ABSTRACT

Information flows in Android can be effectively used to give an informative summary of an application’s behavior, showing how and for what purpose apps use specific pieces of information. This has been shown to be extremely useful to characterize risky behaviors and, ultimately, to identify unwanted or malicious applications in Android. However, identifying information flows in an application is computationally highly expensive and, with more than one million apps in the Google Play market, it is critical to prioritize applications that are likely to pose a risk. In this work, we develop a triage mechanism to rank applications considering their potential risk. Our approach, called TriFlow, relies on static features that are quick to obtain. TriFlow combines a probabilistic model to predict the existence of information flows with a metric of how significant a flow is in benign and malicious apps. Based on this, TriFlow provides a score for each application that can be used to prioritize analysis. TriFlow also provides an explanatory report of the associated risk. We evaluate our tool with a representative dataset of benign and malicious Android apps. Our results show that it can predict the presence of information flows very accurately and that the overall triage mechanism enables significant resource saving.

Keywords

Android security; malware analysis; information flow; app triage

1. INTRODUCTION

The amount and complexity of malware in Android platforms has rapidly grown in the last years. By early 2016, both Symantec and McAfee report more than 300 malware families totalling over 12 million unique samples [26, 40]. Every malware family (and, sometimes, every sample within a family) may pose a different threat. The sheer number of apps available in current markets, along with the ratio at which new apps are submitted, makes impossible to manually analyze all of them. Automated analyses also have their limitations and some techniques might require a substantial amount of time per app [25]. This has motivated the need for a multi-staged analysis pipeline in which apps should be initially triaged to allocate resources intelligently and guarantee that the analysis effort is devoted to those samples that potentially have more security interest.

One of the salient features of Android’s security model is its permission-based access control system. Apps may request access to security- and privacy-sensitive resources in their manifest file. These requests are presented to end users through permission dialogs at install time or, since Android version 6 (Marshmallow), at runtime for a reduced subset of permissions. Requesting access to protected resources is a clear indicator of risk and most triage systems for Android apps have relied quite heavily on requested permissions (see, e.g., [11, 17, 20, 29, 34]), since they have proven effective to identify apps carrying malicious functionality. The majority of these approaches rely on metrics that combine the prevalence (or rarity) of each permission in benign and malicious apps with the criticality of the resources protected by the permission. Using permissions alone to assess risk has important limitations [2]. Permission-based risk metrics might be highly inaccurate for two reasons. First, apps are often overprivileged and many permissions requested in the manifest might not be actually used during execution. Second, they assign a risk to a particular permission (e.g., INTERNET) just because it could be used as a vehicle for a malicious purpose, such as leaking out a piece of sensitive data, without considering if sensitive data is actually being sent or not. Determining risk using Information Flows (IFs), as done by the approach introduced in this paper, overcomes this limitation and provides a more accurate assessment of the app’s actual behavior. However, IF analysis presents a number of challenges. Identifying flows in an app involves a non-negligible amount of resources both in time and memory. For instance, according to our experiments, it can take more than 30 minutes per app to extract IFs from at least half of the samples in the Drebin dataset [4] using a relatively powerful computer (40 processors and 200 GB RAM). The situation may even be worse when analyzing apps with sufficiently large call graphs. In those scenarios, the IF extraction might not even be practical [6].

Overview of our system. In this work we describe TriFlow, an IF-based triage mechanism for Android apps that attempts to overcome the issues discussed above for permission-based systems and also the limitations of existing IF analysis tools. Since extracting IFs from an app is an unreliable and computationally expensive process, TriFlow introduces the notion of speculative information flows. This means that TriFlow extracts some features from apps and then predicts the existence of a flow based on them. Pre-
diction is done on the basis of a model that is previously trained using ground truth obtained with flow extraction tools. Each predicted flow is then scored by TriFlow in terms of its potential risk, which depends on the flow’s observed prevalence in goodware and malware. To do this we rely on the cross-entropy between the empirical probability distributions of each flow in goodware and malware. This provides a simple but sound quantification of the intuition that an information flow is risky if it is frequent in malware and rare in benign apps.

