important that Android malware analysis scales so that such malware does not remain undetected long enough to do major damage, or even any damage. A slow analysis of Android apps can allow malware to propagate undetected longer. Furthermore, an efficient analysis of malware apps is particularly beneficial for Android end users, since they can protect themselves further by using RevealDroid's classifiers and extractors on their Android devices. Note that this device-level detection and family identification is particularly useful since Android markets have relatively poor vetting processes [58], [23].

To assess RevealDroid's efficiency, we measured run-times for both (1) feature extraction and (2) classifier training and testing. Note that once a classifier is trained, classifying an app using it—whether for malware detection or family identification—is practically instantaneous. Consequently, feature extraction and classifier training are the key bottlenecks for machine learning-based malware detection and family identification.

General Feature-Extraction. To determine RevealDroid's general run-time efficiency for extracting features, we selected 100 apps randomly from our dataset. We then ran our three types of feature extractors on each app. Such an experiment allows us to assess the general run-time efficiency of each type of feature. For this experiment, we ran our analyses on CC2 on a 64-core node.

TABLE XII: Average feature-extraction times for each type of RevealDroid feature in seconds.

	SAPI	IA	PAPI
Average (s)	15.67	7.84	14.98

Table XII depicts the results of our general feature-extraction efficiency analysis. Each feature takes under 16 seconds on average to compute. Furthermore, RevealDroid is designed to extract features in parallel, making the total feature extraction, on average, also under 16 seconds. These run-times are reasonable for practical malware detection and family identification that is obfuscation-resilient and accurate.

Information-Flow Comparison. To assess the efficiency improvement of RevealDroid's feature extraction over a state-of-the-research detection approach, we compare RevealDroid's efficiency extraction against MUDFLOW's feature extraction, which is a state-of-the-art machine learning-based approach for Android malware detection. MUDFLOW utilizes flow features alone. Another state-of-the-art approach for malware detection and family identification, DroidSIFT [53], also relies on flow-based features and machine learning. However, the tool is unavailable, preventing us from comparing against it. Due to the use of flow features in multiple state-of-the-art learning-based malware detection approaches [53], [20], [54], flow features are important to compare against for efficiency, since they require advanced program analysis, as opposed to RevealDroid's lightweight features. MUDFLOW extracts flow features utilizing, FlowDroid [19], a tool for accurately detecting information flows that are potential data leaks in an Android app. We determine the performance of RevealDroid's and MUDFLOW's feature extraction by computing the runtime for extracting their features from a selection of apps.

To ensure a fair comparison for our efficiency analysis, we utilized CC1 and selected nodes with the same hardware configuration. This experimental design prevents bias that may occur due to the varied compute nodes available on the computing clusters we used. However, due to fair scheduling algorithms utilized on CC1, we ended up using significantly slower nodes than those available on CC2, with only 8-cores on a node, resulting in higher run-time averages for RevealDroid's three feature types.

We configured FlowDroid in order to maximize performance and achieve the fairest efficiency comparison possible. For alias analyses, we set FlowDroid to be flow-insensitive. We disabled tracking of static fields and emulation of Android callbacks.

