Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

# Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware

Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek
{jgarci40, mhammad2, bpedrood, abagheri, smalek}@gmu.edu

Technical Report GMU-CS-TR-2015-10

## 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. Techniques for detecting Android malware, and determining their families, lack the ability to deal with obfuscations (i.e., transformations of application to thwart detection). Moreover, some of the prior techniques are highly inefficient, making them inapplicable for real-time detection of threats. To address these limitations, we present a novel machine learning-based Android malware detection and family identification approach, RevealDroid, that provides selectable features. We assess RevealDroid to determine a selection of features that enable obfuscation resiliency, efficiency, and accuracy for detection and family identification. 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, and compare RevealDroid against an existing Android malware-family identification approach and another Android malware detection approach.

## 1 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 [4, 2], 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 [21, 40, 45, 24, 23, 28, 43, 34, 38, 14, 26] has focused on distinguishing between benign and malware 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 [23, 19]. 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 [27]. 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 [44], DroidDream [44], Oldboot [9], 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 [8, 16, 32]. For example, *Agent.BH!tr.spy* steals information by sending emails using SMTP with TLS authentication [16], 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 [32]. 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 [35, 42]. These techniques use features that are not resilient to obfuscations (e.g., features based on control flow [35] or constant strings [42]).

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 [26, 19]. 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 [45]. Consequently, it is desirable to utilize features that can be extracted efficiently for detection and family identification of Android malware apps, even obfuscated ones.

This paper makes the following contributions:

- We introduce *RevealDroid*, a machine-learning based approach for detecting malicious Android apps and identifying their families that provides a selectable set of features for achieving different trade-offs between obfuscation resiliency, efficiency of analysis, and accuracy. RevealDroid is capable of accurately detecting malicious apps and identifying their families at above 93% for untransformed apps and above 87% for transformed apps, and can do so, on average, for an app in under a minute. 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 2,593 benign apps and 9,054 malware apps from two different malware repositories. We further compare RevealDroid's detection and family identification accuracy against state-of-the-art approaches: MUDFLOW [19], an approach for malware detection, and Dendroid [35], 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. RevealDroid achieves a 14%-60% higher classification rate than Dendroid.

- We construct an updated dataset of 857 malware apps labeled with their malware families and assess RevealDroid's family identification accuracy on that dataset. We make this updated dataset available for researchers and practitioners [7].

- 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. We compare RevealDroid's accuracy for detection under obfuscation against MUDFLOW, and for family identification under obfuscation against Dendroid.

- We assess the efficiency of RevealDroid's feature extraction, which is the major bottleneck of machine learning-based techniques that detect or identify families of malware. We show that a subset of RevealDroid's features can be more than 33-85 times faster than the features utilized by MUDFLOW, while still exhibiting obfuscation resiliency and accuracy for detection and family identification.

The remainder of this paper is structured as follows. Section 2 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 3 introduces RevealDroid and its design. Section 4 covers the design and configuration for our evaluation, including the research questions we study; Section 5 discusses the evaluation results for each research question, and examines and interprets our results. Section 6 covers work related to RevealDroid. Section 7 concludes the paper and discusses possible future work.

## 2 Foundation

Malware detection and family identification can be placed into two categories: signature-based and machine learning-based [42]. 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. To ensure the highest degree of automation, we focus on learning-based methods for our Android malware detection.

## 3 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 must 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 Android malware to design an approach for classifying Android malware families.