TriFlow has been implemented in Python and tested using a combined dataset of more than 17,000 apps. Our results suggest that it is possible to predict information flows efficiently, with prediction errors remarkably small for the majority of information flows. The evaluation of the flow scoring measure reveals that 75% of information flows have no value at all for risk prediction, and only 1% of the remaining flows receive high weights. This suggests that malicious behavior (at least in the samples contained in the datasets used in this work) can be modeled using a relatively small subset of all possible information flows.

We evaluate TriFlow by simulating a triage process in which apps must be prioritized as they arrive. Our experimental results demonstrate that TriFlow outperforms existing permission-based risk metrics in all considered scenarios. Additionally, TriFlow provides an explicative report that describes the flows that most contribute to the overall risk assessment.

Contributions. In summary, in this paper we make three main contributions:

- We introduce the idea of predicting the existence of a particular information flow using static features extracted from an app’s code. We believe this idea might have potential beyond the scope of this work and, more generally, could be extended to predict the presence of other program artifacts whose precise identification requires computationally expensive static or dynamic analysis procedures.

- We extend to information flows the notion of “rare equals risky” that has been largely explored and tested in the field of permission-based risk metrics. Based on this, we design an information-theoretic risk measure related to the cross-entropy between the distribution of information flows in benign and malicious apps, thus quantifying how informative a flow is.

- Finally, we make our results and our implementation of TriFlow publicly available at

https://github.com/OMirzaei/TriFlow

to allow future works in this area to benefit from our research. TriFlow can be easily extended for new API methods and new information flows appearing in upcoming versions of Android, and its modular architecture facilitates its integration in existing risk assessment frameworks.

Organization. The rest of this paper is organized as follows. Section 2 describes in detail our approach for fast triage of apps based on speculative information flows. In Section 3 we present and discuss the results of our evaluation, including our prototype implementation and the datasets used (3.1). Additionally, we report: (i) the accuracy of the flow prediction (3.2) and the flow weighting (3.3) mechanisms; (ii) the triage results and the reports generated by TriFlow (3.4); and (iii) the efficiency of the tool (3.5). In

Section 4 we discuss a number of issues and limitations of our approach. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2. APPROACH

This section describes our approach for fast triage of Android apps. We first provide an overview of our proposal in Section 2.1. We then describe its two key ideas: a probabilistic estimator for information flows (Section 2.2) and a weighting scheme based on the a priori risk contribution of each information flow (Section 2.3). This is later used to rank apps and prioritize analysis.

2.1 Overview of Our System

A high-level view of TriFlow is provided in Fig. 1. The system is first trained using a dataset of benign and malicious apps. The goal of this phase is to obtain the two items that will be later used to score apps:

(i) A predictive model that outputs the probability

$$\theta_f(\phi_1, \ldots, \phi_n) = P[f | (\phi_1, \ldots, \phi_n)]$$  \hspace{1cm} (1)

of each possible information flow $f$ present in the app given a feature vector $(\phi_1, \ldots, \phi_n)$ obtained from the app’s code.

(ii) A risk model consisting of a function $I(f)$ that measures how informative each information flow $f$ is considering its relative frequency of occurrence in malware and benign apps.

The predictive model is estimated using both the feature vectors obtained from each app and the ground truth, i.e., the actual information flows present in the app, hence the flow identification component in our architecture. Note that we also tag each flow with the app’s label, i.e., whether it is benign or malicious.

Obtaining the score for an app (bottom part of Fig. 1) is done by simply multiplying each flow’s likelihood by its weight and summing up for all flows:

$$\text{score}(a) = \sum_f \theta_f I(f).$$  \hspace{1cm} (2)

Note that this only requires extracting the feature vector from the app and getting the $\theta_f$ and $I(f)$ values. As described in detail later, in TriFlow both models (prediction and risk) are implemented as look-up tables, so the overall scoring process is extremely fast.