TABLE XI: Efficiency analysis of selected apps for feature extraction

				Extraction Runtime (s)			
Name or Hash	**Description**	**Rep.**	**Size**	**SAPI**	**IA**	**PAPI**	**Flow**
com.socialnmobile.hd.flashlight	flashlight app	B	1.3MB	29.18	5.72	54.93	280.04
com.netqin.mobileguard	system optimizer	B	2.6MB	60.42	11.70	61.49	446.66
org.sufficientlysecure.keychain_27000	file and communcation encryption	B	3.5MB	58.98	14.08	89.54	1503.82
com.opentable	restaurant reservations	B	4.5MB	132.97	19.60	120.51	2550.36
com.yahoo.mobile.client.android.atom	Yahoo news reader	B	5.7MB	48.42	12.37	87.62	1672.98
com.twitter.android	Twitter app	B	15MB	91.86	23.88	173.74	4464.50
fc8012f0f79d44c930449a4725a106a1	from the PJApps family	M	634KB	11.30	4.03	21.63	936.74
a85446e62ea283542653b6d7599d2e8f	adware and information stealer	M	574KB	32.70	4.98	37.68	458.36
7316dcd5c397ac0644a5a41eaae9db05	trojan and information stealer	M	692KB	33.21	4.65	39.86	35782.35
a3110b41d078d60979f147342c88a6d0	adware, information stealing	M	1.3MB	22.03	4.16	30.22	417.67
83b960675682705f94464fd7e26def55	adware and information stealer	M	985KB	10.91	3.56	23.24	1199.36
030b481d0f1014efa6f730bf4fcaff3d4b4c85ac	from the PJApps family	M	3.1MB	33.99	5.22	34.77	1642.97
da58fdfc0042315ab3393904ec602c6115d240a5	from the PJApps family	M	634KB	17.19	3.72	17.55	926.58
207fd9f3619ee825d38cf5e48efc3522e42a9c83	from the DroidKungFu3 family	M	392KB	11.90	2.70	14.17	278.00
0274a66cd43a39151c39b6c940cf99b459344e3a	from the DogWars family	M	4.3MB	13.27	2.79	13.24	259.00
d1643fb08bbb8bf5759c73cdb4ea98800700950c	from the GingerMaster family	M	199KB	9.12	2.44	10.55	588.00
f8c6d33e8dbd2172654bae104a484fcd80cf22ba	from the BaseBridge family	M	1.1MB	19.54	3.57	21.55	440.60
			AVG	**37.47**	**7.60**	**50.13**	**3167.53**

We do not compute exact propagation paths for FlowDroid, since this setting does not affect flow feature accuracy, but is expensive to compute. We set FlowDroid's layout mode to none, preventing analysis of GUI elements (e.g., input fields). Lastly, the access paths propagated by FlowDroid's taint analysis is set to 1. This setting specifies that fields of objects (e.g., $o.f$) are propagated, where o is an object and f is a field; however, no fields of fields are propagated (e.g., $o.f.g$).

For our runtime analysis, Table XI shows the apps we selected including their *Reputation* as either *B*enign or *M*alicious, their *Size* in KB or MB, a short *Description* of each app, and the name of the benign apps or the hashes of malicious apps. We selected 6 benign apps, 5 malicious apps from VirusShare, and 6 malicious apps from Malware Genome. Due to the potentially long run-times of flow-based features, we selected a smaller number of apps for efficiency analysis than in our previous analysis. Our selection of apps vary across several dimensions, allowing us to draw broader lessens about RevealDroid's and MUDFLOW's feature extraction efficiency. Focusing on a selection of apps and their dimensions further allows us to understand how the feature-extraction times vary based on the different dimensions. A gray cell for flow extraction indicates an analysis of an app that ran out of memory before feature extraction completed. In such a case, we show the runtime up until the out-of-memory error occurred.

Flow features, which MUDFLOW leverages, took the longest to run with an average runtime of 53 minutes—with one malware sample taking almost 10 hours to run before the analysis ran out of memory. This lack of scalability for flow extraction is consistent with previous findings [20].

The other features could all be extracted, on average, under a minute. Sensitive API feature extraction ran from 9 seconds to 133 seconds. Package API feature extraction ran from 10 seconds to 174 seconds. Package API features likely take longer to extract than sensitive API features simply because there are more Android API packages than security-sensitive categories. Intent action features were the fastest to analyze, taking on average under 8 seconds.

Sensitive API features alone obtain high accuracy in general. However, there is a greater than 85 times speedup from extracting sensitive API features instead of flow features. Recall that some apps (marked with gray cells in Table XI) actually take more memory and time to run than we were able to allocate even on a high performance computing cluster.

To obtain higher accuracy for detection or family identification, sensitive API and package API features can be used together—which results in an over 36 times speedup compared to flow feature extraction, even when both features are extracted serially. Combining sensitive API, package API, and Intent action feature extraction times together still achieves a 33 times speedup compared to flow feature extraction, even when all features are extracted serially, as opposed to in parallel.