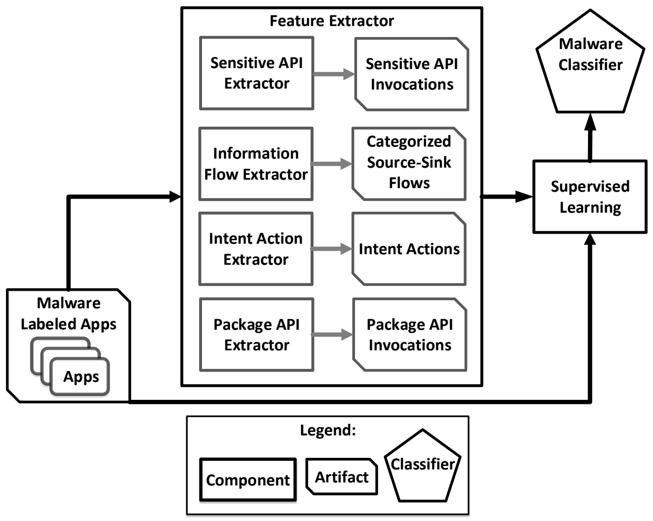

Figure 1: Overview of RevealDroid's malware classifier production

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. 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. RevealDroid allows its features to be used in different combinations, resulting in different levels of obfuscation resiliency, efficiency, and accuracy. RevealDroid contains the following four 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) data flows between Android APIs, i.e., possible information leakages—obtained by *Information Flow Extractor* in Figure 1; (3) *actions* of Android messages that an app may listen to—which is identified by *Intent Action Extractor* in Figure 1; and (4) Android API usage categorized by the package to which the API belongs, which is determined by *Package API Extractor* in Figure 1.

For each type of feature, this section explains its importance, and the manner in which the feature type is represented and extracted. The section ends by covering how apps are labeled and supervised-learning algorithms are used in RevealDroid to produce classifiers for detecting malware and identifying their families.

### 3.1 Sensitive API-Usage Extraction

Malware apps must invoke or access Android APIs in order 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 [31], 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 $i \in C$. These features are similar to and inspired by those found in [19]. However, unlike those features, RevealDroid considers a wider set of categories, including a category for GUI-widget methods and an additional category for all other method invocations that are neither sensitive nor widget-based.

To illustrate how such features can help distinguish malware families, Table 1 depicts features for a subset of categories from three Android malware families. For example, in Table 1, 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 1: 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 [37, 41, 39].

## 3.2 API Flow Extraction

Data flows between Android APIs correspond to possible information leakages. Specifically, RevealDroid must determine information flows between Android *source* API methods, capable of retrieving Android data, and *sink* API methods, which can store or send data from source methods. An example of a data flow leaking information is the flow from an API method that returns a device's IMEI, an identifier that uniquely identifies an Android device, to a message-sending method, which may send the IMEI to an entity outside the app. These features are similar to and inspired by those utilized in [19]. However, they vary in two key ways: the number of categories utilized and the level of abstraction. RevealDroid has an extra category for GUI widget-based methods, and only one category to represent features that are neither sensitive API usage nor widget-based.

A feature space for Android information leakage that straightforwardly represents the flow between API methods as features can result in over 92,000 features, due to the fact that there are over 300 source and sink API methods. This large feature space, just for a single type of feature, would cause scalability and accuracy issues for machine learning [37, 41, 39], especially since we aim to accurately assign one of many possible families to an app. To address that issue, RevealDroid uses the following feature vector for information flow $Flow_a = (f_1, ..., f_i, ..., f_{|C_{src} \otimes C_{snk}|})$, where $C_{src} \in C$ is the set of source API method categories; $C_{snk} \in C$ is the set of sink API method categories, $f_i = |\{(m_x, m_y) \bullet m_x \in C_{src} \wedge m_y \in C_{snk}\}|$, and $m_x$ and $m_y$ are respectively source and sink methods involved in an information flow within Android app $a$. Other than the widget category, which we specify as a source category, the rest of the source and sink categories are obtained from SuSi. We assign source API methods to a set of 20 categories ($|C_{src}| = 20$), and sink API methods to a set of 21 categories ($|C_{snk}| = 21$). Consequently, for these information-flow features, we only need 420 features rather than over 92,000 features, which alleviates the feature-space issue. As an example, a flow from a method that retrieves the Android device's IMEI and sends the information over SMS is represented as a flow between the UNIQUE_IDENTIFIER and SMS_MMS categories. To help a learning algorithm better distinguish between information flows of malware families, each flow feature is a count of the number of flow instances between categories. For instance, if the IMEI and SIM card ID of a device—each obtained from two different source methods—flow to SMS sink methods, then the value for the feature (UNIQUE_IDENTIFIER,SMS_MMS) is 2.