Figure 1: Architecture of the proposed system.
2.2 Predicting Information Flows

Let $f = (s,k)$ denote an information flow from source $s$ to sink $k$. We aim at coming up with a predictor $P_f(a)$ that outputs whether $f$ is present in an app $a$ without actually performing an information flow analysis over the app. Our emphasis is on efficient predictors, so $P_f$ has to base its decision on features that can be extracted very efficiently from the app. TriFiFlow uses the presence of a call to the source $s$ and another to the sink $k$ in the app code as features. Determining the set of sources and sinks called by an app is straightforward. It can be done very efficiently by simply decompiling the app’s DEX file and matching the resulting code against a list of predefined sources and sinks.

We explored this idea using a probabilistic estimator as follows. Let $S(a)$ and $K(a)$ be the set of sources and sinks identified in the code of an app $a$. The set of all possible information flows in $a$ is the product set $\mathcal{F}(a) = S(a) \times K(a)$; that is, for each possible source $s \in S(a)$ and sink $k \in K(a)$, there is a potential flow $f = (s,k) \in \mathcal{F}(a)$. We now assume that the occurrence of each flow $f = (s,k)$ in an app is given by a probability distribution $\Theta = (\theta_1, \theta_2, \ldots)$ where $\theta_f = P[f = (s,k) | s,k]$. The estimator can be obtained using a dataset $\mathcal{D}$ of apps (malicious or not) as

\[
\theta_f = \frac{\sum_{a \in \mathcal{D}} \text{ind}_f(\mathcal{F}(a))}{\sum_{a \in \mathcal{D}} \text{ind}_f(\mathcal{F}(a))},
\]

where $\text{ind}_f(A) = 1$ if $x \in A$ or 0 otherwise, and $\mathcal{F}(a)$ is the set of actual information flows of the app $a$ extracted using an information flow analysis tool. Note that the denominator in Eq. (3) is always greater than the numerator, since the presence of a flow in an app requires a call to both the source and the sink, and, therefore, such a flow will appear in the $\mathcal{F}$ set.

Obtaining the $\theta_f$ estimator requires some computational effort since it involves obtaining the actual information flows for each app. However, once this task is done offline, the $\theta_f$ values can be stored in a look-up table and used after extracting the sources and sinks present in an app. Furthermore, the estimators can be incrementally refined when more apps become available, i.e., it does not require to go again through the set of potential and real flows for the already processed apps.

2.3 Informative Information Flows

The second component of our risk metric is a measure that quantifies how important a particular information flow is to distinguish malicious from benign apps. To do so, we adopt an empirical approach based on the relative frequencies of occurrence of information flows in both classes of apps. A similar idea has been leveraged in approach based on the relative frequencies of occurrence of information flows in both classes of apps. A similar idea has been leveraged in the SuSi project [30].

The evaluation is based on two distinct and non-overlapping splits of the datasets, i.e., training and testing. The predictive model is extracted using the former, while the latter is used to perform triage over unseen apps. For training we retained 4,000 samples (71%) from Drebin and an additional set of 4,000 (35%) from Google-Play. The training set thus contains the same amount of malware and goodware, i.e., a 1:1 malware-to-goodware ratio. Although the occurrence of malware in official markets is much lower than the presence of goodware, undersampling the training set is a common practice to equally weight both classes when building the model [8, 32]. For testing, we increased the malware-to-goodware ratio to 1:5, which is a common practice in other works in the area [1, 6, 46]. All these splits were done randomly and using a hold-out validation approach, i.e., the set of samples used for training differs from those selected for testing.
Table 1: Overview of the datasets used in this work. The upper part of the table shows the source of our dataset together with the number of samples from each source. The bottom part shows the training/testing splits used during cross-validation and the malware-to-goodware ratios.

