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

JOSHUA GARCIA, MAHMOUD HAMMAD, AND SAM MALEK, Department of Informatics, University of California, Irvine

The number of malicious Android apps is increasing rapidly. Android malware can damage or alter other files or settings, install additional applications, etc. To determine such behaviors, a security analyst can significantly benefit from identifying the family to which an Android malware belongs, rather than only detecting if an app is malicious. 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 operates without the need to perform complex program analyses or to extract large sets of features. Specifically, our selected features leverage categorized Android API usage, reflection-based features, and features from native binaries of apps. We assess RevealDroid for accuracy, efficiency, and obfuscation resilience using a large dataset consisting of more than 54,000 malicious and benign apps. Our experiments show that RevealDroid achieves an accuracy of 98% in detection of malware and an accuracy of 95% in determination of their families. We further demonstrate RevealDroid's superiority against state-of-the-art approaches.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software reliability*;

Additional Key Words and Phrases: Android malware, obfuscation, machine learning, lightweight, native code, reflection

ACM Reference Format:
Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2016. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Trans. Softw. Eng. Methodol.* 9, 4, Article 39 (June 2016), 27 pages.
https://doi.org/0000001.0000001

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 [1, 3], demonstrating the growing sophistication of mobile malware. Consequently, a significant amount of effort has been expended on producing techniques to detect Android malware.

1049-331X/2016/6-ART39 $15.00
https://doi.org/0000001.0000001

Existing work on Android malware detection has focused on distinguishing between benign and malicious apps. A number of these approaches utilize permissions requested or used by an app to identify Android malware, including through the use of custom signatures [26, 64, 66] or machine learning [17, 54]. Other techniques identify malicious apps by ranking their riskiness [32, 42]. Alazab et al. and Adagio [12, 30] use graph structures to identify malware. Other techniques identify malicious apps by comparing program behavior with other aspects of an app, including the user interface [34] and app descriptions on app markets [31]. These approaches have made significant and important steps toward identifying malicious apps from individual devices and app markets.

Although accurately identifying if an app is benign or malicious 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 [40]. 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 [65], DroidDream [65], Oldboot [7], 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 [5, 16, 45]. For example, a variety of malware uses reflection to obfuscate security-sensitive behaviors [44]. 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 [45]. 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 [17, 30, 50, 60]. These techniques use features that are not resilient to obfuscations. For example, some features utilized by existing approaches are based on control flow [30, 50], which are susceptible to control-flow obfuscations (e.g., addition of junk code or call indirection). As another example, features involving constant strings [17, 60] are susceptible to encryption or renaming obfuscations.

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 [18, 34]. 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 [23, 66]. 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 set of features selected to achieve obfuscation resiliency, efficiency of analysis, and accuracy. It does not require complex program analyses (e.g., data-flow analysis [18, 60]) or large sets of features (e.g., hundreds of thousands of features [17, 30]), which can lead to scalability problems. More specifically, our selected machine-learning features are based on Android-API usage, including resolution of APIs invoked using reflection, and function calls (e.g., system calls) made by native binaries within an Android app. No previous work has included native-code feature extraction to detect malware. Including features based on reflection and native code significantly aids RevealDroid with achieving obfuscation resiliency.

RevealDroid is capable of accurately detecting malicious apps with a 98% accuracy, and identifying their families with a 95% accuracy, in under 90 seconds on average. RevealDroid can maintain high accuracy even for obfuscated apps. 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 24,600 benign apps and over 30,000 malicious apps from two different malware repositories. We further compare RevealDroid's detection and family-identification accuracy against state-of-the-research approaches: Adagio [30], Drebin [17], and MUDFLOW [18], both of which are approaches for malware detection; and Dendroid [50], an approach for malware-family identification. RevealDroid has an overall greater accuracy by about 11%-25% and mislabels 25%-54% fewer benign apps as malicious than MUDFLOW; RevealDroid achieves up to 23% greater accuracy than Adagio and up to 60% greater accuracy than Drebin. Additionally, RevealDroid achieves a 24%-70% higher classification rate than Dendroid.

This paper makes the following contributions:

- RevealDroid demonstrates that highly lightweight analyses that extract API-based features—including those based on reflection—and native code features combined with machine learning, can achieve high accuracy, scalability, and obfuscation resiliency.
- We construct an updated dataset of over 27,900 malware apps labeled with their 447 malware families and assess RevealDroid's family-identification accuracy on that dataset. We make this updated dataset available for researchers and practitioners [4].
- 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, Drebin, and MUDFLOW, and for family identification against Dendroid. We also make the transformed dataset available online [4].
- We assess the efficiency of RevealDroid's feature extraction and machine-learning classification. We show that RevealDroid's features can be more than 13 times faster than information-flow feature extraction—which are features used in a variety of Android malware detection tools [18, 60, 62]—while still exhibiting obfuscation resiliency and accuracy. We further demonstrate that RevealDroid can produce classifiers efficiently, as compared to other state-of-the-research tools.

The remainder of this paper is structured as follows. Section 2 introduces RevealDroid and its design. Section 3 covers our evaluation design, the research questions we study, evaluation results, and RevealDroid's limitations. The last sections cover work related to RevealDroid (Section 4), and conclude the paper (Section 5).

2 REVEALDROID

Malware detection and family identification can be placed into two categories: signature-based and machine learning-based [60]. 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.

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 Android API methods, the system calls, and other low-level library calls invoked by an Android app 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 framework-, system-, and library-level invocations, which in different combinations tend to be malicious or benign, RevealDroid is capable of achieving obfuscation resilience.

In the rest of this section, we discuss the features utilized by RevealDroid, the labeling of apps and RevealDroid's use of supervised learning to produce classifiers for detecting malware and identifying their families, and other features we considered but ultimately excluded from RevealDroid.

2.1 Features Chosen for Learning

To construct RevealDroid, we explored a variety of statically extractable features, both those previously used by other researchers and novel ones. Our goal when designing RevealDroid is to select features that meet three criteria: *accuracy* since any malware detection or family identification should be as correct as possible; *efficiency*, in order to quickly detect malware and its malicious behaviors before it propagates widely; and *obfuscation resiliency* to address different ways malware may evade detection. No malware detection or family identification technique is obfuscation proof, i.e., capable of identifying malware or its family for all possible evasion techniques. However, RevealDroid's aim is to be resilient to as many obfuscation techniques as possible.

To achieve accuracy, efficiency, and obfuscation resiliency, RevealDroid contains the following four types of features: package-level Android API usage (PAPI), method-level Android API usage (MAPI), reflection, and native code. We describe each of these features in more detail in the following paragraphs.

Android API invocations or accesses have been used as features [17, 54]. MAPI features in our formulation are the number of invocations of a specific Android API method. An example of a

MAPI feature value is simply the number of times **`TelephonyManager.getSimNumber()`** is invoked. Categorization of API usage has been shown to be useful in previous malware detection work [18]. To obtain categories of APIs, we simply used PAPI features since packages are specified by Android framework developers themselves.

Increasingly, Android malware are relying on reflection, i.e., the ability of a program to modify or inspect itself at runtime, in order to perform malicious behaviors or obfuscate such behaviors [43]. At the same time, benign apps utilize such behaviors to perform legitimate operations (e.g., update an app with the latest features or bug fixes without having to re-install the entire app). Due to the increasing prevalence of reflection, we included it as part of RevealDroid. Furthermore, given that obfuscating the target (e.g., method or field) of a reflective call is an indicator of suspicious behavior, it is possible—as our evaluation will demonstrate—to identify malicious reflective usage without needing to fully resolve all reflective calls.

An Android app can use native code to improve the performance of the app, which is often used for games, or to make use of shared native libraries. However, malware authors can utilize native binaries to package exploits, hide behavior from anti-malware techniques that do not scan native binaries, or perform other malicious functionalities [65]. Native code is often ignored as part of Android malware analysis, especially if that analysis is static. Consequently, we included it in RevealDroid.

2.2 Labeling and Classifier Selection

Through supervised learning, RevealDroid aims to utilize the aforementioned features to determine which combinations of them indicate malicious behavior and the specific family most likely to exhibit that behavior. As a result, 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.