Significant time savings are gained from using combinations of features not involving data flow. For feature combinations that exhibited high accuracy, a 33-85 times speedup for feature extraction is achievable compared to flow feature extraction, allowing thousands of more apps to be analyzed in the same amount of time.

Consequently, the accuracy results from the previous sections combined with our efficiency results indicate that very high accuracy can be obtained without computing flow features, i.e., MUDFLOW's most accurate features. In the next section, we will demonstrate that RevealDroid can achieve higher accuracy than MUDFLOW, even with computationally inexpensive feature extraction.

Feature Extraction and Classification. Another bottleneck for learning-based malware detection and family identification is the time it takes for a supervised-learning algorithm to train a classifier and, subsequently, test it. In a practical setting, classifiers need to be regularly updated and re-trained in order to maximize the possibility that such a classifier detects new Android malware.

Table XIII depicts execution times, in hours, for both feature extraction and classification on 9,731 apps. For this experiment, we compared RevealDroid's C4.5 (*RD-C4.5*) and 1NN (*RD-1NN*) classifiers with Adagio's and MUDFLOW's classifiers. Each approach was run on CC2, using the same hardware configuration. MUDFLOW took approximately 46 days to execute. RevealDroid's classifiers take less than two

TABLE XIII: Feature extraction and classification run-times in hours on over 9,731 apps

	RD-C4.5	RD-1NN	Adagio	MUDFLOW
Feature Extraction	42.36	42.36	56.12	1101.28
Classification	0.02	0.12	21.59	0.20
Total	42.37	42.48	77.70	1101.48

days to execute, with Adagio taking more than three days. Overall, RevealDroid is about 1.83 times faster than Adagio. Consequently, RevealDroid's classifiers are scalable compared to other learning-based malware-detection approaches.

E. RQ5: Detection Comparison

Research-Prototype Comparison. To determine RevealDroid's accuracy improvement over the state-of-the-research in Android malware detection, we compared it against two research prototypes, MUDFLOW and Adagio [26]. Besides MUDFLOW, we attempted to obtain state-of-the-research tools, DroidSIFT and Drebin [18], by contacting their respective authors. Drebin is another machine learning-based Android malware detection approach. Unfortunately, both tools are unavailable, preventing us from comparing against them directly. However, in the place of Drebin, its authors suggested we use their other tool, Adagio, which achieves similar accuracy and efficiency results, and also utilizes machine learning. We downloaded MUDFLOW and consulted with its authors to verify that we are using their implementation correctly by re-running MUDFLOW to replicate their results on their original dataset. We further computed method-level flows from FlowDroid as described in Section V-D, used those flows as inputs to MUDFLOW, and verified that we can replicate the high accuracy results from MUDFLOW's original study on a subset of apps from their dataset. We performed a similar verification in the case of Adagio.

We compared Adagio, MUDFLOW, and RevealDroid in the following two scenarios: one involving only the original untransformed apps, and another involving apps transformed using DroidChameleon, as described in Section V-C. In the scenario with no transformed apps, we split a dataset consisting of 7,801 malicious apps and 1,747 benign apps into a training set that has half of the benign apps and half of the malicious apps, while the testing set has the remaining apps. For the other scenario, the training set consists of 6,827 malicious apps and 876 benign apps; the testing set contains (1) 1,186 malicious AMG apps obfuscated as described in Section V-C and (2) the remaining 866 benign apps.

For classifier selection, we used the most accurate classifiers of MUDFLOW and Adagio. For RevealDroid, we selected a C4.5 classifier that uses sensitive API features. This selection of a RevealDroid classifier is obfuscation resilient and highly efficient, but not necessarily the most accurate, as demonstrated in the previous sections. However, our results will demonstrate that it still outperforms MUDFLOW and Adagio.

Table XIV showcases the *Prec*ision, *Rec*all, and *F-M*easure results for each approach and both scenarios (*No Obf*uscations and *With Obf*uscations). For each of those metrics, the table depicts results for *Ben*ign apps and *Mal*icious ones. Overall, RevealDroid's classifier outperforms MUDFLOW's two-way classifier. In the scenario with no obfuscations, RevealDroid obtains an average F-Measure of 87% compared to MUDFLOW's 71%. For the scenario with obfuscated apps, RevealDroid obtains an average F-measure of 87% compared to 74%.