<table>
<thead>
<tr>
<th>Type</th>
<th>Dataset</th>
<th>Type</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware (MW)</td>
<td>Drebin</td>
<td>Malware</td>
<td>5,560</td>
</tr>
<tr>
<td>Goodware (GW)</td>
<td>GooglePlay</td>
<td>Goodware</td>
<td>11,456</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>17,016</td>
</tr>
</tbody>
</table>

Table 2: Statistics of the training dataset. The size (in MB), number of sources (sr), number of sinks (skn), memory consumed (in GB), and time (in seconds) are given on average per app. The amount of memory (in GB) required represents the maximum average.

<table>
<thead>
<tr>
<th>#Apps</th>
<th>Size</th>
<th>#Src</th>
<th>#Skn</th>
<th>#Flow</th>
<th>Mem</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,000 MW</td>
<td>0.9</td>
<td>150.5</td>
<td>100.6</td>
<td>63.5</td>
<td>14.3</td>
<td>55.0</td>
</tr>
<tr>
<td>4,000 GW</td>
<td>6.2</td>
<td>223.1</td>
<td>124.4</td>
<td>255.5</td>
<td>88.3</td>
<td>132.1</td>
</tr>
<tr>
<td>8,000 ALL</td>
<td>3.5</td>
<td>186.8</td>
<td>112.5</td>
<td>159.5</td>
<td>51.3</td>
<td>93.6</td>
</tr>
</tbody>
</table>

We then used FLOWDROID [5] to identify data flows in all apps in our dataset. We ran FLOWDROID considering all Android API sources and sinks proposed in the SuSi project [30]. The extraction took place on a 2.6 GHz Intel Xeon Ubuntu server with 40 processors and 200 GB of RAM. We set a timeout of 30 minutes and between 40 GB and 100 GB of RAM per app in FLOWDROID. Even with this configuration, FLOWDROID could not finish the flow extraction process entirely for all the apps in our datasets. This lack of reliability has been reported before [6] and is indicative of the limitations (and computational cost) of techniques that rely on extracted information flows. For instance, analyzing a popular gaming app with more than 1 million installations in Google Play took about 90 GB of RAM and almost 2 hours of analysis time. Table 2 summarizes the main statistics of the dataset used to train TriFiFlow. In total, we identified 7,802 unique flows in the malware dataset and 28,163 unique flows in the goodware dataset. This difference can be attributed to the fact that apps in the benign apps set are, on average, much bigger in size and number of data flows than the apps in the malware dataset.

### 3.2 Flow Prediction Accuracy

Our first experiment evaluates the accuracy of the flow predictor introduced in Section 2.2. Our aim is to quantify the error made by the predictor and also to determine if such an error is somehow different for malware than for benign apps. Recall that for flow \( f \) appearing in an app the flow’s source and sink are located in the app. We define the prediction error for \( f \) in an app \( a \) as

\[
\text{error}(f) = \begin{cases} 
1 - \theta_f & \text{if } f \in \mathcal{F}(a) \\
\theta_f & \text{otherwise},
\end{cases}
\]  

where \( \mathcal{F}(a) \) is the set of actual information flows of \( a \). The error defined quantifies how far from the true value (i.e., 1 if the flow appears, and 0 otherwise) the prediction is.

In order to obtain a robust estimation of the prediction error, we applied 5-fold cross-validation to the two modeling (training) datasets described in Section 3.1. We used non-stratified cross-validation, i.e., folds are randomly built. Thus, each dataset is split into 5 folds of approximately equal number of apps. In each of the 5 iterations we estimated \( \theta_f \) using 4 out of the 5 folds and then obtained the error for all the apps in the remaining fold.

Table 3 provides the mean, standard deviation and median values for all the prediction errors obtained. In all cases, the results show that the predictor works remarkably well. Interestingly, it seems to be slightly easier to predict flows for benign than for malicious apps. We elaborate on this later on in this section when analyzing prediction errors for individual flows. When combining both datasets, the average error is similar to the one observed for goodware. This could be attributed to the fact that malware specimens in our dataset are often repackaged (i.e., the malicious app is built by piggybacking a benign app with a malicious payload), so many of the flows seen in malicious apps are not malicious as they do not originate in the piggybacked payload.