To illustrate our information-flow feature space for learning-based Android malware detection and family identification, Table 2 depicts example information-flow features for a set of real malware apps, where we elide irrelevant features for brevity. Three malware apps are depicted, each from a different malware *Fam*ily. The number of information flows between the following categories are shown for each app: *SMS*; inter-process commmunication (*IPC*); *CONT*act information; *EMAIL*; *BROWS*er information; *SYNC*hronization data; *BUNDLE*s, which contain data that can be included as part of an Intent; *NET*work; *FILE* manipulation; and *UNC*ategorized, which are Android API methods not classified into their own specific categories. For example, malware *mal3*, an instance of DroidKungFu3, has 5 flows between source API methods from the Bundle category to sink API methods of the Network category.

Table 2 demonstrates the intuition behind how a classifier can be built from these features: A learning algorithm can determine that a low value for (SMS,IPC) uniquely identifies Geinimi, a low value for (CONT,EMAIL) uniquely identifies GoldDream, and non-zero values for (BROWS,SYNC), (BUNDLE,NET), and (FILE,IPC) distinguish DroidKungFu3.

## 3.3 Intent Action Extraction

Different families of malware activate based on different *actions* of *Intents* [44], 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

Table 2: Example information-flow features from three known Android malware families

| | SMS,IPC | CONT,EMAIL | BROWS,SYNC | BUNDLE,NET | FILE,IPC | Fam |
|---|---|---|---|---|---|---|
| mal1 | 1 | 0 | 0 | 0 | 0 | Geinimi |
| mal2 | 0 | 1 | 0 | 0 | 0 | GoldDream |
| mal3 | 0 | 0 | 2 | 5 | 1 | DroidKungFu3 |

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 3: 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 3 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.

### 3.4 Package API-Usage Extraction

In situations where data flows and Intent actions are insufficient, Android API usage information is included as a feature to aid a classifier in distinguishing between malware families. These features have been shown to be useful features for distinguishing malware families when manually specifying their signatures [22]. 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.

### 3.5 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 functionalities. The classifier constructed by RevealDroid depends on the labels used when training a classifier. Furthermore, RevealDroid is designed to use different machine-learning 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 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 5.5-5.6.

## 4 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-art 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**: How efficient is RevealDroid's extraction of features compared to another state-of-the-art detection approach?

- **RQ5**: How does RevealDroid's detection accuracy compare to another state-of-the-art detection approach?
- **RQ6**: How does RevealDroid's family identification capability compare to another state-of-the-art malware-family identification approach?

We implemented RevealDroid in Java for its feature extraction, malware detection, and malware-family identification. We utilized FlowDroid [18], a technique for obtaining information flows in Android, to implement *Information Flow Extractor*. To construct the *Sensitive API Extractor*, *Intent Action Extractor*, and *API Extractor*, we leveraged Soot [36], a static analysis framework, and Dexpler [20], a translator from Android Dalvik Bytecode to Soot's intermediate representation. For machine learning, we selected Weka [25], a widely-used machine-learning toolkit for Java.

We configured FlowDroid to maximize performance by setting it as follows. Our experiences showed that RevealDroid's correctness remains high despite configuring FlowDroid for maximum performance. For alias analyses, we set FlowDroid to be flow-insensitive. We disabled tracking of static fields and emulation of Android callbacks. We do not compute exact propagation paths for FlowDroid, which are unnecessary for RevealDroid's design. 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 conducting feature extraction, we leveraged George Mason University's ARGO computing cluster [1]. 35 of Argo's compute nodes each have 8-core 2.60GHz CPUs and 64GB RAM, which are the compute nodes we utilized for our experiments.

To assess RevealDroid's accuracy, we constructed a dataset of both benign and malicious Android apps. To obtain benign apps, we downloaded 2,593 apps from two sources: *Google Play* [6], Google's official Android app repository, and *F-Droid* [5], an open-source repository of Android apps. For Google Play, we selected popular apps to increase the likelihood of them being benign. 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 two Android malware repositories: the Android *Malware Genome* project [44] and *VirusShare* [10]. Malware Genome contains over 1,200 Android malware apps from 49 different malware families. We utilized 9,054 Android malware samples from VirusShare.