The most striking difference between MUDFLOW's and RevealDroid's results for both scenarios is each classifier's recall for benign apps. In the scenario with obfuscations, RevealDroid achieves a 77% recall for benign apps compared to MUDFLOW's 47%. For benign apps in the other scenario, RevealDroid obtains a 73% recall compared to MUDFLOW's 49% recall. These results indicate that MUDFLOW's classifier has a strong tendency to mark benign apps as malicious, unlike RevealDroid's classifier.

Adagio obtains 3% higher F-Measure results than RevealDroid in the scenario with no DroidChameleon obfuscation. However, with the DroidChameleon obfuscations, RevealDroid significantly outperforms Adagio by 25%. In fact, Adagio is the least obfuscation-resilient approach of the three that we evaluated. This result is expected, particularly due to the use of call-indirection transformations, which changes the expected call graph that Adagio utilizes to identify malware.

In summary, RevealDroid obtains obfuscation resiliency and accuracy for detection, as compared to two state-of-the-research malware detection approaches.

Anti-Virus Comparison. In addition to comparing against research prototypes, we assess RevealDroid's detection rate against commercial anti-virus (AV) products available on VirusTotal. This experiment allows us to compare RevealDroid against the state-of-the-practice, rather than just the state-of-the-research. To that end, we conducted two experiments for this AV comparison: one with 6,766 apps from VirusShare with no additional obfuscations from DroidChameleon, and another with 1,200 Android Malware Genome apps with such obfuscations. We utilized RevealDroid's most accurate classifiers for our comparisons. Among the 60 commercial AVs available on VirusTotal, we depict our results for 12 common AVs. However, none of the AVs obtained higher detection rates, for either experiment, than RevealDroid. Furthermore, RevealDroid learns to identify malware automatically, while AVs typically need manually-produced signatures.

Figure 2 shows the results for the first experiment. RevealDroid is capable of achieving a very high detection rate (94%), which is slightly greater than the highest rate from a commercial AV, ESET-NOD32 at 93%.

Figure 3 illustrates the results for the second experiment. In this experiment, RevealDroid achieves a near-perfect detection rate at 99%, similar to the highest AV, Kasperky—also at 99%.

Across both experiments, only a few AVs achieve above 90% detection rates: AVG, ESET-NOD32, Kasperky, and McAfee. Only RevealDroid and ESET-NOD32 achieve over 90% detection rates for both experiments.

Overall, these experiments demonstrate that RevealDroid achieves as high of a detection rate, or higher, as the best AV tools. However, RevealDroid can learn to identify new malware automatically as new malware samples become available, while achieving obfuscation resiliency, and efficient feature extraction and classifier construction.

TABLE XIV: Research-Prototype Detection Comparison

	MUDFLOW						RevealDroid						Adagio					
	No Obfuscations			With Obfuscations			No Obfuscations			With Obfuscations			No Obfuscations			With Obfuscations		
	Prec	**Rec**	**F-M**	**Prec**	**Rec**	**F-M**	**Prec**	**Rec**	**F-M**	**Prec**	**Rec**	**F-M**	**Prec**	**Rec**	**F-M**	**Prec**	**Rec**	**F-M**
Ben	85.14%	34.17%	48.77%	98.09%	47.46%	63.97%	84.30%	73.20%	78.36%	93.30%	77.30%	84.55%	89.96%	76.29%	82.57%	53.55%	73.21%	61.85%
Mal	87.29%	98.70%	92.65%	72.14%	99.33%	83.58%	94.30%	97.00%	95.63%	85.20%	96.00%	90.28%	94.89%	98.10%	96.47%	73.18%	53.51%	61.82%
AVG	**86.22%**	**66.44%**	**70.71%**	**85.11%**	**73.39%**	**73.77%**	**89.30%**	**85.10%**	**86.99%**	**89.25%**	**86.65%**	**87.41%**	**92.43%**	**87.20%**	**89.52%**	**63.36%**	**63.36%**	**61.84%**

