Mining Apps for Abnormal Usage of Sensitive Data

Vitalii Avdiienko[♣], Konstantin Kuznetsov[♣], Alessandra Gorla[♠], Andreas Zeller[♣],
Steven Arzt[†], Siegfried Rasthofer[†], and Eric Bodden[†‡]

[♣]*Saarland University, Saarbrücken, Germany*
[♠]*IMDEA Software Institute, Madrid, Spain*
[†]*TU Darmstadt, Darmstadt, Germany*
[‡]*Fraunhofer SIT, Darmstadt, Germany*

***Abstract*—What is it that makes an app malicious? One important factor is that malicious apps *treat sensitive data differently from benign apps*. To capture such differences, we mined 2,866 benign Android applications for their data flow from sensitive sources, and compare these flows against those found in malicious apps. We find that (a) for every sensitive source, the data ends up in a small number of typical sinks; (b) these sinks differ considerably between benign and malicious apps; (c) these differences can be used to flag malicious apps due to their abnormal data flow; and (d) malicious apps can be identified by their abnormal data flow alone, without requiring known malware samples. In our evaluation, our MUDFLOW prototype correctly identified 86.4% of all novel malware, and 90.1% of novel malware leaking sensitive data.**

I. Introduction

Most existing malware detectors work *retrospectively*, checking an unknown app against features and patterns known to be malicious. Such patterns can either be given explicitly ("Text messages must only be sent after user's consent"), or induced implicitly from samples of known malware ("This app contains code known to be part of the TDSS trojan."). If a novel app is sufficiently different from known malware, though, this retrospective detection can fail.

In this work, we thus conversely investigate the idea that, given access to a sufficiently large set of "benign" apps, one might be able to detect novel malware not by its similarity with respect to existing malware, but rather through its *dissimilarity with respect to those benign applications*. Checking for dissimilarity is different from checking for similarity, though, because in terms of functionality or code fragments, we already have lots of dissimilarity across benign applications themselves. As a measure for establishing similarity or dissimilarity with respect to the norm, we thus explore the *usage of sensitive data* in an app. Specifically, we apply *static taint analysis* on the 2,866 most popular Android apps from the Google Play Store to determine, for every sensitive data source, the sensitive APIs to which this data flows. We consider these flows to constitute the "normal" usage of sensitive data; as we assume the most popular Google Play Store apps to be benign, these flows also resemble "benign" usage.

As an example of such flows, consider the well known Android *Twitter* app. Table I shows its extracted data flows. We can see that, while the Twitter app accesses sensitive account information, it uses this information only to manage synchronization across multiple devices. Network information is being accessed (as part of the main functionality of the app), saved in logs, and passed on to other components.

TABLE I
FLOWS IN ANDROID TWITTER APP

AccountManager.get()	⇝ *ContentResolver.setSyncAutomatically()*
AccountManager.get()	⇝ *AccountManager.addOnAccountsUL()*
AccountManager.get()	⇝ *Activity.setResult()*
AccountManager.get()	⇝ *Log.w()*
AccountManager.getAccountsByType()	⇝ *ContentResolver.setSyncAutomatically()*
AccountManager.getAccountsByType()	⇝ *Activity.setResult()*
AccountManager.getAccountsByType()	⇝ *Log.w()*
Uri.getQueryParameter()	⇝ *Activity.startActivity()*
Uri.getQueryParameter()	⇝ *Activity.setResult()*
Uri.getQueryParameter()	⇝ *Activity.startActivityForResult()*
Uri.getQueryParameter()	⇝ *Log.w()*
SQLiteDatabase.query()	⇝ *Log.d()*
SQLiteOpenHelper.getReadableDatabase()	⇝ *Log.d()*
SQLiteOpenHelper.getWritableDatabase()	⇝ *Log.d()*

TABLE II
FLOWS IN COM.KEJI.DANTI604 MALWARE

TelephonyManager.getSubscriberId()	⇝ *URL.openConnection()*
TelephonyManager.getDeviceId()	⇝ *URL.openConnection()*

In contrast, consider the *com.keji.danti604* malware from the VirusShare database [28]. Table II shows the two flows in that application; they leak the subscriber and device ID to a Web server. Both these flows are very uncommon for benign applications; furthermore, *danti604* does not contain any of the flows that would normally come with apps that use the *TelephonyManager* for legitimate reasons. Thus, *danti604* is an anomaly—not only because it may be similar to known malware, but in particular because its data flows are *dissimilar* to flows found in benignware such as Twitter.

We have built a tool called MUDFLOW[1] which leverages the FLOWDROID [3] static analysis tool to determine such flows for all sensitive Android sources. MUDFLOW implements multiple *classifiers*, trained on the data flow of benign apps, to automatically flag apps with suspicious features. To the best of our knowledge, MUDFLOW is the first approach to massively mine application collections for patterns of "normal" data flow, and to use these mined patterns to detect malicious behavior.

The remainder of this paper is organized as follows. After introducing data flow and taint analysis in Section II, this paper presents the following contributions:

1) We present MUDFLOW, an approach to *mine, compare, and classify the data flow* in large sets of Android applications (Section III).
2) We apply MUDFLOW to the 2,866 most popular apps collected from the Google Play Store (Section IV). For each significant sensitive data source, we summarize

[1]MUDFLOW = Mining Unusual Data Flow

the *typical usage of this source* across these apps, and contrast it against the usage found in a collection of 15,338 malware apps.

3) We use the data-flow differences to *automatically identify malware based on its flow of sensitive data* (Section V). In particular, we train a model using data flows of benign apps only, and use it to detect novel malware even if no earlier malware samples are known.

Our experiments show that dissimilarity with benign apps, determined through data flow from sensitive sources, can be a significant factor in characterizing malware. In our experiment on a set of 10,552 malicious apps leaking sensitive data, MUDFLOW recognized 90.1% of the malware as such, with a false positive rate of 18.7%, which is remarkable given that MUDFLOW is not trained on malware samples.

After discussing threats to validity (Section VI) and relating to existing work (Section VII), we close with conclusion and future work (Section VIII).

II. Data Flow and Taint Analysis in Android

When installing an Android application, a user only sees a textual description of the alleged behavior of the app and a list of required permissions that the app needs in order to work. An app may, for instance, require access to the list of contacts on the mobile device but, unless clearly stated in the description, it is vague how the app is using this data. A benign messaging application may need to access the contact list to send messages, while a malicious application may collect the list of contacts and send them to a third-party server that gathers them for spamming. To detect this kind of behavior, one can use *taint* analysis, which is a particular instance of *data flow analysis*. In essence, given a *source* of information (e.g., the SQLite database containing the list of contacts) and a sink (e.g., an HTTP connection to a third party server), taint analysis can tell whether that information can flow to the undesired sink.