## 5 Evaluation Results

For each research question, we convey its importance, specific experimental setup needed to study it, and our corresponding results. After examining each research question in detail, we discuss the overall findings and limitations of our study.

### 5.1 RQ1: Detection Accuracy

In order 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* [30] and a *1-nearest-neighbor (1NN) classifier* [13] for labeling an app as either benign or malicious. We also experimented with a few others, including *support vector machines* [15], that did not show the same level of accuracy.

Table 4 shows the correct classification rate among the different combinations of four features: API *Flow*s, sensitive APIs (*SAPI*), Intent Actions (*IA*), and package APIs (*PAPI*). The number of *Ben*ign and *Mal*icious apps vary across different experiments due to either limits on computational resources preventing timely extraction of flow features (which in some cases could take many hours to execute even on Argo cluster), or errors with Soot and FlowDroid that sometimes fail on certain Android apps. For each combination of features, classifiers, and apps, we performed a 10-fold cross-validation and report the rate of correctly classified apps. However, we do not combine flow features and sensitive API features in our study because the two features overlap: The categorized sensitive API methods serve as the source and sink methods of information flow.

Table 4: Detection results for different combinations of RevealDroid's features and classifiers.

| Features | C4.5 | 1NN | Ben | Mal |
|---|---|---|---|---|
| Flow | 87.57% | 85.23% | 1,747 | 7,800 |
| Flow, IA | 90.43% | 88.53% | 1,747 | 7,786 |
| Flow, IA, PAPI | 95.32% | 94.41% | 1,104 | 7,780 |
| SAPI | 93.88% | 92.94% | 2,593 | 10,313 |
| SAPI, IA | 94.78% | 94.02% | 2,583 | 10,283 |
| SAPI, IA, PAPI | 96.35% | 95.56% | 1,268 | 10,288 |

All combinations of features exhibit a high correct classification rate. Feature combinations with flow features have a correct classification rate between 85% and 95%. Feature combinations with sensitive API features have a correct classification rate between 93% and 96%. The addition of Intent action features and package API features increases the classification rate for flow features by 7% for C4.5 and 9% for 1NN. The addition of Intent action and package API features to sensitive API features only increases its classification rate by about 2%-3%.

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 5 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 5: Cross-validation results for the combination of sensitive API, Intent action, and package API features using a C4.5 classifier

| | Prec | Rec | F-Meas | ROC Area |
|---|---|---|---|---|
| Benign | 84.8% | 81.3% | 83.0% | 91.1% |
| Malicious | 97.7% | 98.2% | 98.0% | 91.1% |
| WAvg. | 96.3% | 96.3% | 96.3% | 91.1% |

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 96%. RevealDroid also demonstrates a high discriminatory power, as demonstrated by the 91% ROC Area for benign apps, malicious apps, and the weighted average.

## 5.2 RQ2: Family Identification

Simply identifying an Android app as malware is insufficient for dealing with the app. Once a malicious app is deployed, it may install other apps, steal information, modify settings, etc. Consequently, 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.

To determine RevealDroid's ability to classify Android malware apps into families, we assessed RQ2 by utilizing the Android Malware Genome (AMG) [44], which contains over 1200 apps and 49 malware families. To that end, we used RevealDroid to construct classifiers with up to 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 6 depicts the classification rate for the two most accurate classifiers among the different combinations of four features. As in the prior experiment, the numbers of apps (No. Apps) in Table 6 vary due to the types of features used. The increased computational resources required to extract flow features reduced the number of apps we could analyze for that type of feature. Furthermore, errors from Soot and FlowDroid further limited the number of apps from which we can extract features.

Table 6: RevealDroid's classification rate for family identification utilizing different features and classifiers on AMG

| **Features** | **C4.5** | **1NN** | **No. Apps** |
|---|---|---|---|
| Flow | 91.54% | 91.78% | 1,217 |
| Flow, IA | 94.17% | 93.43% | 1,217 |
| Flow, IA, PAPI | 95.07% | 94.66% | 1,217 |
| SAPI | 87.69% | 87.29% | 1,259 |
| SAPI, IA | 91.51% | 91.75% | 1,248 |
| SAPI, IA, PAPI | 93.62% | 92.98% | 1,253 |

Overall, the accuracy of RevealDroid's malware-family classifiers is between 87% and 95% for all combinations of features and classifiers. These results showcase RevealDroid's ability to identify a malicious app with high accuracy. Sets of features based on flows (top half of Table 6) are about 2%-3% more accurate than features based on sensitive APIs without flows (the bottom half of Table 6). This outcome indicates that our API-based features are well-chosen for discriminating between malware families.