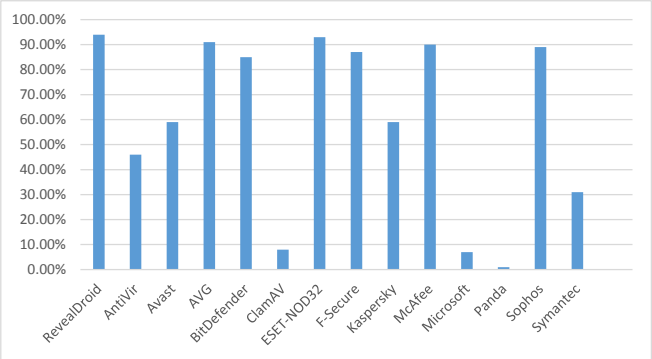

Fig. 2: Detection rates for RevealDroid and anti-virus products on 6,776 apps from VirusShare

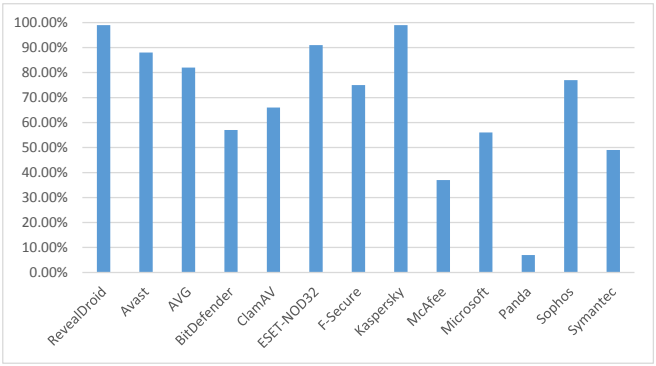

Fig. 3: Detection rates for RevealDroid and AV products on 1,200 AMG apps obfuscated using DroidChameleon

F. RQ6: Family-Identification Comparison

To demonstrate the improvement in accuracy of RevealDroid's family identification over the state-of-the-art, we compare RevealDroid against a state-of-the-art Android-malware family-identification approach, Dendroid [41], which also utilizes machine learning to classify malware. Dendroid uses features that represent each method of an app as a sequence of typed statements. We contacted the authors of another approach, DroidSIFT [53], which also identifies families. However, DroidSIFT's authors are unable to share their implementation. Consequently, we could not compare against DroidSIFT. Note that neither MUDFLOW nor Adagio perform family identification.

We closely consulted with the authors of Dendroid to ensure we obtain the most accurate results using their tool as possible. To that end, we replicated their evaluation and verified the accuracy of our results with Dendroid's authors. To compare Dendroid and RevealDroid, we assessed both approaches using AMG. Specifically, we split AMG apps into a training and testing set of approximately equal size using the second training strategy from Section V-C. Given that 13 families in AMG only have a single sample, we selected families which had at least two samples, resulting in 33 families in total. For each family, half of the samples were placed into the test set and half into the training set. For families with odd-numbered samples, the remaining sample was added to the training set. This splitting strategy resulted in a training set of 626 apps and a testing set of 607 apps. For RevealDroid, we selected its 1NN classifier with sensitive API and package API features since it demonstrated high accuracy in our earlier experiments (see Section V-C).

Using that experimental setup, Dendroid correctly classified 73% of the test apps, while RevealDroid achieves an 87% correct classification rate. Although our replicated results for Dendroid are significantly lower than the Dendroid author's original results [41], we verified our results with those authors and discovered an error in their experiment.

We further compared RevealDroid's and Dendroid's obfuscation resiliency. To that end, we trained both Dendroid and RevealDroid using the training set consisting of half of AMG. We then replaced apps in the test set with their obfuscated versions—transformed as discussed in Section V-C. The resulting test set contains 590 apps.

RevealDroid demonstrated overwhelmingly greater obfuscation resiliency than Dendroid: RevealDroid maintains an 87% correct classification rate, while Dendroid's classification rate falls to 27%. This low result for Dendroid is unsurprising since it relies on the structure of a method as features. Given that the call indirection transformation that we applied to the test apps alters that structure, the transformation prevents proper classification by Dendroid.