To that end, RevealDroid can build multiple n-way classifiers, where n is the number of labels for Android apps. To 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 48 malware families, resulting in $n = 48$ for a malware classifier trained on Malware Genome. Given that SVMs (Support Vector Machines) are inherently two-way classifiers [15], we select a linear SVM for malware detection. For family identification, RevealDroid produces a CART (Classification and Regression Trees) classifier [20], which is a type of decision tree classifier that handles multiclass classification effectively [15]. We demonstrate the efficacy of our choice of classifiers in Section 3.

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 families of malicious apps.

2.3 Android API-Usage Extraction

The Android API contains security-sensitive functionality (e.g., sending SMS messages, and accessing private repositories or location information). We leverage two means of representing Android API usage: the number of Android API method invocations, representing MAPI features, and the number of method invocations for specific Android API packages, representing PAPI features. For example, in the case of PAPI, **`android.account`** provides APIs for handling account information; **`android.media`** contains APIs for managing media interfaces to audio and video. These features

have been shown to be useful for distinguishing malware families when manually specifying their signatures [27]. Consequently, we chose to include such features for detecting and identifying families of Android malware using machine learning.

To that end, we built the *Android API-Usage Extractor*, which determines the number of API invocations per Android package and the number of invocations per Android API method. As an example of the number of package-level invocations, 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.

To illustrate how such features can help distinguish malware families, Table 1 depicts features from three Android malware families. Each feature is denoted by a package name within the **`android.*`** top-level Android API package. For example, in Table 1, jSMSHider accesses sqlite APIs twice and the telephony package 8 times. The table shows that a supervised learning algorithm can determine that Geinimi samples access location APIs significantly more than jSMSHider or BaseBridge. The learning algorithm can also determine that both jSMSHider and BaseBridge access the telephony and sqlite packages. However, jSMSHider accesses telephony packages more than BaseBridge does; and BaseBridge accesses sqlite packages more than telephony packages.

Table 1. Example package API features from known Android malware families

	telephony	location	sqlite	Fam
mal1	8	0	2	jSMSHider
mal2	0	12	0	Geinimi
mal3	2	0	7	BaseBridge

In the case of method-level invocations, if **`telephony.TeleMgr.listen`** is invoked by an app twice, then that method obtains a value of 2. Formally, the feature vector $MAPI_a = (m_1, ..., m_i, ..., m_{|M|})$, where $m_i = |\{m\}|$, M is the set of Android API methods, m is an invocation of a method $\mu \in M$. Table 2 depicts the example in Table 1 where packages are expanded into methods.

Table 2. Example method-level API features from known Android malware families

	telephony.TeleMgr.listen	location.LocMgr.rmUpdates	location.LocMgr.reqLocUpdates	sqlite.Db.execSQL	Fam
mal1	8	0	0	2	jSMSHider
mal2	0	6	6	0	Geinimi
mal3	2	0	0	7	BaseBridge

2.4 Reflective Feature Extraction

A type of feature often ignored by existing Android malware detection and classification techniques are those that involve reflection and, as a result, dynamic class loading. Dynamic class loading through reflection allows an app to modify or inspect itself during runtime, and violate certain language constructs related to information hiding (e.g., allow access to the private members of a class). At the same time, Android malware are increasingly utilizing reflection to obfuscate their malicious behaviors [43]. To address this issue, RevealDroid extracts statically attainable information about reflection. Specifically, RevealDroid determines if a method is reflectively invoked, and the extent to which reflective APIs are used. RevealDroid further separates reflective invocations into three categories [39]:

- *fully resolved:* Both the reflectively invoked method and class names (e.g., **`TelephonyManager.getSimNumber()`**) can be statically determined;
- *partially resolved:* Only the invoked method name (e.g., **`getSimNumber()`**) can be statically determined
- *unresolved:* Neither the method name nor the class name can be determined statically (e.g., a non-constant string provided as input during runtime).

Partially extracted reflective invocations occur in cases where non-constant strings, or inputs, are used as target methods of a reflective call. Additionally, as we have observed in Android malware, a high number of unresolvable, reflective method invocations (e.g., reflective calls whose target is encrypted) tend to be malicious. Although reflection enables useful abilities, such as allowing an app to update itself so that users can have the latest features or bug fixes, reflection when used in excess is a strong indicator of malicious behaviors.

Reflective invocation of a method, for both constructor and non-constructor methods, occurs in three stages: (1) class procurement (i.e., a class with the method of interest is obtained) (2) method procurement (i.e., the method of interest to be invoked is identified), and (3) the method of interest is actually invoked. *Reflection Extractor* attempts to identify information at each stage for the three types of reflective invocations described above.

```
1  ClassLoader cl = MyClass.getClassLoader();
2  try { Class c = cl.loadClass("MyActivity");
3        ...
4        Method m = c.getMethod("onPause",...);
5        ...
6        m.invoke(...); }
7  catch { ... }
```

Fig. 1. Simple reflective method invocation example

A simple example, based on those found in real-world apps, of reflective method invocation, not involving constructors, is depicted in Figure 1. In this example, a **`ClassLoader`** for **`MyClass`** is obtained (line 1), which is responsible for loading classes. The **`MyActivity`** class is loaded using that **`ClassLoader`** (line 2). The **`onPause`** callback of **`MyActivity`** (line 4)—which pauses a component that has been running—is retrieved and eventually invoked using reflection (line 6). Apps that try to alter the standard Android lifecycle, by invoking callbacks, are indicators of potentially malicious behaviors. Furthermore, any security-sensitive behavior invoked using reflection is, at the least, suspicious.

Our analysis identifies reflectively invoked methods using a backwards analysis. That analysis begins by identifying all reflective invocations (e.g., line 6 in Figure 1). The analysis currently does not consider the various different parameters or arguments that can be passed to a method invoked reflectively. However, it does track the number of times a particular method is reflectively invoked, which is used as a feature for supervised learning.

Next, the analysis follows the use-def chain of the invoked **`java.lang.reflect.Method`** instance (e.g., **`m`** on line 6) to identify all possible definitions of the **`Method`** instance (e.g., line 4). Our analysis considers various methods that return **`Method`** instances, i.e., using **`getMethod`** or **`getDeclaredMethod`** of **`java.lang.Class`**. The analysis then records each identified method name. If the analysis cannot resolve the name, this information is also recorded.

At that point, the analysis attempts to identify the class name that is being invoked. Similar to the resolution of method names, the analysis follows the use-def chain of the **`java.lang.Class`** instance from which a **`java.lang.Class`** is retrieved (e.g., following the use-def chain of **`c`** on

line 4). We model various means of obtaining a **`java.lang.Class`** instance. For example, the class may be loaded by name using a **`ClassLoader`**'s **`loadClass(...)`** method (e.g., line 2), using **`java.lang.Class`**'s **`forName`** method, or through a class constant (e.g., using **`MyClass.class`**). The analysis then records the class name it can find statically, or stores that it could not resolve that name. Note that our analysis considers any subclass of **`ClassLoader`**, including the Android-specific **`DexClassLoader`** that allows dynamic loading of classes stored in the Android Dalvik Executable format. Our reflection analysis involving constructors works in a similar manner by analyzing invocations of **`java.lang.reflect.Constructor`** and invocations of its **`newInstance`** method.

Overall, for the three categories of reflective invocations described previously (full, partial, or unresolved), the analysis obtains the following feature information for each app: the full or partial method names invoked; the number of times the full or partial method name is invoked; and the total numbers of fully, partially, or unresolved reflective invocations. As an example, Table 3 shows samples of features for a fully resolved method name (**`SmsManager.sendTextMessage`**), a partially resolved one (**`getSimSerialNumber`**), and unresolved method names.