As for the provenance of the prediction error, Fig. 2 shows the error distribution for all flows in our datasets. We can observe that most flows are actually very easy to predict with low error. For the malware dataset, 4.31% of the flows (i.e., 337 out of 7802) are predicted perfectly (i.e., their prediction error is 0); around 83% of the flows can be predicted with an error lower than 0.1; and for around 90% of them the error is less than 0.25. The most frequent source API methods observed in these flows come from the TelephonyManager, Location, and Date packages. Similarly, the most relevant sink API methods observed come from the Camera.Parameters, and Log packages. For the goodware dataset, 1.04% of the flows (i.e., 293 out of 28163) are also predicted with no error and the figures are similar to the case of malicious apps (i.e., more than 90% of the flows can be predicted with an error lower than 0.25). Here, we observe that the most relevant source API methods come from Intent, Bundle, File, AudioManager, and View packages, while the most relevant sink API methods come from AudioManager, MediaRecorder, Log, Intent, and Bundle.

On the other hand, we observed a number of flows that are very hard to predict. In the case of malicious apps, flows from source methods used to retrieve data from intents (e.g., `getIntentExtra(java.lang.String,int)) to sinks related to media (such as `setVideoEncodingBitRate(int)) are error prone. For benign apps, we observe difficult-to-predict flows from sources that are used to retrieve PendingIntent before starting a new activity to sinks which are commonly used to set an intent when interacting with widgets (setPendingIntentTemplate(int,android.app.PendingIntent)). We did not examine further the reasons for such errors in certain flows and decided to leave this question for future work.

### 3.3 Flow Weights

We calculated the \( I(f) \) values for all the 31,175 unique infor-
Figure 2: Distribution of the prediction errors for all information flows in the two datasets. Note that in both plots the y-axis is in logarithmic scale.

Source API methods from sensitive categories that appear in malicious flows (see Table 5) try to access sensitive unique identifiers, including DeviceID, SubscriberID, NetworkOperator and SimSerialNumber. Interestingly, sink API methods appearing in those flows often check if unique identifiers start with a given prefix.

Figure 3: Cumulative probability distribution of the flow weight values $I(f)$. Note that the x-axis is given in logarithmic scale.

Table 4: Top ranked flows and their weight.

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>$I(f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM.getDeviceId()</td>
<td>String.startsWith()</td>
<td>0.69</td>
</tr>
<tr>
<td>TM.getDeviceId()</td>
<td>OutputStream.write()</td>
<td>0.26</td>
</tr>
<tr>
<td>TM.getDeviceId()</td>
<td>Intent.putExtra()</td>
<td>0.52</td>
</tr>
<tr>
<td>TM.getDeviceId()</td>
<td>String.substring()</td>
<td>0.28</td>
</tr>
<tr>
<td>TM.getDeviceId()</td>
<td>URL.openConnection()</td>
<td>0.37</td>
</tr>
<tr>
<td>TM.getSubscriberId()</td>
<td>String.startsWith()</td>
<td>0.88</td>
</tr>
<tr>
<td>TM.getSubscriberId()</td>
<td>OutputStream.write()</td>
<td>0.24</td>
</tr>
<tr>
<td>TM.getSubscriberId()</td>
<td>HttpURLConnection.setRequestMethod()</td>
<td>0.25</td>
</tr>
<tr>
<td>TM.getSubscriberId()</td>
<td>URL.openConnection()</td>
<td>0.42</td>
</tr>
<tr>
<td>TM.getSubscriberId()</td>
<td>Intent.putExtra()</td>
<td>0.58</td>
</tr>
<tr>
<td>TM.getSimCountryIso()</td>
<td>Log.i()</td>
<td>0.37</td>
</tr>
<tr>
<td>TM.getSimCountryIso()</td>
<td>String.substring()</td>
<td>0.25</td>
</tr>
<tr>
<td>gsm.SM.getDefault()</td>
<td>gsm.SM.sendTextMessage()</td>
<td>0.82</td>
</tr>
<tr>
<td>SM.getDefault()</td>
<td>SM.sendTextMessage()</td>
<td>1.81</td>
</tr>
<tr>
<td>NetworkInfo.getExtraInfo()</td>
<td>Log.d()</td>
<td>0.68</td>
</tr>
<tr>
<td>NetworkInfo.getExtraInfo()</td>
<td>String.startsWith()</td>
<td>0.45</td>
</tr>
<tr>
<td>WebView.getSettings()</td>
<td>WebS.setAllowFileAccess()</td>
<td>0.45</td>
</tr>
<tr>
<td>WebView.getSettings()</td>
<td>WebS.setGeolocationEnabled()</td>
<td>0.46</td>
</tr>
<tr>
<td>WebView.getSettings()</td>
<td>WebS.setPluginsEnabled()</td>
<td>0.50</td>
</tr>
<tr>
<td>System.getProperties()</td>
<td>String.substring()</td>
<td>0.45</td>
</tr>
<tr>
<td>PL.broadcast()</td>
<td>SM.sendTextMessage()</td>
<td>1.28</td>
</tr>
<tr>
<td>HashMap.get()</td>
<td>SM.sendTextMessage()</td>
<td>1.33</td>
</tr>
</tbody>
</table>