The Intent action and package API features combined with either flow or sensitive API significantly increased the accuracy for family identification, which is difficult to do given the already high classification rate of either flow 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 97% from 51% for flow features. As another example, package API features increased the accuracy for the GPSSMSSpy family, consisting of 10 samples, from 67% to 92% for flow features.

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 [10], 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 [11], 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.

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 7 shows our results for EAMG. Just as before, the numbers of apps per combination of features vary due to limits on computational resources or errors in Soot and FlowDroid when extracting features from apps.

Similar to our previous results, RevealDroid correctly classifies 84%-94% 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 flow features improved the accuracy for the GoldDream family— consisting of 63 samples— from 67% to 88% and for the Zitmo family—consisting of 15 samples—from 67% to 90%. 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 flow features obtained up to 89% accuracy, while combinations involving sensitive API features obtained up to 91% accuracy.

The results for AMG and EAMG indicate that either combinations of flow features or combinations of sensitive API features are highly accurate for identifying malware families. At the same time, flow features tend to be slightly more accurate for family identification.

Table 7: RevealDroid's classification rate for family identification utilizing different features and classifiers on EAMG

| Features | C4.5 | 1NN | No. Apps |
|---|---|---|---|
| Flow | 89.15% | 88.94% | 1,907 |
| Flow, IA | 92.13% | 91.29% | 1,906 |
| Flow, IA, PAPI | 93.02% | 93.81% | 1,905 |
| SAPI | 84.38% | 86.11% | 2,080 |
| SAPI, IA | 89.93% | 91.09% | 2,066 |
| SAPI, IA, PAPI | 90.54% | 91.74% | 2,071 |

## 5.3 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 [32, 33]. To assess RevealDroid's resiliency to obfuscations, we transformed existing malware using *DroidChameleon* [32, 33], a tool suite capable of transforming malware in a variety of ways. 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 four types of features for their obfuscation resiliency.

Table 8 depicts the *sets of transformations* we applied: *call indirection*, where a method invocation is moved into a 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. These DroidChameleon transformations have previously been shown to prevent 10 commercial antivirus products from detecting the resulting transformed apps [33]. 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. 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, we ended up with 1,201 obfuscated malware samples for this experiment.

Table 8: Sets of transformations attempted or applied on Android malware

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

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 [42]. 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 in any form by a RevealDroid classifier. For example, this kind of strategy simulates the case 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. 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 9 showcases the *Detection* and *Family Identification* rates for the two most accurate classifiers (*C4.5* and *1NN*) produced by RevealDroid for a combination of either flow or sensitive API *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. As before, limitations in computational resources and errors from Soot or FlowDroid limit our *Train*ing or *Test Size*s—as shown in Table 9 and measured in terms of the number of apps—particularly for the more expensive to compute flows. Gray cells indicate the highest detection or family-identification rate for a combination of features and a training strategy.

Table 9: Detection rates for obfuscated, malicious apps using two training strategies.