G. Discussion and Limitations

One of the major goals of RevealDroid is to aid in the selection of features that are obfuscation-resilient, highly accurate, and highly efficient. Our results demonstrate that these three qualities are best achieved, in tandem, using sensitive API features and package API features, with either a C4.5 or 1NN RevealDroid classifier.

Although Intent actions tend to improve detection and family-identification results, they are, unfortunately, not obfuscation-resilient. However, for malware that a RevealDroid classifier has already trained on, that classifier is likely to learn the malicious properties of the app, despite pre-existing obfuscations. In this case, Intent action features become particularly useful for distinguishing between malware families, reducing the time needed to assess the damage an Android malware sample is capable of.

One possible obfuscation that RevealDroid may be susceptible to, is the addition of fake API calls, i.e., calls that do

not affect the semantics of the app, but may potentially affect the features learned by RevealDroid classifiers. We have not observed such obfuscations in the wild or through automated transformations of Android apps. It is unclear the extent to which such API calls can be injected into a program without actually affecting it's semantics. Adding too many API calls will either make the program look highly suspicious, possibly flagging it as malware, or break its semantics. Furthermore, adding API calls that are not actually invoked at run-time (e.g., using non-feasible conditionals), is detectable through static analysis (e.g., through analyses that find conditionals that always evaluate to false).

Nevertheless, we assessed RevealDroid's ability to deal with such obfuscations. We simulated the addition of such fake API calls by randomly adding such calls to the features of thousands of malware samples in our dataset. Adding fake API calls in the dozens only reduced RevealDroid's classifier results by a few percentage points. This is a reasonable number of obfuscations to include, otherwise, the app may break or no longer achieve the same semantics. Any higher number of calls was perfectly detectable—achieving 100% detection rates—by using a Support Vector Machine (SVM) [40] classifier. This result is sensible since apps with an unusually high number of APIs act as outliers and resemble abnormal behaviors that are potentially malicious, and thus marked by an SVM classifier as such. However, such classifiers are considerably slower than C4.5 and 1NN classifiers, taking up to days more to train and test on our datasets.

Limitations of the dataset utilized by RevealDroid represents a threat to external validity. However, we carefully selected apps to maximize the probability that they are correctly marked as benign or malicious (see Section IV). We further utilized family labels already verified by security experts (see Section IV and Section V-B). Moreover, machine-learning algorithms themselves are partially self-corrective, through statistical methods, for errors in the datasets. Furthermore, malware that minimizes use of Android APIs or leverages mechanisms such as reflection, native code, or dynamic class loading may not be properly classified by RevealDroid. However, some apps (e.g., those in the Android Malware Genome) already include such mechanisms. Including package API features, already include representations of some of these features (e.g., dynamic class loading). Nevertheless, extracting features that represent novel malware behaviors that can be identified through machine learning remains as interesting future work.

VI. Related Work

We provide an overview of the current state of Android malware detection and family identification. We first discuss the techniques that solely aim to detect malicious Android apps. We then cover signature-based and machine learning-based techniques that aim to identify the family of such apps.

Some techniques detect Android malware by focusing on specific risk factors. RiskRanker [28] ranks apps as either high-risk, medium-risk, or low-risk in order to identify malware. Peng et al. [33] perform risk ranking and scoring by leveraging probabilistic generative models to identify malware apps. MAST [22] ranks apps according to their suspiciousness using a social-sciences technique.

AppAudit [47] is a recent approach that combines static and dynamic analysis to identify apps that leak data. AppIntent [51] is another approach for determining data leaks that focuses on identifying whether the leak was intended by the user.

Other techniques utilize virtualization or sandboxes to aid in the detection of Android malware. DroidScope [49] is a virtualization-based malware analysis engine that utilizes different dynamic analyses to monitor malware. CopperDroid [42], [39] is an approach for reconstructing Android-malware behaviors through virtualization, a focus on system calls, automatic IPC unmarshalling, and value-based data-flow analysis. A5 [45] is an open-source sandbox for Android that aims to trigger malicious behaviors.