As discussed before, such a risk score can be used to rank apps and prioritize analysis. In addition to this, TRIFlow provides an explanation of the risk score similar to the one offered by Drebin [4] for the case of malware detection. In TRIFlow, this consists of a break down of the score into the flows that contribute the most to it and a presentation to the user grouped by SuSi categories, which are generally easier to understand than the specific source-sink pair.

We compared TRIFlow with other quantitative risk assessment metrics proposed in the literature. To do this we implemented various representative permission-based systems, including DroidRisk [41], Rarity Based Risk Score (RS) [14], and Rarity Based Risk Score with Scaling (RSS) [34]. As all these systems presented similar performance, in this section we only report results for RSS due to space limitations.

### 3.4.1 Scoring and Prioritizing Apps

Ideally, a triage system should maximize the time an analyst spends analyzing potentially harmful applications. Due to this reason, in this work we are primarily interested in reporting top ranked apps. Thus, we do not discuss the presence of other suspicious software such as grayware [3, 37] or obfuscated malware; we refer the reader to Section 4 for a more detailed discussion on this.

To quantify the performance of our triage system, we carried out the following experiment. We assume that the market operator only has time to manually vet a limited number of apps per unit of time (e.g., per day). We simulate a vetting process at different operational workloads \( w \), ranging from 10% to 100% of the

---

**Table 1**

(a) Average and (b) maximum values of the flow weight distribution with flows grouped by SuSi categories (sources are placed in rows and sinks in columns). The group NO_CATEGORY refers to sources and sinks classified as non-sensitive in SuSi.