Table 3. Sample reflection features from real Android malware apps

	SmsManager.sendTextMessage	getSimSerialNumber	UNRESOLV
mal1	6	0	0
mal2	0	4	0
mal3	0	0	6

2.5 Native Call Extraction

A capability of Android apps that is almost never taken into account is use of native code. In particular, to the best of our knowledge, no analysis that utilizes machine learning and static analysis examines the internal behaviors of an app's native binaries. This allows malware authors to package malicious payloads in native binaries, since they are largely ignored. To address this issue, RevealDroid includes a *Native Call Extractor (NCE)* that records calls (e.g., system calls and calls to shared libraries) made by a native binary to entities outside of it, and the extent to which these calls are invoked.

To extract information about security-sensitive invocations in native binaries, NCE must disassemble binaries in a popular binary format for Unix-like systems called the Executable and Linkable Format (ELF). A typical ELF binary, in Android, consists of headers describing meta-data about the binary (e.g., address format, sections of the binary, memory layout information, etc.). After the header, the binary file is divided into sections containing code, data, and potentially other extra information.

To identify malicious behaviors, we focus on calls that the binary may make external to itself and represent key semantics of security-sensitive behavior. In particular, the NCE extracts system calls and other calls that the binary makes to external binaries (e.g., shared libraries). While the main code segment of a native binary on Android is relatively easy to obfuscate (e.g., storing data at non-standard addresses, or adding dead code), these external calls are not easy to obfuscate, particularly system calls.

To identify external binary calls, NCE must identify any call within the code segment of a native binary, and the appropriate assembly instructions that realize a function call. Within an ELF binary in Android, this information is stored in the Procedure Linkage Table (PLT) of the binary. Simply put, the PLT is used to determine the address of external functions not known at linking time.

As an example, consider the disassembled PLT section of a native binary containing the GingerBreak root exploit, shown in Figure 2 and reduced due to space limitations. In that section,

```
1  00008b54 <sendmsg@plt>:
2      8b54: e28fc600 add ip, pc, #0, 12
3      8b58: e28cca02 add ip, ip, #8192
4      8b5c: e5bcf61c ldr pc, [ip, #1564]!
5  00008af4 <chmod@plt>:
6      8af4: e28fc600 add ip, pc, #0, 12
7      8af8: e28cca02 add ip, ip, #8192
8      8afc: e5bcf65c ldr pc, [ip, #1628]!
```

Fig. 2. PLT of a GingerBreak sample

```
1      99ec: e59d0010 ldr r0, [sp, #16]
2      99f0: e59f13c0 ldr r1, [pc, #960]
3      99f4: ebfffc3e bl  8af4 <chmod@plt>
```

Fig. 3. Code segment where chmod is invoked

the location of two security-sensitive system calls are identified: sendmsg for sending messages over sockets (starting at address 8b54), and chmod (starting at address 8af4) for modifying the permissions of a file. Each sequence of instructions modifies the program counter so that the machine begins executing at the address of the appropriate system call. For example, lines 2-4 of Figure 2 first computes an address at which the sendmsg code resides, and then loads that address to the program counter (pc), so that the code will execute. This binary is stored in the app's **assets/** directory, intended to contain raw resources of an Android app, of the package containing the archive, and is named *gbfm.png* to make the file appear to be simply an image. To identify binaries obfuscated in that manner, NCE scans every file in the package containing the app, i.e., the APK, and checks the format of the file to see if it matches that of an ELF binary.

To properly identify the binary as an Android ELF file, it must be analyzed using the appropriate matching Application Binary Interface (ABI), which is the analog of an Application Programming Interface at the binary level. An ABI defines the manner in which an application's machine code interacts with the system or other binaries. Android supports a variety of hardware architectures (e.g., ARM, MIPS, and x86) built against an Android-specific C library; each of these architectures use a different ABI. Disassembly and proper identification of the particular ABIs requires use of the Android toolchain for that ABI. Incorrect selection of an ABI or toolchain (e.g., using standard GNU ARM disassembly for Android ARM binaries) will result in incorrectly disassembled code, which may appear to look correct.

Identification of system calls, or other external calls, actually invoked in a binary requires analyzing the `.text` section of an Android ELF binary, which contains its executable code. In such a binary, branching instructions realize invocations of external calls. Specifically, NCE scans the `.text` section of every native binary within an Android app for branch, branch with link, and branch with link and exchange instructions. For each instruction, our analysis determines if the instruction references a label for a function in the binary's PLT.

To illustrate, Figure 3 depicts an invocation of the chmod system call. The initial two instructions prepare the first (r0) and second (r1) arguments that are passed to chmod. The final instruction invokes chmod using the address of the external call in the PLT (8af4 in Figure 2).

Overall, NCE records each external call of every binary in an Android app, and the number of times each external call is invoked, which together serve as feature types for supervised learning. By scanning every binary, our analysis ensures that no code is missed, even if the code is not invoked using Android's native code interface. For example, this behavior is common for Android

root exploits (e.g., executing the binary by using the **`Process`** or **`Runtime`** Java classes). An example of three system call features from three different Android malware apps is depicted in Table 4.

Table 4. Native call features from real Android malware apps

	chmod	rename	unlink
mal1	6	9	13
mal2	4	6	0
mal3	3	0	6

2.6 Other Features Considered

To construct RevealDroid, we explored a variety of statically extractable features, both those previously used by other researchers and novel ones. Table 5 depicts the various features we considered in comparison with our three criteria of interest (accuracy, efficiency, and obfuscation resiliency) but did not include since they did not meet one or more of those criteria. The various features include the following : *Perm*issions, *Comp*onent names, and Intent Filters (*IFilters*) attainable from an Android app manifest; security-sensitive data *Flows*; and Intent actions (*IActions*). ✓indicates the feature meets the criterion in question; ✘ indicates that it does not. Next, we discuss each feature, and our reasons for discarding it.

(1) *Android manifest properties.* An *Android application archive (APK)*, i.e., a compressed archive containing an installable Android app, is distributed with an XML manifest file that contains a variety of metadata about an app. Extracting information from a manifest file can be highly efficient, since it requires simply parsing an XML file. Among the information available in an app's manifest file, we considered using the following as features. However, we discard them due to either not contributing significantly to accuracy or for not being obfuscation resilient, as demonstrated in previous studies [46].
   - *Permissions.* Before Android 6.0, Android apps needed to request permissions at installation time. Starting with Android 6.0, users can revoke or grant permissions at app runtime. However, app permissions are highly granular. Although an app may even request more permissions than it actually uses, it may simply be requesting extra permissions in anticipation of its use in future versions.
   - *App components.* A variety of component types, with specific functionalities (e.g., components for providing GUIs, and others for running background services) are declared within an Android app's manifest. However, presence of particular components, especially simply tracking their name, as conducted by some approaches [46], can be obfuscated easily through renaming.
   - *Intent filters.* An app's manifest often declares messages, called *Intents*, it can receive and process through filters indicating Intent properties of interest. Although this information can

Table 5. Considered features and desired approach criteria: ✓indicates the feature meets the criterion; ✘ indicates that it does not

	Perm	**Comp**	**IFilters**	**Flows**	**IActions**
Acc	✘	✘	✘	✓	✓
Eff	✓	✓	✓	✘	✓
Obf	✓	✘	✘	✘	✘

be useful for identifying malware (e.g., those that listen for Intents indicating system actions), an app may simply declare filters in code, allowing for another form of obfuscation.

(2) *Security-sensitive data flows.* A few approaches for Android malware detection [18, 60] use data flows between security-sensitive Android interfaces to determine if an app is malicious. Tracking this form of information is particularly useful for identifying privacy leaks, but can be computationally expensive to compute [18]. For that reason, we exclude this feature from RevealDroid. Section 3.6 further examines efficiency issues with such flows. Furthermore, our experimentation demonstrated that call indirection actually affects data-flow analyses which, in turn, obfuscates privacy leaks, particularly in the case of family identification [29].

(3) *Intent actions.* Android malware are known to rely upon tracking the actions of an Intent (e.g., whether a package is installed, or if a device has recently completed booting) to determine when to perform a malicious behavior [27, 65]. In fact, these features are particularly useful for distinguishing between different malware families. Unfortunately, these features are relatively easy to obfuscate, due to the fact that Intent actions are stored as strings, which can be encrypted. In fact, we found that such features can cause a classifier to miss up to 27% of malware obfuscated using custom encryption transformations [28]. Thus, we exclude such a feature in RevealDroid.