| | | **Detection** | | | | **Family Identification** | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Trained on Original** | **Features** | **C4.5** | **1NN** | **Train Size** | **Test Size** | **C4.5** | **1NN** | **Train Size** | **Test Size** |
| **Yes** | **Flow** | 98.82% | 98.48% | 9,547 | 1,188 | 69.78% | 71.46% | 1,217 | 1,188 |
| | **Flow, IA** | 96.38% | 96.89% | 9,533 | 1,188 | 71.89% | 69.87% | 1,217 | 1,188 |
| | **Flow, PAPI** | 99.49% | 99.92% | 8,898 | 1,186 | 96.42% | 95.57% | 1,217 | 1,186 |
| | **SAPI** | 97.42% | 99.00% | 12,906 | 1,186 | 92.84% | 99.50% | 1,259 | 1,186 |
| | **SAPI, IA** | 97.98% | 93.11% | 12,866 | 1,186 | 74.79% | 72.52% | 1,248 | 1,186 |
| | **SAPI, PAPI** | 99.50% | 100.00% | 11,588 | 1,201 | 96.67% | 99.67% | 1,259 | 1,201 |
| **No** | **Flow** | 90.66% | 90.32% | 8,359 | 1,188 | 68.15% | 66.95% | 607 | 584 |
| | **Flow, IA** | 88.55% | 86.70% | 8,345 | 1,188 | 61.30% | 65.24% | 607 | 584 |
| | **Flow, PAPI** | 96.96% | 95.36% | 7,712 | 1,186 | 77.49% | 87.46% | 607 | 582 |
| | **SAPI** | 93.42% | 92.01% | 11,720 | 1,186 | 83.22% | 86.44% | 626 | 590 |
| | **SAPI, IA** | 92.35% | 87.40% | 11,680 | 1,186 | 75.17% | 68.66% | 627 | 584 |
| | **SAPI, PAPI** | 92.01% | 94.09% | 10,383 | 1,201 | 88.81% | 87.63% | 626 | 590 |

For the first training strategy, the detection rate is very high for all combinations of features and classifiers—ranging from 93% to 100%. The combination of sensitive API and package API features along with a 1NN RevealDroid classifier obtains a perfect detection rate. Flow features and package API features used with a 1NN classifier obtains a near perfect 99% detection rate.

For the second training strategy, the detection rate remains high for all combinations of features and classifiers, ranging from 86% to 97%. Unlike for the first strategy, the C4.5 classifiers tend to slightly outperform 1NN classifiers. The combination of flow and package API features still outperforms other combinations of flow features, similar to the first training strategy. Sensitive API features alone also outperform their combination with Intent action features.

As can be observed in Table 9, Intent action features slightly reduce the detection rate for obfuscated apps, unlike for non-obfuscated detection (see Section 5.1). This result indicates that Intent action features are sensitive to obfuscations. For that reason, we do not include a combination involving Intent action features as a third feature involving two other features (e.g., Flow, IA, and PAPI). We will examine the affect of obfuscations on Intent action features more below in the case of family identification under obfuscation.

For family identification, the effect of obfuscation on different features varies widely. The set of transformations that we have applied affect standalone flow features significantly for both training strategies—with only a 67%-71% correct classification rate. 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 being 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.

The results for family identification under obfuscation improve significantly for flow features when combined with package API features—with an improvement of about 25% for both classifiers when using the first training strategy. The improvement is still considerable but less dramatic for the second training strategy, where the classification rate increases by 9% for C4.5 and 20% for 1NN.

Sensitive API combined with package features exhibit very high obfuscation resiliency for both training strategies. For the first training strategy, a 1NN classifier with sensitive and package API features achieves a 99.67% classification rate—the highest classification rate for the first training strategy. For the second training strategy, a C4.5 classifier with sensitive and package API features achieves an 89% classification rate.

Across all combinations of features, sensitive API features combined with package API features exhibit the most obfuscation resiliency for both detection and family identification. These results suggest that, without having to compute expensive flows, both detection and family identification of Android malware can be accurately performed, even for obfuscated malware.

## 5.4 RQ4: Efficiency Comparison

The number of both benign and malicious Android apps is growing very quickly [12] 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 [45].

To assess the efficiency improvement of RevealDroid's feature extraction over a state-of-the-art 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 obtains its highest accuracy by using flow features alone. Furthermore, MUDFLOW and RevealDroid ex-

Table 10: 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** |

tract flow features utilizing the same tool, FlowDroid. 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. We do not focus on classifier training and testing times because all of our datasets and combinations of features—which contain thousands of apps—can execute in a matter of minutes on a PC laptop. However, execution of feature extraction can take hours for certain features on the nodes of the computing cluster we utilized (recall Section 4).