Information flow can be analyzed *statically,* i.e., the analysis would report whether there exist some paths in the program that may lead to this information flow, or *dynamically,* i.e., by reporting information flows that actually occur during a specific execution. In the first case, the analysis might report false positives, i.e., information flows that are not feasible in practice. In the second case, the analysis would likely miss information flows, namely if those flows occur in part of the behavior that was not observed.

Both static and dynamic data and information flow analysis techniques have been used to analyze Android applications, since several malicious applications use this platform to collect sensitive information of users. In fact, compared to the list of requested permissions, information flows can better describe the behavior of an Android application, as they can show *how* the application is using a specific piece of information.

Data and information flow analysis has been used to detect typical spyware behavior such as collecting sensitive information and sending it to third-party servers. However, flows of sensitive data are quite present in benign applications as well. The sole fact that an app has specific flows does not necessarily mean that the app is malicious.

A. Static Taint Analysis with FLOWDROID

Internally, MUDFLOW uses the static taint tracking tool FLOWDROID [3] to identify data flows in Android applications. We chose FLOWDROID because of its precise analysis and because it accurately models the lifecycle of Android applications and their interactions with the operating system. Unlike normal Java programs, Android apps are tightly coupled with their execution framework which can, for instance, pause and resume them at any time. Callbacks notify the application of system events, such as a low battery level or an incoming text message. FLOWDROID first analyzes the apps for all registered event handlers and components, and then creates dummy code that simulates these interactions with the operating system, causing the static analysis to take this possible runtime behavior correctly into account. FLOWDROID provides a *highly precise taint analysis* that is fully object-, flow-, and context-sensitive. High precision is required to reduce false positives during data flow analysis; it also reduces the amount of noise within the input data on which the machine-learning approach of MUDFLOW is later trained.

Listing 1 shows an example of how an Android application can obtain leaked data. The example reads the phone's unique identifier and sends it to the example telephone number "+1 234" using an SMS message. In real-world applications, the path between source (the call to *getDeviceId()* and sink (the call to *sendTextMessage()* can be substantially longer, and may include field and array accesses, polymorphic (library) method calls, conditionals, etc.

FLOWDROID uses an instantiation of the IFDS framework by Reps and Horwitz [24]. IFDS reduces data flow problems to reachability in a graph whose nodes represent combinations of possible facts about the program (e.g., "variable *devId* is tainted at line x in file f"). If one fact can be derived from another, the respective nodes are connected with an edge, causing the latter to be reachable from the former. If a certain fact at a sink is reachable from the node modeling a source, the analysis has discovered a leak from this source to this sink.

In the example in Listing 1, the node for variable *devId* at Line 4 forms the root of the graph. It is connected to the node that models "*a* is tainted" at Line 5 due to normal

```
1  void onCreate() {
2      TelephonyManager mgr = (TelephonyManager)
3          this.getSystemService(TELEPHONY_SERVICE);
4      String devId = mgr.getDeviceId();
5      String a = devId;
6      String str = prefix(a);
7      SmsManager sms = SmsManager.getDefault();
8      sms.sendTextMessage("+1 234", null, str, null, null);
9  }
10 String prefix(String s) {
11     return "DeviceId: " + s;
12 }
```

Listing 1. Android Data Leak Example

forward propagation. When processing a method invocation, the algorithm creates an edge into the callee, in the example causing the node representing "*s* is tainted" in Line 11 to become reachable. On method returns, the return values are mapped back into the caller, creating an edge to "*str* is tainted" in Line 8. The analysis then recognizes this line as a sink. The fact that parameter "*str* is tainted" is transitively reachable from the source, means that there exists a leak of the device id. The context-sensitivity of FLOWDROID makes it possible to distinguish different calls to the *prefix()* method with different parameter values. A context-insensitive analysis would act conservatively, marking as tainted all program variables capturing the return value of *prefix()*, even at those call sites that call *prefix()* with benign values.

FLOWDROID uses special hand-written summaries for calls to library methods for which no source code is available. Numerous optimizations make sure that the taint analysis scales to large apps such as the popular social network applications in the Google Play Store. Space limitations preclude us from explaining these optimizations further. We kindly point the interested reader to the original FLOWDROID paper [3].

B. Sensitive Sources

MUDFLOW leverages static taint analysis with FLOWDROID to *characterize* the behavior of individual apps with respect to their usage of sensitive data. The key idea is to identify data flows that *originate from sensitive sources.* We first describe our concept of a "sensitive source", followed by how we characterize the originating flows.

In Android, all sensitive data can be accessed through specific APIs—for instance, the Android method *getLastKnownLocation()* returns the user's current location. By tracing where this data flows to, it is possible to characterize the app's behavior. For this, though, we need to identify the APIs that access sensitive data. This is less easy than it might seem, because several Android APIs are not or hardly documented; furthermore, lists crafted by researchers get outdated with every new Android version.

We therefore leverage the SUSI technique by Rasthofer et al. [23], which automatically classifies all methods in the whole Android API as a source, sink, or neither, using a small hand-annotated fraction of an Android API to train a classifier. Beside providing a list of APIs that access sensitive sources, SUSI also provides a *categorization* of these APIs, listed in Table III. The method *getLastKnownLocation()*, for instance, falls into the LOCATION_INFORMATION category.

In addition to the originally published SUSI categories, we created three new categories to further break down the behavior of Android apps, marked with (*) in Table III.

Sensitive Resources. During our investigation, we found that Android apps also access sensitive resources through *content providers*—external components that resolve appropriate resource identifiers; a *CalendarContract* provider, for instance, can access calendar data. All these flows start from the *android.content.ContentResolver* API, which gets the desired resource identifier as an argument. SUSI assigns the *ContentResolver* API to NO_CATEGORY because the same API can be used to access all sorts of resources, sensitive or non-sensitive. In 2012, however, Au et al. [4] published a list of sensitive resource schemes as used in Android. We therefore conducted an additional step of static analysis: Using SOOT [16], we extracted *android.net.URI* usages from all Android applications and assigned them to the appropriate SUSI source categories. Any resource usage not in the list would be classified into the CONTENT_RESOLVER sensitive source category.

TABLE III
SUSI API CATEGORIES OF SENSITIVE SOURCES AND SINKS

Sources	Sinks	Shared
• HARDWARE_INFO	• PHONE_CONNECTION	• AUDIO
• UNIQUE_IDENTIFIER	• VOIP	• SMS_MMS
• LOCATION_INFORMATION	• PHONE_STATE	• CONTACT_INFORMATION
• NETWORK_INFORMATION	• EMAIL	• CALENDAR_INFORMATION
• ACCOUNT_INFORMATION	• BLUETOOTH	• SYSTEM_SETTINGS
• EMAIL_INFORMATION	• ACCOUNT_SETTINGS	• IMAGE
• FILE_INFORMATION	• VIDEO	• BROWSER_INFORMATION
• BLUETOOTH_INFORMATION	• SYNCHRONIZATION_DATA	• NFC
• VOIP_INFORMATION	• NETWORK	
• DATABASE_INFORMATION	• EMAIL_SETTINGS	
• PHONE_INFORMATION	• FILE	
• CONTENT_RESOLVER (*)	• LOG	
• NO_SENSITIVE_SOURCE (*)	• INTENT (*)	
	• NO_SENSITIVE_SINK (*)	

(*) New category, see text

Intents. We found that several flows end in communication between multiple app components ("Intent" in Android parlance). As of now, the precise identification of activities launched by intents, as well as identification of flows across intents is out of scope for this paper; our main objective, namely classification of malware, is still fulfilled despite this imprecision. To mark flows into intents, we introduced a special category INTENT for this kind of potentially sensitive sink.