3 EVALUATION DESIGN AND RESULTS

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 three other detection approaches, MUDFLOW, Drebin, and Adagio. Specifically, we answer the following research questions:

- **RQ1**: How accurate is RevealDroid for distinguishing between benign and malicious Android apps in a time-agnostic and time-aware scenarios?
- **RQ2**: How accurate is RevealDroid for identifying the specific family of a malicious Android app?
- **RQ3**: How does RevealDroid's detection accuracy compare to other detection approaches?
- **RQ4**: How does RevealDroid's family identification capability compare to another state-of-the-research malware-family identification approach?
- **RQ5**: Which features were selected and account for the detection or family identification capabilities of RevealDroid?
- **RQ6**: What is RevealDroid's run-time efficiency? How does this run-time efficiency compare to other learning-based approaches for malware detection?

We implemented RevealDroid in Java and Python. To construct the *Android API-Usage Extractor* and *Reflection Feature Extractor*, we leveraged Soot [52], a static analysis framework, and Dexpler [19], a translator from Android Dalvik Bytecode to Soot's intermediate representation. For *Native Call Extractor*, we utilized the Android ABI toolchain to disassemble binaries and identify Android ELF binaries, and constructed a custom-built extractor using Python. For machine learning, we selected Scikit-learn [41], a widely used machine-learning toolkit for Python. For our experiments, we used a machine with 64 cores and 256GB RAM.

To assess RevealDroid's accuracy, we constructed a dataset of both benign and malicious Android apps. To obtain benign apps, we utilized AndroZoo [14], which is a repository of more than 5.5 million apps collected from several sources, including Google Play, the official Android market—and scanned by commercial anti-malware products from VirusTotal [9], an online service provided by Google that scans URLs, files, and Android apps to determine if they are malicious or benign. After scanning the AndroZoo dataset, we found over 24,600 Google Play apps, out of nearly 2 million apps, that are marked as benign by all 55 anti-malware products.

We obtained malware samples from four Android malware repositories: the Android *Malware Genome* project [65], the Drebin dataset [6], *VirusShare* [8], and VirusTotal [9]. Malware Genome contains over 1,200 Android malware apps from 48 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. The remaining apps were obtained from VirusTotal.

3.1 RQ1: Detection Accuracy

To answer RQ1, we assess how accurate RevealDroid is for detecting whether an app is benign or malicious in both *time-agnostic* and *time-aware* scenarios. In a time-agnostic scenario, training and testing as part of machine learning is conducted without considering the age of apps in the dataset. This scenario has been utilized to evaluate an overwhelming majority of machine learning-based Android malware-detection approaches [13]. A time-aware scenario uses the modification date of apps to determine training and testing sets, which avoids training on apps from the future to test on apps from the past.

To evaluate RevealDroid in a time-agnostic scenario, we utilized our entire dataset of Android apps. Table 6 depicts results for a 10-fold cross-validation, 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; *F1* score is the weighted harmonic mean of precision and recall; the *No.* of *Apps* used; averages (*Avg.*) for precision, recall, and the F1 score; and the *Total* number of apps.

Table 6. Detection results for time-agnostic scenario

	Prec	Rec	F1	No. Apps
Benign	98%	97%	98%	24,679
Malicious	98%	98%	98%	30,203
Avg./Total	98%	98%	98%	54,882

The table illustrates that RevealDroid achieves high accuracy across the board, for both benign and malicious apps, with an average F1 score of 98%. For just benign apps, RevealDroid obtains a 98% F1 score. For malicious apps alone, RevealDroid attains a 98% F1 score. These consistently high results across multiple measures demonstrates RevealDroid's ability to detect malicious apps with high accuracy.

Figure 4 shows the detection results for the time-agnostic scenario as part of a precision-recall (PR) curve. This PR curve is nearly perfect, as shown by being close to the upper right corner and having on overall area under the curve (AUC) of .98, where the ideal is 1.00.

To evaluate RevealDroid in a time-aware scenario, we followed the methodology described by Allix et al. [13]. Specifically, we extracted the modification date of the `classes.dex` file in each app's APK file. `classes.dex` contains the compiled implementation classes of the app. The date at which the file is modified allows us to determine the age of the app, which is used to split our datasets into training and testing.

We split our apps into training and testing for a particular date as follows: For each date, any apps older than that date are assigned to the training set; the remaining apps are assigned to the testing set. We selected dates as the first day of each year from 2012-2015. For example, we selected apps created prior to 1/1/2012 as training, the remaining apps are for testing.

Figure 5 depicts the results obtained for each year with selected dates for splitting from 2012 to 2015. For the first day of 2012, RevealDroid obtains an 87% F1 score and similar results for 2013.

Fig. 4. Precision-recall curve for detection in the time-agnostic scenario

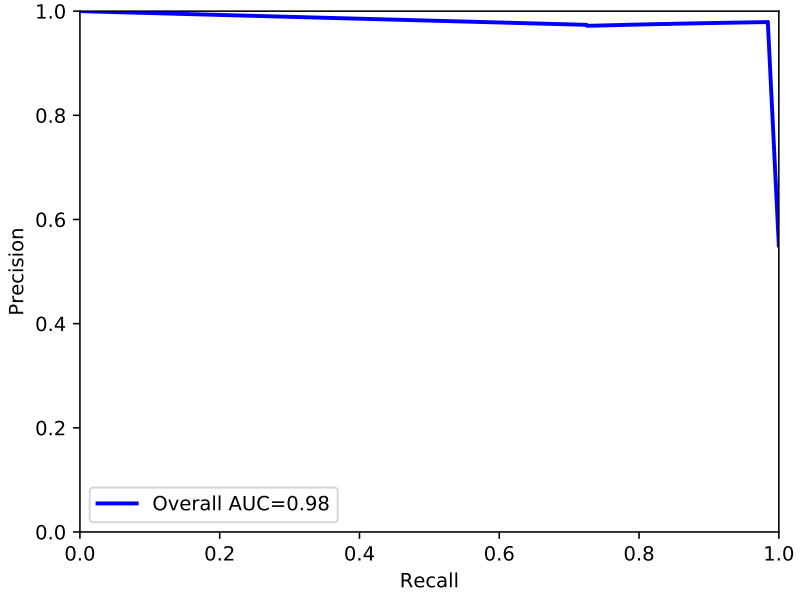
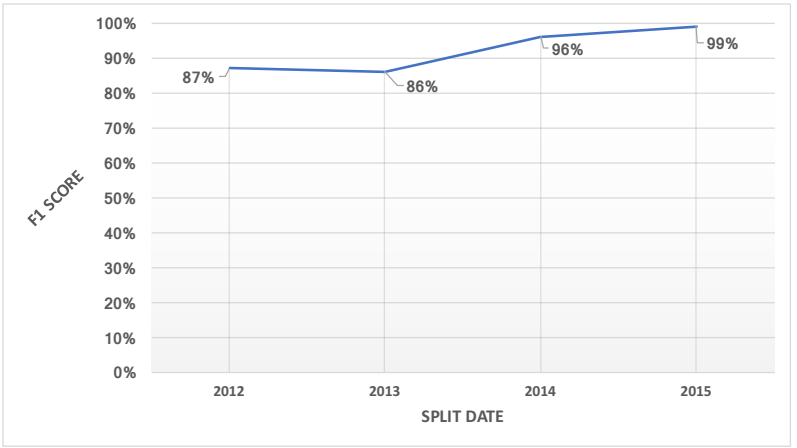


Fig. 5. Detection results for time-aware scenario



However, RevealDroid results only improve for the following years to 96% for 2014 and 99% for 2015. These results show that RevealDroid is able to obtain high accuracy, even when the age of apps is taken into account.

These results are particularly notable since previous work has demonstrated that machine learning-based Android malware detection was unable to obtain an F1 score higher than 70% in a time-aware scenario [13]. In that work, dates newer in time resulted in lower F1 scores; however, RevealDroid actually improves to as high as 99%. Consequently, RevealDroid exhibits a strong ability to obtain high detection results in both time-aware and time-agnostic scenarios.