Machine learning has been used for simply distinguishing between benign and malicious Android apps. MUDFLOW [20] uses information flow-based features and machine learning to distinguish benign and malicious apps. DroidMat [46] distinguishes between benign and malicious apps through various features extracted using static analysis and clustering. Furthermore, it relies on easily obfuscatable features (e.g., names of component classes). We contacted the authors of DroidMat multiple times to obtain its implementation so that we can compare against it. However, none of the authors ever responded to our queries.

Drebin [18] is designed to detect Android malware directly on an Android device and uses machine learning. Drebin also uses pre-defined templates to display potentially useful information about what makes an app malicious. Unlike RevealDroid, Drebin relies heavily on features based on constant strings (e.g., names of components) that are obfuscatable using basic automated transformations (e.g., renaming and encrypting identifiers and string values). Furthermore, their feature space is very large, containing about 545,000 features, as compared to RevealDroid's feature space of 219 features, which allows our classification—and potentially our feature extraction—to be significantly more efficient and scalable. Unfortunately, Drebin is unavailable, so we could not use it in our experiments.

Certain techniques leverage Android-app permissions to identify malware apps. Kirin [24] certifies an Android app against a set of rules to determine if the app may perform malicious behavior. DroidRanger [58] attempts to identify malware based on the permissions and behaviors of an app.

ViewDroid [52] and MassVet [23] are capable of detecting malicious Android apps and both focus on repackaging detection. Both techniques leverage graphs based on UI widgets of an Android app. Due to the use of control flow-based graphs, both of these techniques are potentially susceptible to control flow-based obfuscations. RevealDroid is not vulnerable to such obfuscations, due to the fact that it does not rely on any program-analysis graph representations. Unlike in the case of RevealDroid, no automated transformations were applied to existing malicious apps to assess MassVet; automated transformations were applied to benign apps for MassVet. However, as discussed in the MassVet paper [23], obfuscations of malicious methods may be problematic for MassVet. Additionally, whether the transformed benign apps utilized combinations of transformations was not discussed. Unlike MassVet and ViewDroid, RevealDroid is capable of accurately identifying the family to which a malware belongs,

and not just identifying an app as malicious. Furthermore, RevealDroid is not limited to only detecting and identifying families of repackaged malicious apps.

A variety of other techniques use different mechanisms for detecting Android malware. DroidAnalytics [56] provides an automated workflow for the collection and signature generation of Android malware by analyzing apps at the opcode level. AsDroid [31] detects stealthy behaviors of possibly malicious apps characterized by mismatches between program behavior and the UI. Poeplau et al. [34] construct a static analysis tool for identifying unsafe and malicious dynamic code loading.

Besides not identifying malware families, most of the above techniques rely on heavyweight program analysis, unlike RevealDroid's lightweight analysis.

Several approaches focus on identifying specific malware families. Apposcopy [25] provides a language to specify malware signatures and a static analysis to identify apps matching those signatures. For Apposcopy, security engineers must manually construct malware signatures, which is a time-consuming and error-prone task.

A few approaches automatically identify the family of Android malware. Dendroid [41] utilizes text-mining techniques and control-flow features to identify families of malicious apps. DroidSIFT [53] employs extracted dependency graphs to determine whether an app is benign or malicious, and the family of a malicious app.

The two approaches that automatically identify the family of Android malware—Dendroid and DroidSIFT—are both limited, when compared to RevealDroid, in two key ways: (1) they use a highly outdated malware dataset; and (2) they perform a highly limited assessment for obfuscation resiliency, or no such assessment at all. Both approaches are evaluated on a limited number of malware families and apps, and use malware datasets that are antiquated, dating back to 2011. On the other hand, we evaluate RevealDroid on a dataset consisting of thousands of more apps discovered up until early 2014, and nearly double the malware families studied as part of the DroidSIFT paper. Additionally, DroidSIFT utilizes flow features, which are heavyweight to extract, as demonstrated in our experiments.

Both techniques have limited obfuscation resiliency, and rely on representations (e.g., control-flow features or constant strings) that can be evaded by using standard automated transformations. Furthermore, DroidSIFT is only assessed using unstated obfuscations applied to a small number of apps from a single malware family.