Non-sensitive Sources and Sinks. Almost all applications in the Google Play Store access sensitive sources. However, the data accessed does not necessarily end up in a sensitive sink—wallpaper apps, for instance, access the user's images as sensitive sources, but the user's display is not a sensitive sink. To leverage these accesses, every source used that does not flow into a sensitive sink is modeled as a flow from that source "to" the special category NO_SENSITIVE_SINK. Similarly, we modeled flow that does not start from sensitive source and ends up in a sensitive sink from the special category NO_SENSITIVE_SOURCE.

III. Mining and Classifying Flows

A. Extracting Flows

Applied on a single *app*, MUDFLOW uses FLOWDROID to extract all data flows from *all sensitive data sources* to *all sensitive data sinks.* The result is a set of pairs that characterizes the sensitive flows in the application—and thus the application itself:

$$Flows(app) = \{source_1 \rightsquigarrow sink_1, source_2 \rightsquigarrow sink_2, \dots\}$$

where each $source_i$ and $sink_i$ is a sensitive Android API method. (Again, "sensitive" means that it falls into one of the SUSI categories listed in Table III). As examples of such flows, consider Table I and Table II discussed in Section I.

TABLE IV
FLOWS IN ANDROID TWITTER APP, BY SUSI CATEGORY

ACCOUNT_INFORMATION	⇝ SYNCHRONIZATION_DATA
ACCOUNT_INFORMATION	⇝ ACCOUNT_SETTINGS
ACCOUNT_INFORMATION	⇝ INTENT
ACCOUNT_INFORMATION	⇝ LOG
NETWORK_INFORMATION	⇝ INTENT
NETWORK_INFORMATION	⇝ LOG
DATABASE_INFORMATION	⇝ LOG

B. Flow Granularity

By default, each source and sink contains the full method name and signature. For the sake of obtaining a coarser granularity, this information can be shortened, allowing for multiple sources and sinks to be aggregated. MUDFLOW supports the following three granularity levels, from finest to most coarse:

Method. This is the full method signature—for instance, *LocationManager.requestLocationUpdates(...)*.

Class. Considering only the class name (*LocationManager*) allows to express flows between classes rather than methods. This treats all methods of a class uniformly.

Category. Considering only the SUSI category of the API (LOCATION_INFORMATION) allows to express flows between categories. This is the coarsest way of expressing flows, yet one that could be made accessible to end users. Table IV shows the flows in the Android Twitter app at the category level. Here, it is indeed easy to spot how the sensitive data is being used.

C. Automatic Classification

As shown above, the flow within malicious apps may differ significantly from the flow within benign apps. MUDFLOW leverages such differences to *automatically classify* novel apps whether they are malicious or not. While most malware detection is *retrospective* in nature—checking apps against patterns found in known malware—, MUDFLOW is able to compare a new app *against benignware only*, and check whether it contains abnormal flows with respect to this set. This allows MUDFLOW to detect malware as abnormal even if the specific attack is the first of its kind.

The MUDFLOW malware classification takes a set s of benign apps (say, all apps from an app store) and then works in three steps, detailed below.

1) Outlier Detection in Category: In its first step, illustrated in Figure 1, MUDFLOW identifies which apps have unusual flows within each category using s as a ground set. Specifically, for a category c, MUDFLOW:

- takes all apps in s that use at least one API from c,
- extracts all flows within these apps (that is, flows originating from sources in c as well as flows originating from other sources). To compute the flows in this step, MUDFLOW uses the API methods classified as sources and sinks,
- assigns 0.5 as a weight to all flows that lead to LOG, since these flows are highly prevalent in apps, but are less harmful, since starting from Android 4.1 LOG files can only be accessed by diagnostic and administrative tools. Finally, it
- determines outlierness score of apps considering these flows as features by using ORCA method [5] with s as a reference set.

The resulting model can then be used to assess a novel (potentially malicious) app a which also uses at least one API from c. For this purpose, MUDFLOW extracts the flows from a, and computes the *distance* between a and its $k-$nearest neighbors from the set s. For this step MUDFLOW resorts to the ORCA outlier detector, and considers, by default, the 5 nearest neighbors of a sample a. To measure the dissimilarity between samples, MUDFLOW uses the *weighted Jaccard* distance metric, since it is more suitable for data with a huge number of features, as in this case.

The resulting distance serves as an *outlier score:* The higher a's distance, the less "normal" are its features—resulting, according to our hypothesis, in a higher likelihood of being malicious.

2) Aggregating Scores: In its second step, MUDFLOW applies the above outlier detection on all SUSI source categories, resulting in a single outlier detector for each sensitive source. We can now take an app a and determine its scores for each category. As illustrated in Figure 2, we thus obtain a vector of distances, telling for each sensitive category how much a deviates from the norm. If an application does not use the APIs of the appropriate source, its outlierness score is set to zero. We have dubbed this vector a "maliciogram", as it may guide investigators towards potential issues, or inform end users about potential risks, allowing them to focus on the categories they care about most.

However, the quality of SUSI categories depends on how similar the benign applications are within the particular category. To take this fact into account, MUDFLOW assigns a

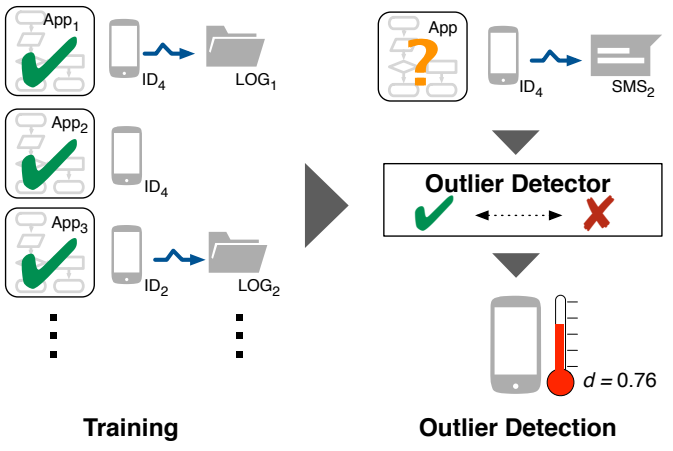

Fig. 1. Per-category outlier detection. For each SUSI category such as UNIQUE_IDENTIFIER (shortened to "ID"), MUDFLOW selects apps that use APIs of that category as source and uses their flows as features. It then takes a new unknown app, and determines its outlierness score with respect to the "normal" apps. The higher the score, the less "normal" the app behaves inside a particular SUSI category.

weight to each category, emphasizing its importance. When the mean value of scores within a category is high, it means that all benign apps considerably differ from each other. As a consequence of this induced noise, we have less confidence that an outlier would be malicious. For this reason we give a lower weight to such categories; if, in contrast, the mean of score values is low, the category will have higher weight. More precisely, we use $exp(1/mean)$ as the formula to weigh categories.