3.2 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 with determining how to deal with the malicious app, besides simply removing it.

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) [65], which contains over 1,200 apps and 48 malware families, labeled by other researchers.

Figure 6 depicts a histogram of malware families in AMG. Notice that no family constitutes more than 25% of the apps in the dataset. Consequently, a naive classifier that labels all samples with the most frequently appearing label would only obtain an accuracy of 25%. The histogram indicates that this particular classification task requires a sophisticated classifier.

We used RevealDroid to construct a classifier with 48 different labels, one for each family in AMG. For this experiment, we conducted a 10-fold cross-validation to assess the accuracy of our classifier.

On the AMG dataset, RevealDroid's malware-family classifier obtains a 95% correct classification rate, far above the 25% correct classification rate for a naive classifier. These results showcase RevealDroid's ability to identify a malicious app's family with high accuracy. This outcome indicates that our features are well-chosen for discriminating between malware families.

RevealDroid's classifier did not reach perfect correctness due to a lack of samples for certain malware families: Malware families with less than 10 samples obtained lower results, since our cross-validation uses 10 folds. Ideally, when performing a cross-validation by selecting folds, the number of labels should be greater than or equal to the number of folds.

Expanded Families. To assess RevealDroid's classifiers' effectiveness on more recent malware families, we evaluated those classifiers on a much larger set of malware samples from Drebin, VirusShare, and VirusTotal. To produce a ground truth of families for malicious apps beyond those found in AMG, we again leveraged VirusTotal and a tool called AVClass [49], which provides an algorithm for identifying the malware family label of a malicious app using VirusTotal labels. We uploaded every sample in our malicious dataset to VirusTotal to obtain labels from all the anti-malware products on it. These labels are then passed to AVClass which then provides a family label for each malicious app. If AVClass could identify a family for the malicious app, we utilized it for this experiment. This process resulted in a dataset of 447 families and 27,979 malware samples.

Figure 7 shows the histogram of the 37 families among the 447 families that contain 100 or more samples. A random classifier would obtain only about a 0.22% correct classification rate; and a naive classifier that simply marks every app with the most frequent family label (jifake) would only obtain a 26% classification rate. As with the AMG dataset, this expanded family dataset poses a challenging classification problem, requiring a sophisticated classifier.

RevealDroid's family identification for this set of apps achieves an 84% correct classification rate, which is far above the 26% classification rate for a naive classifier. This result is particularly useful given the choice of 447 families to which an individual malicious sample may belong.

Note that a major contributing factor to the accuracy of RevealDroid from this expanded family-identification experiment is the use of AVClass and VirusTotal labels. These labels are obtained by an automated technique that relies on (1) heuristics and (2) labels from anti-malware products in VirusTotal, which often disagree with each other. As a result, those labels are likely less accurate than the manually curated AMG labels. That lower quality of family labels for AVClass is likely affecting the correct classification rate of RevealDroid for the expanded family dataset.

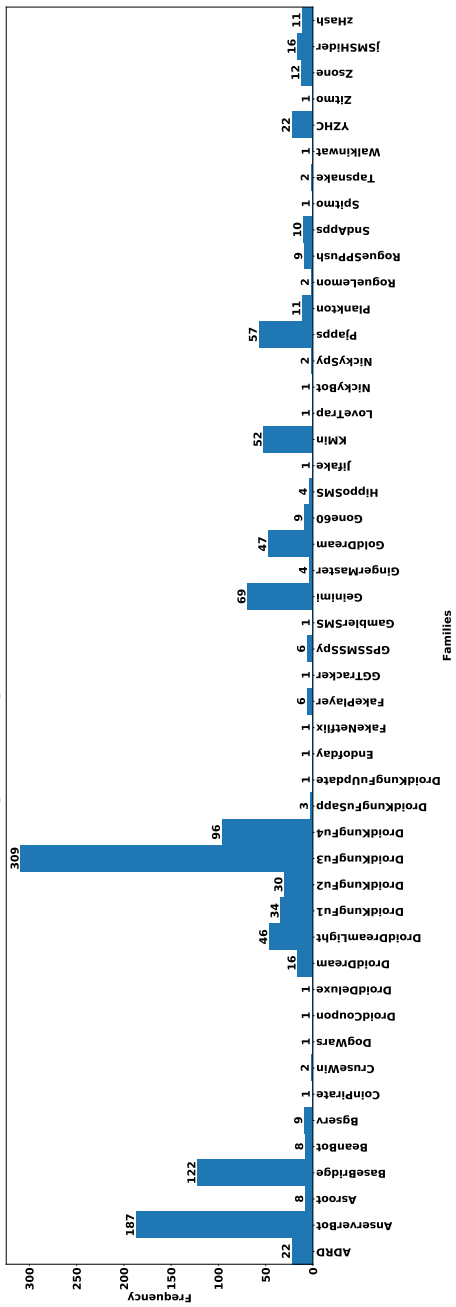


Fig. 6. Histogram of AMG Families

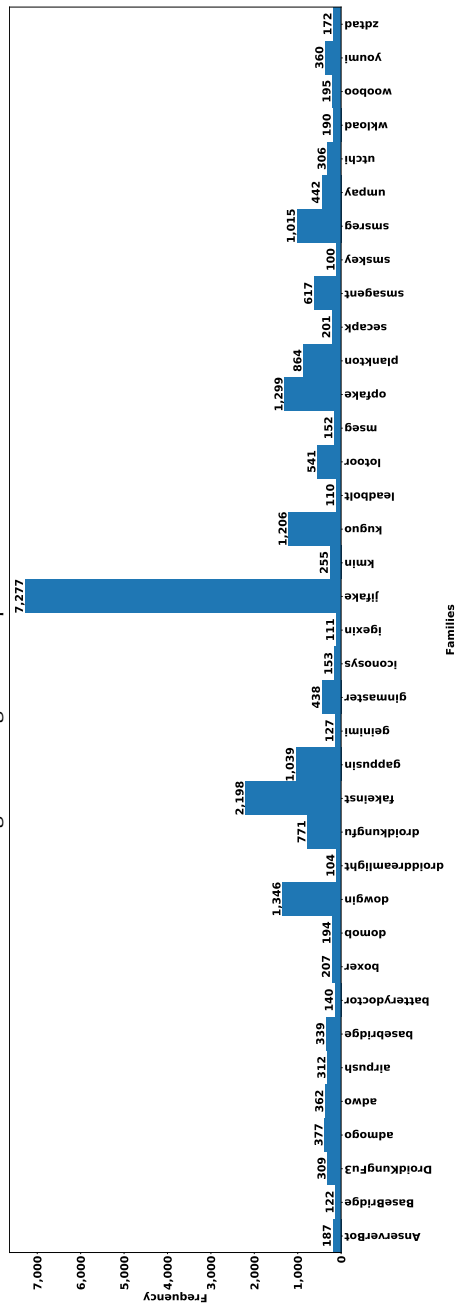


Fig. 7. Histogram of Expanded Families

3.3 RQ3: Detection Comparison

To assess RevealDroid against state-of-the-research approaches for Android malware detection, we compared it against three research prototypes: MUDFLOW, Adagio [30], and Drebin [17]. Besides MUDFLOW, we attempted to obtain state-of-the-research tools, DroidSIFT and Drebin [17], 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. Adagio operates by constructing function call graphs and encoding them as features used for machine learning.

Although the original Drebin implementation is unavailable, we decided to assess the extent to which Drebin's features for machine learning are useful for detection of Android malware. In particular, we selected three key features that are unique to Drebin: network addresses, requested permissions, and used permissions. Network addresses include URLs, IP addresses, and valid hostnames. Requested permissions are permissions an app requests at install time; used permissions are permissions that an app actually utilizes in its code, as determined by static analysis.

For MUDFLOW, we downloaded its implementation 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 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 and Drebin. Due to space limitations, we omit a comparison we conducted with 60 commercial anti-virus products. However, RevealDroid met or exceeded the detection rates of those products. The results of that comparison are available online [4].