VII. Conclusion

This paper has introduced RevealDroid, a machine learning-based approach for Android malware detection and family identification that is accurate, efficient, and obfuscation resilient. We have compared RevealDroid with state-of-the-research and state-of-the-practice tools for Android malware detection. Our experiments showcase RevealDroid's superior accuracy and efficiency, particularly under various obfuscations. We further compared RevealDroid to a state-of-the-research family-identification approach, demonstrating significantly higher accuracy, especially in the face of obfuscations.

In the future, we intend to explore feature characteristics of emerging malware apps—such as those that infect an Android device's Master Boot Record [11] and stealthily utilizing devices to mine cryptocurrency services [2]—in order to detect and identify the families of those malware. Additionally, we further intend to explore lightweight feature-extraction mechanisms to classify malware that leverages native code, dynamic class loading, or reflection.

To enable replication of our results and improvement over RevealDroid, we make our RevealDroid prototype and data available online at [8].

References

[1] Android trojan looks, acts like windows malware. http://www.snoopwall.com/android-trojan-looks-acts-like-windows-malware/.

[2] Bitcoin-mining malware reportedly found on google play. http://www.cnet.com/news/bitcoin-mining-malware-reportedly-discovered-at-google-play/.

[3] <blinded title for CC1>. <blinded URL>.

[4] <blinded title for CC2>. <blinded URL>.

[5] Cisco 2014 annual security report. http://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html.

[6] F-droid. https://f-droid.org/.

[7] Google play market. http://play.google.com/store/apps/.

[8] RevealDroid. http://tiny.cc/revealdroid.

[9] Server-side polymorphic android applications. http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications.

[10] The Drebin Dataset. http://user.informatik.uni-goettingen.de/ darp/-drebin/.

[11] Threat description trojan:android/oldboot.a. https://www.f-secure.com/v-descs/trojan_android_oldboot_a.shtml.

[12] VirusShare.com. http://www.virusshare.com/.

[13] VirusTotal. https://www.virustotal.com/.

[14] Quick Heal Annual Threat Report 2015. http://www.quickheal.co.in/resources/threat-reports, January 2015.

[15] D. Aha and D. Kibler. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[16] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian. Analysis of malicious and benign android applications. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 608–616. IEEE, 2012.

[17] A. Apvrille and R. Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, 2014.

[18] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[20] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. *To appear in the Proceedings of the 37th International Conference on Software Engineering*, 2015.

[21] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[22] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[23] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 659–674, Washington, D.C., Aug. 2015. USENIX Association.

[24] 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*, pages 235–245. ACM, 2009.

[25] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[26] 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*, AISec '13, pages 45–54, New York, NY, USA, 2013. ACM.

[27] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035, New York, NY, USA, 2014. ACM.

[28] 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.

[29] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter*, 11(1):10–18, 2009.

[30] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*, pages 169–186. Springer, 2013.

[31] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046, New York, NY, USA, 2014. ACM.

[32] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[33] 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*, pages 241–252. ACM, 2012.

[34] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[35] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[36] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[37] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 329–334. ACM, 2013.

[38] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, Jan 2014.

[39] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *European Workshop on Systems Security (EuroSec), April*, 2013.

[40] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 1995.

[41] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[42] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[43] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[44] V. N. Vapnik and V. Vapnik. *Statistical Learning Theory*, volume 2. Wiley New York, 1998.

[45] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 39–50, New York, NY, USA, 2014. ACM.

[46] 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.

[47] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *IEEE Symposium on Security and Privacy*, 2015.

[48] E. P. Xing, M. I. Jordan, R. M. Karp, et al. Feature selection for high-dimensional genomic microarray data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, volume 1, pages 601–608. Citeseer, 2001.

[49] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium*, pages 569–584, 2012.

[50] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, volume 97, pages 412–420, 1997.

[51] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[52] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

[53] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[54] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.

[55] M. Zheng, P. P. Lee, and J. C. Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer, 2013.

[56] M. Zheng, M. Sun, and J. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.

[57] 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.

[58] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2012.