3) Overall Classification: In the third and last step, MUDFLOW leverages *one-class classification* to determine whether an app a overall would be considered malicious or not. This decision would be based on the individual scores in each category, as determined in the past two steps.

The process is illustrated in Figure 3:

- MUDFLOW constructs a training set based on benign applications. Their score vectors ("maliciograms") across the SUSI source categories are used as features.
- It trains a ν-SVM one-class classifier [7], which is commonly used for novelty detection purposes.
- After training, it uses the ν-SVM for *classification* of a novel app a, based on its score vector, into either "likely benign" or "likely malicious".

The result is a fully automatic warning flag for any novel app, which when raised, would trigger further investigation such as additional analysis or testing. This investigation would be guided by the individual category scores as reported by MUDFLOW ("How does this app deviate from the norm?") as well as the individual flows detected within the app ("Which flows in this app are abnormal?").

D. Advertisement Frameworks

Many Android apps generate revenue through advertisements, which are delivered through specific *advertising frameworks*. These frameworks access sensitive sources such as account data to deliver personalized advertisements; However, these flows are separate from the actual app code. As advertising frameworks are frequently used, their flows thus become "normal" and make malicious flows harder to detect. Furthermore, malicious software may use an advertisement framework to motivate and mask its malicious flows.

Assuming that advertisement frameworks are to be trusted, MUDFLOW therefore ignores all sensitive flows taking place within advertisement frameworks only, allowing it to focus on the actual app code. Table V shows the list of frequently used frameworks whose flows are excluded in MUDFLOW. Currently

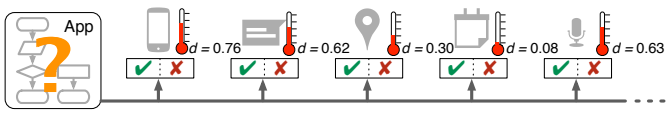

Fig. 2. Aggregating probabilities across API usage. Given an app, for each SUSI category, we use the approach from Figure 1 to determine the distance of the app with respect to the benign training set. The resulting vector of scores ("maliciogram") tells how abnormal the app is in each category.

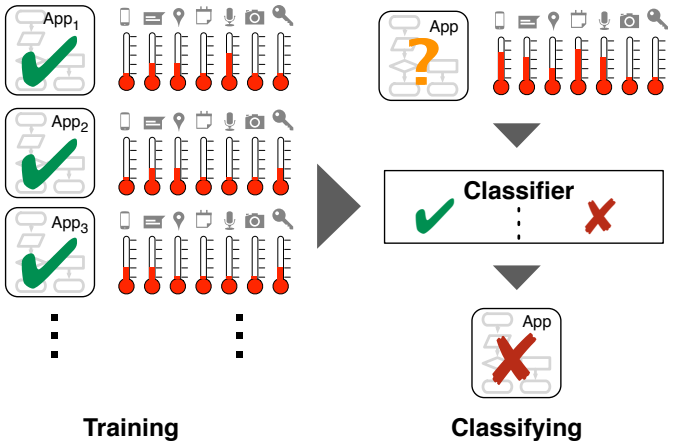

Fig. 3. Classifying apps across multiple categories. For each "benign" app in the Google Play store, we determine its vector of probabilities of being an outlier in each SUSI category (Figure 2). A one-class classifier trained from these vectors can label an unknown app as "likely benign" if it is normal across all categories, or "likely malicious" instead.

TABLE V
AD FRAMEWORKS WHOSE FLOWS ARE EXCLUDED

com.admob.android	com.adsdk.sdk
com.adsmogo	com.aduwant.ads
com.applift.playads	com.google.ads
com.inneractive.api.ads	com.mopub.mobileads
com.revmob.ads	com.smartadserver.android
com.swelen.ads	de.selfadservice

MUDFLOW is unable to detect such advertisement frameworks in presence of code obfuscation. This problem is in the scope of our future work and will be discussed in Section VIII.

IV. APPS AND THEIR FLOW

To evaluate MUDFLOW, we used two sets of "benign" and "malicious" apps for training and classifying. Let us describe these datasets, as well as their characteristic flows.

A. Apps Mined

Benign apps. As source for "benign" apps, we used the Google Play Store, the most popular app store for Android. For each of the 30 app categories in the store, we downloaded the top 100 most popular free applications as of March 1, 2014. As not all categories had 100 such entries, this gave us a total of 2,950 apps as our initial "benign" dataset.

Malicious apps. We used two sources for "malicious" apps:

- 1,260 malware apps from the Genome project [30]. This is the dataset already used in the CHABADA project [12].
- The full set of 24,317 malicious applications from the VirusShare database [28] as of 24 March 2014.

Our initial "malicious" set thus contained a total of 25,577 apps.

B. Analysis Settings

Running a precise static taint analysis on real-world applications is not without challenges. In favor of a faster analysis

or the ability to analyze a larger application which would otherwise not fit in memory, we used the following FLOWDROID settings [3]:

- *No flow across intents.* Android apps use special components, called *intents,* to implement messaging between components, in particular to start activities or provide services in the background. We do not track flows across intents; when sensitive data is sent to an intent, the flow is marked with the INTENT category as a sink;
- *Explicit flow only.* Our static taint analysis settings do not consider conditionals controlling specific flows, nor the flows leading to these conditionals. This is in contrast to *information flow analysis,* which also takes such implicit flow into account;
- *Flow-insensitive alias search.* Making the alias search flow-insensitive may generate false positives, but greatly reduces runtime for large applications;
- *Maximum access path length of 3,* again possibly reducing precision with respect to the default setting of 5;
- *No-layout mode,* ignoring Android GUI components, such as input fields, as data flow sources;
- *No static fields,* ignoring the tracking of static fields.

All these choices sacrifice some amount of precision for speed and memory. As a result, the list of flows determined by MUDFLOW can have false positives (flows that are infeasible during executions) as well as false negatives (missing flows that actually might be possible); but still, FLOWDROID is much more precise than a basic object- or context-insensitive data flow analysis. As ever when applying precise static analysis on real-world programs with finite time and resources, striking a good balance between false positives and false negatives is an important challenge. Let us remind at this point, that our goal is *to detect anomalies,* not to prove the presence or absence of flows; and thus, we can tolerate imprecision as long as the overall results are fine.