We compared Adagio, MUDFLOW, Drebin, and RevealDroid in the following two scenarios: one involving only the original untransformed apps, and another involving apps transformed using DroidChameleon [45, 46], a tool that transforms apps in order to obfuscate them. In the scenario with no transformed apps, we split a dataset consisting of 7,989 malicious apps and 1,742 benign apps into a training set that has half of the benign apps and half of the malicious apps; the testing set has the remaining apps. For the other scenario, the training set consists of 7,995 malicious apps and 878 benign apps; the testing set contains (1) 1,188 malicious AMG apps to which DroidChameleon transformations are applied, and (2) 869 benign apps. For classifier selection, we used the most accurate classifiers of MUDFLOW, Adagio, and Drebin.

DroidChameleon transformations are designed to prevent anti-malware tools from detecting apps to which those transformations are applied. These transformations are based on obfuscations seen in the wild, and have previously been shown to prevent 10 commercial antivirus products from detecting the resulting transformed apps [46]. Another alternative obfuscation tool for Android we considered is ADAM [63]. 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, MUDFLOW's, Drebin's, and Adagio's obfuscation resiliency. Using AMG allows us to assess both the malware detection and family identification abilities of RevealDroid for obfuscation resiliency.

Table 7 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. We selected these transformations because they have been shown to evade anti-virus products [46], 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.

For each malicious app in AMG, we attempted to apply transformation sets in the following order (ts0, ts1, ts2, ts3), where we try each transformation set in that sequence until a set results in an installable app. For example, we first attempt to apply ts0 and if that fails we then try ts1. We continue in that manner until we have tried all four transformation sets.

Table 7. 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			

Table 8 showcases the *Pr*ecision, *Re*call, and *F1* score results for each approach and both scenarios, i.e., without transformations ($\neg\ T$) and with transformations (T). For each of those metrics, the table depicts results for *Ben*ign apps and *Mal*icious ones.

Table 8. Detection comparison, where each numeric result is expressed as a percentage[1]

	MUDFLOW						RevealDroid						Adagio						Drebin					
	$\neg\ T$			T			$\neg\ T$			T			$\neg\ T$			T			$\neg\ T$			T		
	Pr	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**
Ben	85	34	49	98	47	64	90	88	89	91	72	80	90	76	83	54	73	62	97	100	98	42	100	59
Mal	87	99	93	72	99	84	97	98	98	82	95	88	95	98	96	73	54	62	100	99	100	0	0	0
AVG	**86**	**66**	**71**	**85**	**73**	**74**	**96**	**96**	**96**	**86**	**85**	**85**	**92**	**87**	**90**	**63**	**63**	**62**	**99**	**99**	**99**	**18**	**42**	**25**

Overall, RevealDroid's classifier outperforms MUDFLOW's two-way classifier. With no tranformation, RevealDroid obtains an average F1 score of 96% compared to MUDFLOW's 71%. For the obfuscations scenario, RevealDroid obtains an average F1 score of 85% compared to MUDFLOW's 74%. The reason MUDFLOW's results improve or remain unchanged overall is likely due to the fact that transformations applied by DroidChameleon are based on transformations seen in the wild. Thus, the approach is likely learning about combinations of feature values that indicate obfuscations.

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 72% recall for benign apps compared to MUDFLOW's 47%. For benign apps in the other scenario, RevealDroid obtains a 88% recall compared to MUDFLOW's 34% recall. These results indicate that MUDFLOW's classifier has a strong tendency to mark benign apps as malicious, unlike RevealDroid's classifier.

Adagio obtains a 6% lower F1 score than RevealDroid in the scenario with no DroidChameleon obfuscations. Furthermore, with the DroidChameleon obfuscations, RevealDroid significantly outperforms Adagio by 23%. This low obfuscation resiliency for Adagio is particularly due to the

[1]Note that RevealDroid's accuracy changes from Section 3.1 due to differing dataset sizes and splitting strategies.

use of call-indirection transformations, which changes the expected call graph that Adagio utilizes to identify malware.

Drebin obtains a 3% higher F1 score than RevealDroid in the scenario with no DroidChameleon transformations. However, in the scenario with transformations, Drebin exhibits the least obfuscation resiliency compared to the other approaches, obtaining a 60% lower F1 score than RevealDroid. The drastic change in accuracy is due to Drebin's heavy reliance on network addresses, which account for most of the features utilized by Drebin. Network addresses are constant strings which are susceptible to identifier renaming and encryption transformations, utilized by DroidChameleon.

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

3.4 RQ4: 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 [50], 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 [60], which also identifies families. However, DroidSIFT's authors are unable to share their implementation. Consequently, we could not compare against it. Note that neither MUDFLOW, Adagio, nor Drebin 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. Given that 15 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.

Using that experimental setup, Dendroid correctly classified 73% of the test apps, while RevealDroid achieves a 97% correct classification rate. Although our replicated results for Dendroid are significantly lower than the Dendroid authors' original results [50], 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 3.3. The resulting test set contains 590 apps.

RevealDroid demonstrated overwhelmingly greater obfuscation resiliency than Dendroid: RevealDroid obtains a 97% 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.

3.5 RQ5: Feature Selection

To obtain a better understanding of the extent to which RevealDroid's manually chosen features (i.e., method-level, package-level, reflection-based, and native code-based features) affect its classifiers, we used automated feature selection to identify the features that affect RevealDroid's results most. Additionally, feature selection allows for faster creation of classifiers, reduced training and testing time, and reduces the possibility of overfitting [33, 56]. Specifically, we focused on the dataset of

54,882 apps used for RQ1, where each malicious app is labeled using its family name and every benign app is labeled as such. We obtain family labels using the methodology described in Section 3.2. By performing automated feature selection using such labeling, we are better able to understand RevealDroid's ability to identify malware families, rather than just its ability to distinguish benign apps from malicious ones. Our initial dataset contains over a million features. Consequently, to perform feature selection, we used a stochastic gradient descent (SGD) classifier, which is a classifier based on an optimization method for unconstrained optimization problems [11, 61] that supports incremental learning [21]. Incremental learning allows a machine learning algorithm to build a classifier incrementally in order to avoid storing all data in memory; given the number of features and apps we utilize, storing all of them in memory is intractable.

Through the feature-selection process described above, a total of 1,054 features were chosen. The resulting features included 595 method-level and package-level features, 454 native call features, and 5 reflection features. The five reflection features selected were features that aggregate information about specific Android APIs that are reflectively invoked. These features are the number of partially resolved API invocations; the number of fully resolved API invocations; the number of reflective API invocations, where the invoked class cannot be statically determined; the number of unresolvable reflective API invocations; and the total number of reflective invocations in the app.

For method-level and package-level features, the selection process chose a variety of security-sensitive API (SAPI) methods and UI-oriented API methods. The use of both security-sensitive and UI-oriented methods to distinguish between benign and malicious apps makes sense since having both types of information allows a classifier to identify the context necessary to decide whether an app's usage is malicious. For example, this intuition has been used by techniques that do not leverage machine learning to identify malicious apps or behavior, but instead identify mismatches among an app's UI and program behaviors [23, 34]. SAPI methods selected include those related to the sending and receiving of Intents, notifications, and other types of inter-process communication; access and manipulation of security-sensitive data stores, such as a Content Provider (i.e., a type of Android component that stores app-specific data) or SQLite database; preferences and settings of the app or the entire Android system; location information of the device (e.g., GPS coordinates); telephone functionality, including sending SMS messages and listening for changes of telephony state; and low-level Android operating system functionality (e.g., asynchronous task running, threading, power management, process killing, and process priority modification).

An assortment of UI-oriented API method types were selected, including the following: methods for operating on different types of standard Android widgets (e.g., images, pop-up windows, dialog windows, progress bars, etc.); web-based UI widgets (e.g., methods of the Android `WebView` class); methods of the Android component type representing a single UI screen (i.e., the Android `Activity` class) and its sub-components (i.e., Android `Fragments`); and Android graphics rendering and animation methods.