For our runtime analysis, Table 10 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. Our selection of apps vary across several dimensions, allowing us to draw broader lessens about RevealDroid's and MUDFLOW's feature extraction efficiency. 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 both MUDFLOW and RevealDroid extract using the same tool, 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 [19].

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 and ranging from 2 seconds to 24 seconds.

Due to the overlap between sensitive API features and flow features—recall that security information flows to and from sensitive API methods—the use of each feature alone obtains high accuracy in general. However, there is a greater than 85 times speedup from extracting sensitive API features instead of flow features. Remember that some apps (marked with gray cells in Table 10) actually take more memory and time to run than we were able to allocate even on a high performance computer 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. Combining sensitive API, package API, and Intent action feature extraction times together still achieves a 33 times speedup compared to flow feature extraction.

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.

## 5.5 RQ5: Detection Comparison

To determine RevealDroid's accuracy improvement over the state-of-the-art in Android malware detection, we compared it against MUDFLOW. 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 4, 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

Table 11: Comparison of MUDFLOW 2-Way Classifier with RevealDroid's C4.5 SAPI Detection Classifier

| | **MUDFLOW 2-Way Classifier** | | | | | | **RevealDroid C4.5 Classifier with SAPI features** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **No Obfuscations** | | | **For Obfuscations** | | | **No Obfuscations** | | | **For Obfuscations** | | |
| | **Prec** | **Rec** | **F-Meas** | **Prec** | **Rec** | **F-Meas** | **Prec** | **Rec** | **F-Meas** | **Prec** | **Rec** | **F-Meas** |
| **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% |
| **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% |
| **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%** |

further leveraged MUDFLOW's two-way Support Vector Machine (SVM) classifier since it exhibited the greatest accuracy in the prior work [19]. Lastly, we modified MUDFLOW to allow it to accept a pre-defined training and testing set, as it could only perform cross-validation initially. This only modifies the manner in which input is provided to MUDFLOW and allows us to directly compare RevealDroid to it using the same training and testing set. We further verified the correctness of our modification by checking that it obtains the same results as a 2-fold cross-validation.

We compared MUDFLOW and RevealDroid in the following two scenarios for training and testing: one involving only the original untransformed apps, and another involving apps transformed as described in Section 5.3. In the scenario with no transformed apps, we split a dataset consisting of 8,013 malicious apps and 1,742 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 5.3 and (2) the remaining 866 benign apps. As before, computational resource limits and errors from Soot or FlowDroid prevented us from extracting features from all 1,201 obfuscated, malicious apps.

For classifier selection, we compared MUDFLOW's two-way classifier with RevealDroid's C4.5 classifier that uses sensitive API features only. This selection of classifier and features are obfuscation resilient and highly efficient, but not necessarily the most accurate, as demonstrated in the previous sections. However, our results below will show that it still outperforms MUDFLOW in terms of accuracy.

Table 11 showcases the *Prec*ision, *Rec*all, and *F-Meas*ure results for MUDFLOW's classifier and RevealDroid's C4.5 classifier on the two scenarios for both *Ben*ign apps and *Mal*icious ones. Overall, for both scenarios, 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 indicates that MUDFLOW's classifier has a strong tendency to mark benign apps as malicious, unlike RevealDroid's classifier.

## 5.6 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, i.e., Dendroid [35], 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 [42], that is capable of identifying families. However, DroidSIFT's authors are unable to share their implementation or dataset. Consequently, we could not compare against DroidSIFT.

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 5.3. 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 5.3).

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 [35], we verified our results with those authors and discovered an error in their experiment, where they, in fact, trained on the entire dataset.

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 5.3. 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 unsuprising 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.

### 5.7 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. The most inefficient features to extract are flow features, due to the need to compute a potentially expensive data-flow analysis. However, our results strongly indicate that sensitive API features, possibly with the combination of package API features, can replace flow features. Although Intent actions tend to improve detection and family-identification results, they are, unfortunately, not obfuscation-resilient. Overall, the features that best achieve obfuscation resiliency, accuracy, and efficiency are sensitive API features, possibly with package API features.