Still, let us state what "finite time and resources" mean in our setting, and why compromises are badly needed. The main machine we used to run MUDFLOW was a compute server with 730 GB of RAM and 64 Intel Xeon CPU cores, far exceeding the standard memory sizes of today's personal computers. Even with all the compromises listed above, the server sometimes used all its memory, running on all cores for more than 24 hours to analyze *one single* Android app, as shown in Figure 4. Overall, we had this machine run for two months without interruption to extract data flows from Android applications.

C. Analysis Results

A small proportion of downloaded apps proved to be a challenge for precise taint analysis.

Of the 2,950 "benign" apps, 84 (3%) were not analyzable: 16 apps exceeded the RAM limit of 730 GB or the 24-hour timeout, and 68 apps caused a SOOT exception while transforming DEX bytecode to JIMPLE representation. Of the "malicious" apps, 10,239 (40%) were not analyzable because of corrupted or incomplete APKs; most frequently, the required

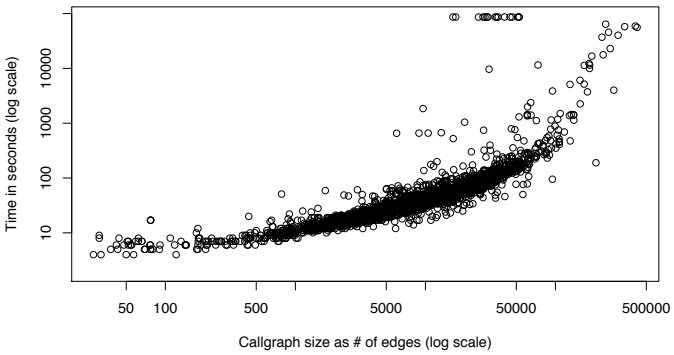

Fig. 4. Analysis time of benign applications with respect to their call graph sizes. All times obtained on an Intel 64-core machine with 730 GB RAM.

Android manifest was missing. We also removed all such non-analyzable apps from our dataset. This resulted in final datasets of 2,866 "benign" apps and 15,338 "malicious" apps.

D. Data Flow in Benign Apps

Table VI summarizes the data flows detected in our set of "benign" apps. Most interesting, 68.3% of all accesses to sensitive data do not end in a sensitive sink. Across the sensitive sinks, we detected 43,371 different data flows, i.e., 43,371 distinct pairs of code locations accessing a sensitive source API and a sensitive sink API, linked by data flow between them. The most important source is DATABASE_INFORMATION, followed by CALENDAR_INFORMATION, NETWORK_INFORMATION, and LOCATION_INFORMATION. This reflects what most Android apps are doing: interacting with external services, using information maintained in their own databases.

As it comes to the least frequently used sources, we find results that reflect programming practices in Android. The source EMAIL shows no flows at all, which might be surprising, considering the number of apps that handle phone calls or e-mails. This is because most e-mail accesses take place via IMAP and POP protocols and thus belong to the NETWORK_INFORMATION source category. The sources SYSTEM_SETTINGS and BROWSER_INFORMATION rarely ever end in sensitive sinks.

The most important sinks are LOG and INTENT, which make up more than 94% of all sinks in sensitive flows. As discussed in Section IV-B, the INTENT category means that the data was used by another activity in the app, a flow we currently cannot analyze; LOG, however, is a true sink, but it is less harmful, as starting from Android 4.1 log files can only be accessed by diagnostic and administrative tools.

The data set coming with this paper contains detailed information on all flows, showing the exact flows between APIs for all of the benign applications.

> *In "benign" apps, 94% of all sensitive data flows are to logging and Inter-Process-Communications (i.e. intents).*

E. Data Flow in Malicious Apps

Table VII summarizes the data flows detected in our set of "malicious" apps, showing similarities, but also important

TABLE VI
DATA FLOWS IN BENIGN APPLICATIONS, BY SUSI CATEGORIES

Source	LOG	INTENT	NETWORK	FILE	SYSTEM_ SETTINGS	ACCOUNT_ SETTINGS	SMS_MMS	AUDIO	NFC	SYNCHRO NIZATION _DATA	CALENDAR _INFORMA TION	LOCATION _INFORMA TION	BLUETOOTH	NO_SENSI TIVE_SINK	Total	
DATABASE_INFORMATION	6643	6174	127	98	108	43	0	61	0	0	0	3	0	18046	**31303**	9,2%
CALENDAR_INFORMATION	3529	7199	244	305	189	17	0	12	15	0	74	3	0	25208	**36795**	10,9%
NETWORK_INFORMATION	4774	1529	122	70	53	37	85	24	0	0	0	0	0	27809	**34503**	10,2%
LOCATION_INFORMATION	3758	1242	37	22	12	0	0	15	2	0	0	28	0	14187	**19303**	5,7%
CONTENT_RESOLVER	1769	823	74	24	22	2	5	12	4	0	2	1	0	3572	**6310**	1,9%
UNIQUE_IDENTIFIER	796	514	17	1	2	3	1	5	0	0	0	0	0	1819	**3158**	0,9%
ACCOUNT_INFORMATION	477	264	18	2	6	266	0	2	0	80	0	0	0	1032	**2147**	0,6%
CONTACT_INFORMATION	390	92	0	0	0	5	0	0	0	0	0	0	0	21	**508**	0,2%
FILE_INFORMATION	250	136	4	14	5	0	0	2	0	0	0	0	0	921	**1332**	0,4%
NFC	53	129	7	6	0	0	0	0	73	0	0	0	0	165	**433**	0,1%
BLUETOOTH_INFORMATION	153	23	0	0	1	0	0	2	0	0	0	0	0	609	**788**	0,2%
SMS_MMS	42	0	0	0	0	0	81	0	0	0	0	0	0	156	**279**	0,1%
SYNCHRONIZATION_DATA	19	0	0	0	0	0	0	0	0	10	0	0	0	137	**166**	0,0%
IMAGE	15	0	0	0	0	0	0	0	0	0	0	0	0	58	**73**	0,0%
BROWSER_INFORMATION	5	3	0	0	0	0	0	0	0	0	0	0	0	1	**9**	0,0%
SYSTEM_SETTINGS	5	0	0	0	0	0	0	0	0	0	0	0	0	26	**31**	0,0%
NO_SENSITIVE_SOURCE	147936	45028	4092	2626	355	39	61	1235	1	49	22	26	2	0	**201472**	59,5%
Total	**170614**	**63156**	**4742**	**3168**	**753**	**412**	**233**	**1370**	**95**	**139**	**98**	**61**	**2**	**93767**	**338610**	
	50,4%	18,7%	1,4%	0,9%	0,2%	0,1%	0,1%	0,4%	0,0%	0,0%	0,0%	0,0%	0,0%	27,7%		