The native code features extracted include a variety of functions associated with exploits and security-sensitive functionality—and features associated with benign functionality. In terms of exploits or security-sensitive functionality, the following types of functions are selected: encryption and decryption functions (e.g. RSA functions); compression and decompression functions (e.g., for BZ2 compression); stack unwinding (e.g., used in return-oriented programming attacks); file and memory manipulation; concurrency control (e.g., threading and mutual-exclusion mechanisms); external application frameworks (e.g., the Mono platform implementation of Microsoft's .NET Framework); and exception handling and manipulation, which is critical for writing exploit code [37].

In terms of benign functionality, selected native code functions include graphics rendering libraries (e.g., OpenGL libraries) and image manipluation (e.g., JPEG and PNG manipulation). In

such cases, native code is sensible to use due to improved performance from executing code compiled for a specific hardware architecture.

3.6 RQ6: Run-Time Efficiency

The number of both benign and malicious Android apps is growing very quickly [10] making it 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.

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 in the following manner. We first created a histogram of app sizes with five bins, as depicted in Figure 8. From each bin, we randomly sampled 20 apps, resulting in 100 apps in total to be used to measure RevealDroid's feature extraction run-time efficiency. We then ran our three types of feature extractors on each app. Recall that both package-level and method-level feature extraction occur concurrently. Such an experiment allows us to assess the general run-time efficiency of each type of feature ensuring that we have sampled from apps with a variety of sizes.

Fig. 8. Histogram of app sizes from our dataset

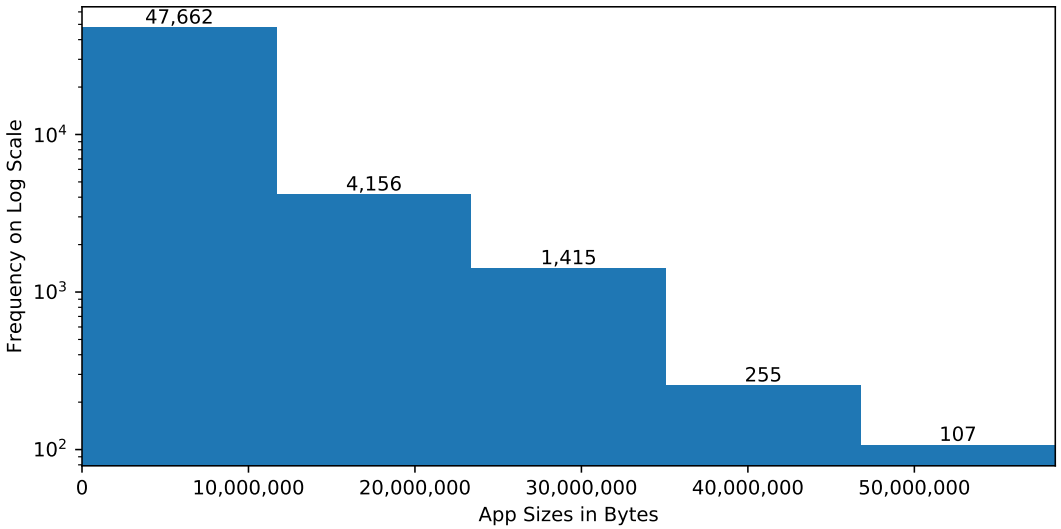


Table 9 shows the results of our feature-extraction efficiency analysis. Each feature takes under 90 seconds on average to compute. Furthermore, RevealDroid is designed to extract features in parallel, making the total feature extraction, on average, also under 90 seconds. This runtime is

reasonable for practical malware detection and family identification that is obfuscation-resilient and accurate.

Table 9. Average feature-extraction times for each type of RevealDroid feature in seconds.

	Native	Reflection	PAPI/MAPI
Average (s)	45.79	89.89	79.48

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 10. Feature extraction and classification run-times in hours

	RD-CART	RD-LSVM	Adagio	MUDFLOW	Drebin
Feature Extraction	84.79	84.79	56.12	1101.28	817.67
Classification	–	–	21.59	0.20	–
Total	84.79	84.79	77.70	1101.48	817.67

Table 10 depicts execution times, in hours, for both feature extraction and classification on 9,731 apps. The – indicates that classification takes under 2 seconds to run. For this experiment, we compared RevealDroid's CART (*RD-CART*) and linear SVM (*RD-LSVM*) classifiers with Adagio's, Drebin's, and MUDFLOW's classifiers. Each approach was run on our experiment machine, using the same hardware configuration. MUDFLOW took approximately 46 days to execute; Drebin took approximately 34 days to execute; RevealDroid's CART and SVM classifiers each take 3.5 days to execute; and Adagio's classifier requires about 3 days to execute. Given RevealDroid's superior obfuscation resiliency and its family identification capability, along with its high accuracy and efficiency, RevealDroid achieves its three main non-functional goals.

3.7 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. Our results demonstrate that these three qualities are achieved, in tandem, using RevealDroid.

Limitations of the dataset utilized by RevealDroid represents a threat to external validity. The number of apps in our dataset affect the generalizability of our results. To maximize our study's generalizability, we used a relatively large dataset, consisting over 50,000 apps, to assess RevealDroid. These apps range from 2011 to 2016.

Internal validity issues mainly arise due to the labeling of apps as benign, malicious, and belonging to a particular malware family. To mitigate this labeling threat, we carefully selected apps to maximize the probability that they are correctly marked as benign or malicious (see the preamble of Section 3). We further utilized family labels already verified by security experts (see Section 3.2). Moreover, machine-learning algorithms themselves are partially self-corrective, through statistical methods, for errors in the datasets.

Our choice of using DroidChameleon transformations to evaluate obfuscation resiliency of RevealDroid, and other approaches, is a threat to construct validity. In particular, DroidChameleon may not apply the most effective, realistic obfuscations to malware. This threat is alleviated by DroidChameleon's demonstrated ability to evade existing anti-virus products; its wide variety of

transformations, including those inspired by obfuscations observed in the wild; and its composite transformations. Moreover, some apps in our dataset use obfuscations, further mitigating this threat.

We further conducted a study where we trained on our entire app dataset and tested on 1,109 apps transformed with reflection transformations using Droid Chameleon. RevealDroid successfully detected all these apps as malicious. In future work, we intend to include reflection transformations as part of comparing RevealDroid with other malware detection approaches.

Although RevealDroid does extract reflection-related features, static analysis is limited in terms of its ability to extract information related to reflection. To reduce the affect of this limitation, we directly account for the partiality of our reflection features by representing the degree to which a reflective call can be resolved by our analysis.

One possible way to obfuscate native calls is to utilize the dynamic linker along with the associated `dlopen` and `dlsym` functions to load dynamically-linked functions. To obfuscate this behavior, a malware author can encrypt names in a native binary and decrypt the names during runtime. To handle this case, which has not been observed in Android malware so far, we can create features similar to those that RevealDroid uses for reflection: RevealDroid can determine the extent to which an invoked native call is encrypted. Given that including this type of feature worked well for reflection-based obfuscation (see Section 3.5), these features should aid in detecting malicious native-code obfuscations that leverage the dynamic linker. To further aid in detecting these types of malicious behaviors, RevealDroid can also include `dlopen` and `dlsym` as native-call features directly.

For family identification, we primarily chose a CART classifier due to the improved performance gain compared with an SVM, with no loss of accuracy. Specifically, we obtained approximately the same F1 score for an SVM and CART classifier, i.e., about 95% for the AMG dataset. However, while the SVM classifier takes 3,539 seconds to run, which is nearly an hour, the CART classifier only takes 194 seconds—which is 18 times faster. For that experiment, we selected an SVM with a linear kernel, a penalty parameter $C = 1.0$, square of the hinge loss as the loss function, and 1,000 as the maximum number of iterations.

One interesting aspect of our experiments, particularly compared to others, is the manner in which we sample benign apps and malicious apps to conduct machine learning. Some approaches choose imbalanced datasets [17]. Other approaches use significantly more malicious apps than benign apps [18]. At least one study has examined a balanced dataset and an imbalanced dataset where the majority class represents benign apps [48].