Flow features had limited obfuscation resiliency for family identification, despite their focus on Android APIs. An in-depth analysis to determine the source of FlowDroid's sensitivity to obfuscations is beyond the scope of this paper. However, it is possible that the DroidChameleon transformations result in special cases not handled by FlowDroid. More importantly, as previously stated, flow features are not needed to achieve high accuracy, obfuscation resiliency, and efficiency of Android malware detection and family identification.

RevealDroid is limited by the underlying analysis and tools utilized for feature extraction. In particular, Soot and FlowDroid could not extract features from all apps in our study. This limitation occurred mainly due to computational resource constraints and scalability issues, particularly for FlowDroid. To address these issues, more robust tools than Soot or FlowDroid may be substituted. Furthermore, missing a small number of apps is mitigated by our use of machine learning, which attempts to learn general characteristics of Android malware.

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 4). We further utilized family labels already verified by security experts (see Section 4 and Section 5.2). 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. Extracting features to represent these characteristics is an interesting challenge for future work.

## 6 Related Work

We provide an overview of the current state of Android malware app 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.

A variety of techniques have been developed to identify Android malware, without attempting to specifically identify the family of malware. Some techniques detect Android malware by focusing on specific risk factors. RiskRanker [24] ranks apps as either high-risk, medium-risk, or low-risk in order to identify malware. Peng et al. [28] perform risk ranking and scoring by leveraging probabilistic generative models to identify malware apps.

Other techniques utilize virtualization to aid in the detection of Android malware. DroidScope [40] is a virtualization-based malware analysis engine that utilizes different dynamic analyses to monitor malware. CopperDroid [34] is an approach for reconstructing Android-malware behaviors through virtualization and a focus on system calls.

Machine learning has been used for simply distinguishing between benign and malcious Android apps. CHABADA [23] compares app descriptions and behaviors through machine learning and information retrieval to distinguish benign and malware apps. MUDFLOW [19], which we compared with RevealDroid in Section 5, is another detection technique that trains on apps and attains higher accuracy than CHABADA.

Certain techniques leverage Android-app permissions to identify malware apps. Kirin [21] certifies an Android app against a set of rules to determine if the app may perform malicious behavior. DroidRanger [45] attempts to identify malware based on the permissions and behaviors of an app. Drebin [17] is designed to detect Android malware directly on an Android device, in part, by using permission information.

A variety of other techniques use different mechanisms for detecting Android malware. DroidAnalytics [43] provides an automated workflow for the collection and signature generation of Android malware by analyzing apps at the opcode level. AsDroid [26] detects stealthy behaviors of possibly malicious apps characterized by mismatches between program behavior and the user interface. Poeplau et al. [29] construct a static analysis tool for identifying unsafe and malicious dynamic code loading.

Besides not identifying malware families, all of these approaches are evaluated on an outdated set of malware—many of these approaches are evaluated on malware no older than 2012.

Several approaches focus on identifying specific malware families. Apposcopy [22] provides a language to specify malware signatures and a static analysis to identify apps matching those signatures. Given that Apposcopy is signature-based, 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 [35] utilizes text-mining techniques and control-flow features to identify families of malicious apps. DroidSIFT [42] 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 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 additional malware apps discovered up until early 2014.

Both techniques have limited obfuscation resiliency, and rely on representations (e.g., control-flow features or constant strings) that can be thwarted by malware using control-flow transformations. Dendroid is not evaluated for its ability to address obfuscations; DroidSIFT is only assessed using unstated obfuscations applied to a small number of apps from a single malware family.

## 7 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 a state-of-the-art Android malware detection approach, showcasing RevealDroid's superior accuracy and efficiency, even under obfuscation. We further compared RevealDroid to a state-of-the-art 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 [9] and stealthily utilizing devices to mine cryptocurrency services [3]—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 [7].

## References

[1] About argo. http://wiki.orc.gmu.edu/index.php/About_ARGO.

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

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

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

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

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

[7] RevealDroid. http://www.sdalab.com/projects/revealdroid.

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

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

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

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

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

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

[14] 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.

[15] E. Alpaydin. *Introduction to Machine Learning*. MIT press, 2014.

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

[17] 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.

[18] 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.

[19] 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. Preprint available at `https://www.st.cs.uni-saarland.de/appmining/mudflow/`.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.