TABLE VII
DATA FLOWS IN OUR SET OF MALICIOUS APPLICATIONS, BY SUSI CATEGORIES

Source	LOG	INTENT	NETWORK	FILE	SYSTEM_ SETTINGS	ACCOUNT _SETTINGS	SMS_MMS	AUDIO	NFC	SYNCHRO NIZATION _DATA	CALENDAR _INFORMA TION	LOCATION _INFORMA TION	BLUETOOTH	NO_SENSI TIVE_SINK	Total	
DATABASE_INFORMATION	21006	25363	402	332	32	4	108	212	0	0	0	0	0	71906	**119365**	8,0%
CALENDAR_INFORMATION	5649	3599	94	74	61	0	4	66	0	0	0	0	0	62683	**72230**	4,8%
NETWORK_INFORMATION	29930	3293	5174	373	73	3	11802	116	0	0	0	0	0	261739	**312503**	20,9%
LOCATION_INFORMATION	22548	1831	1242	6	0	0	68	14	0	0	0	6	0	80152	**105867**	7,1%
CONTENT_RESOLVER	9897	3312	859	79	29	0	36	89	0	0	0	0	0	10176	**24477**	1,6%
UNIQUE_IDENTIFIER	10981	1179	6608	240	1	0	579	17	0	0	0	0	0	49279	**68884**	4,6%
ACCOUNT_INFORMATION	86	114	5	0	2	87	0	2	0	6	0	0	0	1246	**1548**	0,1%
CONTACT_INFORMATION	412	276	0	1	11	0	581	13	0	0	0	0	0	29	**1323**	0,1%
FILE_INFORMATION	706	255	0	0	12	0	0	32	0	0	0	0	0	1791	**2796**	0,2%
NFC	36	7	2	0	0	0	4	0	11	0	0	0	0	95	**155**	0,0%
BLUETOOTH_INFORMATION	79	222	10	6	0	0	0	21	0	0	0	0	1	1968	**2307**	0,2%
SMS_MMS	33	18	0	0	0	0	117	0	0	0	0	0	0	185	**353**	0,0%
SYNCHRONIZATION_DATA	3	1	0	0	0	0	0	0	0	0	0	0	0	30	**34**	0,0%
IMAGE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0,0%
BROWSER_INFORMATION	15	12	0	0	0	0	0	0	0	0	0	0	0	173	**200**	0,0%
SYSTEM_SETTINGS	0	3	0	0	0	0	0	0	0	0	0	0	0	38	**41**	0,0%
NO_SENSITIVE_SOURCE	505584	222320	25897	23293	125	160	331	4362	0	137	0	313	1	0	**782523**	52,4%
Total	**606965**	**261805**	**40293**	**24404**	**346**	**254**	**13630**	**4944**	**11**	**143**	**0**	**319**	**2**	**541490**	**1494606**	
	40,6%	17,5%	2,7%	1,6%	0,0%	0,0%	0,9%	0,3%	0,0%	0,0%	0,0%	0,0%	0,0%	36,2%		

differences to the "benign" apps from Table VI. The most important source here is NETWORK_INFORMATION, almost twice as prevalent as in "benign" apps. CALENDAR_INFORMATION is accessed far less frequently as a sensitive source, as is (to our surprise) ACCOUNT_INFORMATION.

In sinks, we also see important differences. Most striking is the SMS_MMS sink, more than 77 times as prevalent as in our "benign" apps. This reflects the common attack of stealthily sending SMS messages to premium numbers, allowing the owner of these numbers to earn money from the victim. As the flows to SMS_MMS indicate, the malicious apps also include sensitive data such as UNIQUE_IDENTIFIER and CONTACT_INFORMATION in their messages, as well as NETWORK_INFORMATION such as network MAC addresses or SIM card information.

Given that 25% of our "malicious" apps use SMS as a sink, whereas this is the case for only 1% of the "benign" apps, a simple check for the ability to send SMS messages would easily weed out 25% of malicious apps, with a precision of 99%. Note, however, that several of such simple checks may bring conflicting classifications; also, while our "benign" set is representative in that it encompasses the most popular apps, our "malicious" set is in no way representative for malware actually prevalent in the wild, or the types of attacks actually conducted. In that sense, Table VII serves as descriptive statistics of the dataset we use for the evaluation of MUDFLOW.

Our set of "malicious" apps differs from the "benign" apps in terms of sources, sinks, and flows.

V. EVALUATION

With our datasets of "benign" and "malicious" apps available, we are now able to evaluate the classifiers in MUDFLOW. This section reports our results.

A. Detecting Overall Outliers

In our first experiment, we evaluated the full MUDFLOW classifier as described in Section III-C. Using 10-fold cross-validation, we repeated the following ten times:

- We trained the classifier on the score vectors from a random 90% of the "benign" dataset.
- The remaining 10% form the testing dataset, as well as the whole "malicious" dataset.

The average results with the ν-SVM configured as $\nu = 0.15$ are as follows:

True positives (malware recognized as such): 86.4%
True negatives (benignware recognized as such): 81.3%
Accuracy (apps correctly classified): 83.8%

MUDFLOW recognizes 86.4% of malware as such, with a false positive rate of 18.7%.

As we are most interested in apps that access and send sensitive data, we ran a second evaluation on the subset of 10,552 "malicious" apps that have at least one flow from a sensitive source to a sensitive sink (i.e., malware leaking sensitive data). For this "sensitive" subset, we get the following results:

True positives (malware recognized as such): 90.1%
True negatives (benignware recognized as such): 81.3%
Accuracy (apps correctly classified): 86.0%

MUDFLOW recognizes 90.1% of malware leaking sensitive data as such, with a false positive rate of 18.7%.

Again, all these numbers come from one-class classification; that is, no existing malware is used for training.

B. Repackaged Apps

We have seen that MUDFLOW shows good classification results on our "malicious" samples, because they are apparently sufficiently different from the "benign" apps used for training. But what happens if we take "benign" apps and *repackage* them to include malicious behavior on top of their regular functionality? These would include all the behavior (and flows) of the originals, plus additional flows and sources from the added malware components.

To this end, in our Genome set we identified the repackaged apps, which in essence are those apps that come with package names matching existing apps from the Google Play Store, but include malicious payloads. This gave us 96 distinct apps, which we verified manually to be easily confounded with original benign apps. Of these 96 apps, MUDFLOW classified 93 correctly as malicious, and three falsely as benign.

In a sample of 96 repackaged apps (benign apps with added malicious behavior), MUDFLOW identified 97.6% correctly as malicious.

As expected, the repackaged apps included flows that would be unusual for benign apps. The repackaged version of *ES File Manager,* for instance, would include flows from device ID or subscriber ID to the Web; as these flows would normally only occur in advertisement libraries (Section III-D), their presence outside of these libraries would immediately flag the application as an anomaly.

C. Alternate Features

An important question regarding MUDFLOW is whether the individual features in our approach are all necessary, and whether the expensive static analysis could be replaced by something simpler. For this purpose, we repeated the evaluation of Section V-A using different features. Notably, we checked the classification results using *source methods alone* as features, as these would not require complex static analysis. As summarized in Table VIII, *data flow between methods*, as implemented in MUDFLOW and evaluated in Section V-A, shows the best performance across all metrics.