For our malware-detection experiment (Section 3.1), we chose to balance the samples. There are two key reasons for choosing a balanced dataset. First, different Android markets have varying levels of malicious apps compared to benign apps [23]. For example, certain Android markets have been known to have a ratio around 60% benign to 40% malicious apps [23]. Second, previous experiments have often chosen imbalanced datasets, which are not necessarily representative of Android markets in general, and may result in less accurate classifiers [35]. Consequently, given the varying degrees of ratios of benign apps to malicious apps on different Android markets, and to avoid biasing the classifier toward either malicious or benign apps, we chose to balance our dataset.

4 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.

Many non-machine learning-based Android malware detection approaches have been created. Some approaches mainly use Android-app permissions [26, 66]. Others focus on a variety of other risk factors to rank apps according to their suspiciousness [22, 32, 42]. A significant number of approaches focus on data leakage using taint analyses including dynamic taint analysis [25], combined static and dynamic analysis [55], taint analysis that focuses on user intention or user actions in association with data leaks [36, 58], or analysis of leaks that occur through inter-component communication [38, 53]. Other techniques leverage virtualization to monitor [38], reconstruct [47, 51], or trigger [38] malicious Android app behaviors.

Besides MUDFLOW and Adagio, other approaches have used machine learning for distinguishing between benign and malicious Android apps. DroidMat [54] 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.

AppContext [57] utilizes extensive analyses and machine learning involving information flow, Intent filters, Intent actions, and other context factors (e.g., conditions guarding security-sensitive behaviors). Although we considered including AppContext in our study, we could not set up a controlled experiment in the form we used to compare against MUDFLOW and Adagio for two key reasons. First, AppContext is not distributed with its source code, preventing us from modifying its training set as we did with MUDFLOW and Adagio. Second, this limitation further prevents us from comparing with AppContext in terms of its training execution time. However, our analysis takes about 30 seconds on average to analyze a single app; the AppContext study reports an average analysis time of 647 seconds [57]. Furthermore, as discussed in Section 2.1, Intent actions are likely to significantly reduce obfuscation resiliency due to their susceptibility to encryption transformations.

Drebin [17] 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), as demonstrated in our evaluation. Furthermore, their feature space is very large, containing about 545,000 features, as compared to RevealDroid's feature space of about 1,000 features, which allows our classification—and potentially our feature extraction—to be significantly more efficient and scalable.

ViewDroid [59] 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 [64] provides an automated workflow for the collection and signature generation of Android malware by analyzing apps at the opcode level. AsDroid [34] detects stealthy behaviors of possibly

malicious apps characterized by mismatches between program behavior and the UI. Poeplau et al. [43] construct a static analysis tool for identifying unsafe and malicious dynamic code loading. HARVESTER [44] extracts features relevant to anti-analysis techniques (e.g., obfuscations and emulator detection techniques) from Dalvik bytecode using static and dynamic analyses. Unlike HARVESTER, RevealDroid aims to utilize lightweight static analysis and machine learning to identify malicious apps, and directly analyzes apps' native binaries.

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 [27] 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 [50] utilizes text-mining techniques and control-flow features to identify families of malicious apps. DroidSIFT [60] employs extracted dependency graphs to determine whether an app is benign or malicious, and the family of a malicious app.

Two approaches that automatically identify the family of Android malware using static analysis—Dendroid and DroidSIFT—are both limited, when compared to RevealDroid, in three key ways: (1) they have limited or no reflection features, (2) they have no native-code features and (3) 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. On the other hand, we evaluate RevealDroid on a dataset consisting of tens of thousands of more apps, and several hundred more malware families studied as part of the DroidSIFT paper. Additionally, DroidSIFT utilizes flow features, which are heavyweight to extract, as demonstrated in our experiments, and do not account for statically unresolvable or partially resolvable reflective calls.

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.

A third approach, DroidScribe [24], uses dynamic analysis and machine learning to identify the family to which a malicious app belongs. However, it does not determine whether an app is benign or malicious.

None of the aforementioned approaches extract native-code features by actually analyzing an app's native binaries. Furthermore, RevealDroid is the only Android malware detection and family-identification approach that combines machine learning with static analysis extraction of features based on Android API usage, reflection, and native code.

5 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. RevealDroid relies on features involving security-sensitive Android API calls; reflective calls categorized according to the degree to which invoked methods can be resolved; and invocations in native binaries to external functions (e.g., system calls or shared library calls) and functions within the binaries. We have compared RevealDroid with state-of-the-art tools for Android malware detection and family identification. For Android malware detection, RevealDroid obtains an 11%-60% superior accuracy compared to state-of-the-art tools. In the case of family identification, RevealDroid attains a 24%-70% higher classification rate. Our experiments showcase RevealDroid's high accuracy and efficiency (e.g., a 98% F1 score for 54,882 apps and an app extraction time of 90 seconds on average),

with particularly high accuracy under various obfuscations. We further compared RevealDroid to a state-of-the-art family-identification approach, demonstrating significantly higher accuracy—95% accuracy on a high quality Android malware family dataset—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 [7] and stealthily utilizing devices to mine cryptocurrency services [2]—in order to detect and identify the families of those malware. To enable replication of our results and improvement over RevealDroid, we make our RevealDroid prototype and data available online at [4].

ACKNOWLEDGMENTS

This work was supported in part by awards CCF-1252644, CNS-1629771, and CCF-1618132 from the National Science Foundation, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

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] Cisco 2014 annual security report. http://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html.
[4] RevealDroid. http://tiny.cc/revealdroid.
[5] Server-side polymorphic android applications. http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications.
[6] The Drebin Dataset. http://user.informatik.uni-goettingen.de/ darp/drebin/.
[7] Threat description trojan:android/oldboot.a. https://www.f-secure.com/v-descs/trojan_android_oldboot_a.shtml.
[8] VirusShare.com. http://www.virusshare.com/.
[9] VirusTotal. https://www.virustotal.com/.
[10] Quick Heal Annual Threat Report 2015. http://www.quickheal.co.in/resources/threat-reports, January 2015.
[11] 1.5. Stochastic Gradient Descent — scikit-learn 0.18.2 documentation. http://scikit-learn.org/stable/modules/sgd.html, 2017.
[12] 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.
[13] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. *Are Your Training Datasets Yet Relevant?*, pages 51–67. Springer International Publishing, Cham, 2015.
[14] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.
[15] M. Aly. Survey on multiclass classification methods. *Neural Netw*, pages 1–9, 2005.
[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] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. *ICSE*, 2015.
[19] 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.
[20] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.
[21] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In *Advances in neural information processing systems*, pages 409–415, 2001.
[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] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 252–261. IEEE, 2016.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[26] 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.

[27] 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.

[28] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. Technical Report UCI-ISR-16-2, Institute for Software Research, Irvine, California, 2016.

[29] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Technical Report GMU-CS-TR-2015-10, Department of CS, George Mason University, Fairfax, Virginia, 2015.

[30] 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.

[31] 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.

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

[33] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[34] 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.

[35] N. Japkowicz and S. Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449, 2002.

[36] J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan. The case for mobile forensics of private data leaks: Towards large-scale user-oriented privacy protection. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 6. ACM, 2013.

[37] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. The shellcoder's handbook. *Edycja polska. Helion, Gliwice*, 2004.

[38] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[39] B. Livshits, J. Whaley, and M. S. Lam. *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings*, chapter Reflection Analysis for Java, pages 139–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[40] 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.

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

[42] 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.

[43] 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.

[44] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *The Network and Distributed System Security Symposium 2016*, 2016.

[45] 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.

[46] 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.

[47] 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.

[48] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

[49] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[50] 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.

[51] 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.

[52] 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.

[53] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[54] 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.

[55] 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.

[56] 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.

[57] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 303–313, May 2015.

[58] 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.

[59] 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.

[60] 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.

[61] T. Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *ICML 2004: PROCEEDINGS OF THE TWENTY-FIRST INTERNATIONAL CONFERENCE ON MACHINE LEARNING. OMNIPRESS*, pages 919–926, 2004.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.