TABLE VIII
EFFECTIVENESS OF MUDFLOW USING DIFFERENT FEATURES.

Features	Malicious set	True positives	True negatives	Accuracy
Source methods	all	81.7%	82.5%	82.1%
Sink methods	all	71.0%	83.9%	77.2%
Flow between classes	all	82.7%	79.7%	81.2%
	sensitive	87.7%	79.9%	83.7%
Flow between methods	all	86.4%	81.3%	83.8%
	sensitive	90.1%	81.3%	86.0%

If one wants to save analysis time, source methods alone may produce sufficient performance; as shown in Table VIII, this results in a true positive rate of 81.7% (rather than 86.4% with flows), for all apps; still showing a low false positive rate. A classification using source methods alone, though, is easy to fool, as an attacker could repackage an existing app, reuse existing sources and divert the flow to other sinks. Yet, in our experiment on repackaged apps, MUDFLOW performed particularly well (Section V-B).

Data flow from sensitive sources, as used in MUDFLOW, show the best classification results.

D. Per-Category Outlier Detection

As described in Section III-C1, the overall MUDFLOW classifier depends on individual per-category outlier detectors, computing outlier scores for each category. In this section, let us assess the accuracy of these detectors. For this purpose, we computed the *area under the ROC curve* for each outlier detector; this value is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. An area of 1.0 thus represents a perfect test, an area of 0.5 represents a worthless test.

Table IX shows the size of the categories, as well as the area under ROC curve numbers. We see that across all categories, the individual classifiers perform very well. The highest accuracy (1.00) is achieved for apps that use CONTACT_INFORMATION, where all malicious apps are higher ranked than benign apps, a result partially explained by the imbalance between benign and malicious apps. For BLUETOOTH_INFORMATION and other categories, the outlier detectors work well for balanced sets.

Outlier scores in individual categories are good predictors of malicious behavior.

Despite their accuracy, keep in mind that the individual classifiers only work for applications that also use the appropriate category as a source. Furthermore, a single app may get a low "benign" score in some categories, but a high "malicious" score in others. This is why MUDFLOW aggregates these scores to provide an overall classification.

E. Learning from Malware

In our final experiment, we changed the setting in Section III-C1 from the one-class ν-SVM to a *two-class SVM,*

TABLE IX
PERFORMANCE OF PER-CATEGORY OUTLIER DETECTORS

Category	Benign apps	Malicious apps	Area under ROC curve
NETWORK_INFORMATION	2,254	14,086	0.94
DATABASE_INFORMATION	1,525	7,468	0.95
CALENDAR_INFORMATION	1,275	6,928	0.93
LOCATION_INFORMATION	1,168	6,410	0.90
UNIQUE_IDENTIFIER	972	11,456	0.93
FILE_INFORMATION	563	1,358	0.93
CONTENT_RESOLVER	1,067	3,771	0.95
ACCOUNT_INFORMATION	320	542	0.53
BLUETOOTH_INFORMATION	183	283	0.96
SYNCHRONIZATION_DATA	80	19	0.91
NFC	61	20	0.92
CONTACT_INFORMATION	32	404	1.00
BROWSER_INFORMATION	5	193	0.92
SYSTEM_SETTINGS	4	25	(too few samples)
SMS_MMS	3	5	(too few samples)
IMAGE	2	0	(too few samples)

which would be trained with both "benign" and "malicious" apps, thus exploiting the existence of known malware and their respective flows. Again, we used ten-fold cross validation, each time training the SVM with the "malicious" apps not used in the testing set. This setting increases the classification accuracy to 95% (97% for "sensitive" malware)—that is, an even larger fraction of the "benign" apps and "malicious" apps would be correctly classified.

By also learning flows from known malware, MUDFLOW accuracy increases to 95% for all malware, and to 97% for malware leaking sensitive data.

This higher detection rate, however, is due to our malware set being *self-similar,* i.e., incorporating the same attack schemes again and again. These attack schemes result in recurring data flows, which can be exploited by MUDFLOW; in practice, though, malware not only shares similar attack schemes, but even shares code implementing these attacks. Hence, to determine *similarity* with known malware, we see data flow as only one feature besides established effective techniques such as code signatures, APIs used, and others, all of which could show similar or better detection rates. As a *dissimilarity* measure comparing against benignware, though, our results make data flow a promising feature to detect unusual behavior.

"Normal" and "abnormal" data flow can be an important factor in malware detection.

VI. THREATS TO VALIDITY

The main threat to validity in our work is *external validity,* asking how our results generalize to alternate settings. While our set of "benign" apps represents the most popular Google Play Store apps across all categories, our set of "malicious" apps stems from collections of malware where each app at some point has been found and identified as malicious, but we do not know whether it has ever caused damage before being detected. We also do not know which Android malware is currently in circulation; and we do not know its main attack vectors, its code features, and its possible obfuscation features. Our detection results should thus be seen as a result on a publicly available benchmark, and may not necessarily generalize.

A second threat to external validity are deficiencies of our static analysis; as discussed in Section IV-B, our analysis may report flows that are infeasible, as well as miss flows that are feasible (notably implicit flow and flow across components or storages). This impacts our summaries as shown in Table VI and Table VII. Our classification mechanisms, though, would be expected to include misclassifications, and thus would be impaired, but not threatened by such noise.

VII. RELATED WORK

MUDFLOW mines the usage of sensitive data in Android apps, and uses this information to detect malicious applications. MUDFLOW is thus related to the many existing techniques that leverage taint analysis to detect information leaks, to Android malware detection techniques, and to the empirical studies on Android stores.

A. Information Flow Analysis for Android

As mobile devices are a particularly rich store of sensitive data, it is not a surprise that much of the mobile security research work has developed taint analysis techniques to detect information leaks. Among the research works that leverage *dynamic* taint analysis to detect information leaks, TAINTDROID is the de-facto state of the art tool for Android applications [8]. Thanks to an efficient instrumentation of the Android execution environment, TAINTDROID can report, without any false positives, information leaks in apps even when they involve native code.

Dynamic taint analysis has the obvious limitation of reporting only on what has been observed during a limited set of executions. At the opposite side, *static* taint tracking tools report any information leak that *may* occur at runtime. The FLOWDROID tool, described in Section II, employs a highly precise static control and data flow analysis of Android apps to report both explicit and implicit information flows [3]. Other static taint tracking tools work in a similar fashion, but miss several possible information flows since they implement less precise data-flow analyses [29], [11].

Other techniques focus on detecting information flows involving inter-applications communication, and can thus detect when multiple applications can act together to leak sensitive information [15], [17].

MUDFLOW is orthogonal to all these techniques. In fact, while all these techniques can detect whether there is any information flow in Android applications, they cannot tell whether such behavior is likely to be malicious or not. On the other hand, MUDFLOW has no ability to detect information flows on its own, and therefore needs tools such as FLOWDROID to collect information regarding the behavior of apps.

B. Android Malware Detection

As for any other software platform, malware detection received a lot of attention in Android. Several techniques

focus on detecting whether the *claimed* behavior matches the *actual* behavior of the application. Some of these techniques use the textual description to understand what an application should do [12], [20], [22], while others analyze the text associated to GUI elements [14]. MUDFLOW instead, only requires the binary of the application, and therefore can easily be adopted to classify also applications that come without textual descriptions.

Moreover, all the aforementioned techniques either look at the declared permission in the manifest file or at API calls to "critical" functionalities of the Android framework. MUDFLOW, instead, uses sensitive data flows as features to describe the behavior of an application. Thus, instead of knowing whether an application accesses the contact list, it knows what the application does with the contact list. Thanks to more precise features, MUDFLOW can consequently provide better malware detection abilities.

MUDFLOW is not the first work that uses machine learning techniques to detect malicious Android applications. On the other hand, it is the only one, together with CHABADA, to train the model only on benign applications [12]. Other techniques such as MAST and Drebin, instead, train the classifier only on samples of malware, and can therefore be very effective at detecting other samples of similar malware [6], [2]. Differently from MUDFLOW, though, they are quite ineffective at detecting new types of malware. Similarly, techniques that implement static or dynamic analyses to detect known malware features are complementary to MUDFLOW, as they are designed to detect *known* malware [21], [18], [9], [10].

C. Mining of Android Apps on Markets

Together with a malware classifier for Android, this paper presents a novel study of the typical information flows of Android applications that are popular on the Google Play market. Other researchers used Android markets for empirical studies. Harman et al. mined the Blackberry app store to identify correlations between user rating and ranking of applications [13]. Stevens et al. analyzed 10,000 free apps from popular Android markets and found a significant correlation between the popularity of a permission and the number of times it is misused [27]. Ruiz et al., instead, study the prevalence of multiple ad libraries in Android apps [25], and Nagappan et al. analyze the software reuse in the Android mobile app market [19]. More related to MUDFLOW are the empirical studies of Allix et al. and Zhou et al. on Android malware [30], [1].

Shen et al. [26] employ a static data flow analysis technique to enrich the Android permission mechanism with information regarding detected information flows. To the best of our knowledge, their work is the only other work that compares the information flows between benign and malicious applications. Their final goal, though, is radically different from ours.

VIII. Conclusion and Future Work

MUDFLOW learns "normal" flows of sensitive data from trusted applications to detect "abnormal" flows in possibly malicious applications. The approach is effective in detecting novel attacks, learning from benignware only, as well as recognizing known attacks, learning from benign as well as malicious samples. Despite data flow analysis being expensive for real-world apps, we see the flow of sensitive data as a useful abstraction not only for automatic classification, but also for end users to understand what an app does with sensitive data.

Despite these successes, there still are lots of opportunities for improvement. Our own future work will focus on the following topics:

- To fool MUDFLOW, malware writers could use reflection, native code, self-decrypting code, or other features that challenge static analysis. Usage of such techniques in combination with sensitive data, however, would be unusual for benign apps. We are investigating analysis techniques that would detect such obfuscation techniques as anomalies.
- While static taint analysis across components and intermediate data storages is difficult, it is not fundamentally impossible. We want to design analysis techniques specifically tailored to app-wide and system-wide data flows as found in Android.
- Incorporating our earlier CHABADA work [12], we want to associate flows with app descriptions, detecting anomalies within specific application domains such as "travel", "wallpapers", and likewise.
- Where static analysis is challenged, combinations of automated test generation and dynamic flow analysis may prove to be helpful alternatives. We are investigating such combinations in conjunction with static analysis to combine the strengths of both static and dynamic flow analysis.

To support further research in app mining, as well as replication and extension of the results in this paper, all our mined data as well as the scripts for our statistical analysis are available for download. For details, see our project page

http://www.st.cs.uni-saarland.de/appmining/

Acknowledgment. This work was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IC12S01 as well as by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing". The work was also supported by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, by the DFG's Priority Program 1496 Reliably Secure Software Systems, and the project DFG RUNSECURE. Florian Gross provided the script for downloading the "benign" apps. Marcel Böhme, Úlfar Erlingsson, Florian Gross, Matthias Höschele, Clemens Hammacher, and Kevin Streit provided useful feedback on earlier revisions of this paper.

References

[1] K. Allix, Q. Jerome, R. S. Tegawendé F. Bissyandé, Jacques Klein, and Y. L. Traon. A forensic analysis of Android malware. how is malware

written and how it could be detected? In *Proceedings of the IEEE 38th Annual International Computers, Software & Applications Conference*, pages 384–393, 2014.

[2] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *2014 Network and Distributed System Security Symposium*, NDSS'14, 2014.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In *Proceedings of the 19th Conference on Computer and Communications Security (CCS)*, pages 217–228, New York, NY, USA, 2012. ACM.

[5] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 29–38, New York, NY, USA, 2003. ACM.

[6] 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.

[7] P.-H. Chen, C.-J. Lin, and B. Schölkopf. A tutorial on ν-support vector machines. *Applied Stochastic Models in Business and Industry*, 21:111–136, 2005.

[8] W. Enck, P. Gilbert, 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. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[9] 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*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[10] 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 the Foundations of Software Engineering (FSE 2014)*, 2014.

[11] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] 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*, ICSE 2014, pages 1025–1035, New York, NY, USA, June 2014. ACM.

[13] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: MSR for app stores. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, pages 108–111, 2012.

[14] 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*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[15] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[16] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011.

[17] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. I know what leaked in your pocket: uncovering privacy leaks on Android apps with static taint analysis. *arXiv*, 1404.7431, 2014.

[18] H. Lockheimer. Android and security, 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[19] M. Nagappan, B. Adams, and A. Hassan. Understanding reuse in the Android market. In *Proceedings of International Conference on Program Comprehension*, pages 113 – 122, 2012.

[20] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium*, pages 527–542, 2013.

[21] 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 ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[22] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, , and Z. Chen. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *Proceedings of the 21st Conference on Computer and Communications Security (CCS)*, 2014.

[23] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *2014 Network and Distributed System Security Symposium*, NDSS'14, 2014.

[24] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[25] I. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan. On the relationship between the number of ad libraries in an Android app and its rating. *IEEE Software*, 99(PrePrints), 2014.

[26] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek. Information flows as a permission mechanism. In *Proceeding of the 29th IEEE/ACM International Conference on Automated Software Engineering*, ASE'14, pages 515–526, 2014.

[27] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) permissions used by Android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 31–40, Piscataway, NJ, USA, 2013. IEEE Press.

[28] http://virusshare.com.

[29] Z. Yang and M. Yang. Leakminer: Detect information leakage on Android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE '12, pages 101–104, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.