<table>
<thead>
<tr>
<th>Category</th>
<th>LOG</th>
<th>FILE</th>
<th>NETWORK</th>
<th>SMS_MMS</th>
<th>AUDIO</th>
<th>NO_CATEGORY</th>
<th>LOCATION_INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETWORK_INFORMATION</td>
<td>0.0369</td>
<td>0.0074</td>
<td>0.0111</td>
<td>0.1767</td>
<td>N/A</td>
<td>0.044</td>
<td>N/A</td>
</tr>
<tr>
<td>CALANDER_INFORMATION</td>
<td>0.0104</td>
<td>0.0096</td>
<td>0.0063</td>
<td>N/A</td>
<td>N/A</td>
<td>0.0148</td>
<td>N/A</td>
</tr>
<tr>
<td>LOCATION_INFORMATION</td>
<td>0.0342</td>
<td>N/A</td>
<td>0.031</td>
<td>0.0054</td>
<td>N/A</td>
<td>0.0173</td>
<td>N/A</td>
</tr>
<tr>
<td>DATABASE_INFORMATION</td>
<td>0.0277</td>
<td>0.0157</td>
<td>0.022</td>
<td>0.0655</td>
<td>0.0032</td>
<td>0.0179</td>
<td>N/A</td>
</tr>
<tr>
<td>ACCOUNT_INFORMATION</td>
<td>0.0027</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.032</td>
<td>N/A</td>
</tr>
<tr>
<td>UNIQUE_IDENTIFIER</td>
<td>0.0824</td>
<td>0.0079</td>
<td>0.3059</td>
<td>0.0919</td>
<td>N/A</td>
<td>0.0508</td>
<td>N/A</td>
</tr>
<tr>
<td>BLUETOOTH_INFORMATION</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.0031</td>
<td>N/A</td>
</tr>
<tr>
<td>NO_CATEGORY</td>
<td>0.0284</td>
<td>0.0173</td>
<td>0.0382</td>
<td>0.0799</td>
<td>0.0097</td>
<td>0.0222</td>
<td>0.0088</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Category</th>
<th>LOG</th>
<th>FILE</th>
<th>NETWORK</th>
<th>SMS_MMS</th>
<th>AUDIO</th>
<th>NO_CATEGORY</th>
<th>LOCATION_INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETWORK_INFORMATION</td>
<td>0.6840</td>
<td>0.0096</td>
<td>0.0257</td>
<td>1.8161</td>
<td>N/A</td>
<td>1.1881</td>
<td>N/A</td>
</tr>
<tr>
<td>CALANDER_INFORMATION</td>
<td>0.0421</td>
<td>0.0128</td>
<td>0.0075</td>
<td>N/A</td>
<td>N/A</td>
<td>0.1284</td>
<td>N/A</td>
</tr>
<tr>
<td>LOCATION_INFORMATION</td>
<td>0.1403</td>
<td>N/A</td>
<td>0.1175</td>
<td>0.0128</td>
<td>N/A</td>
<td>0.1626</td>
<td>N/A</td>
</tr>
<tr>
<td>DATABASE_INFORMATION</td>
<td>0.2092</td>
<td>0.0544</td>
<td>0.0471</td>
<td>0.2336</td>
<td>0.0032</td>
<td>0.2766</td>
<td>N/A</td>
</tr>
<tr>
<td>ACCOUNT_INFORMATION</td>
<td>0.0028</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.1056</td>
<td>N/A</td>
</tr>
<tr>
<td>UNIQUE_IDENTIFIER</td>
<td>0.5216</td>
<td>0.0187</td>
<td>0.4279</td>
<td>0.1536</td>
<td>N/A</td>
<td>1.0901</td>
<td>N/A</td>
</tr>
<tr>
<td>BLUETOOTH_INFORMATION</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.0032</td>
<td>N/A</td>
</tr>
<tr>
<td>NO_CATEGORY</td>
<td>0.6032</td>
<td>0.4618</td>
<td>0.5292</td>
<td>1.3315</td>
<td>0.0376</td>
<td>1.1776</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Figure 4: (a) Average and (b) maximum values of the flow weight distribution with flows grouped by SuSi categories (sources are placed in rows and sinks in columns). The group NO_CATEGORY refers to sources and sinks classified as non-sensitive in SuSi.
analyzed samples. More precisely, we assume that the operator receives batches of \( N \) samples per minute and their analysts are capable of processing 10\%, 20\%, \ldots, 100\% of them. This constitutes a realistic scenario as some market operators can be more agile than others. The same applies to antivirus vendors. For instance, out of the 310,000 new samples received every day, Kaspersky Labs only processes 1\% manually (2 per minute)\(^2\). For our experiments we set \( N = 10 \), though the particular value is irrelevant for our analysis as it only constitutes a scale factor.

For each workload, we prepare a batch of samples containing randomly chosen samples from the joint goodware and malware datasets (recall that the malware-to-goodware ratio for testint is 1:5, so on average there will be 5 times more goodware than malware in each batch). Each sample in the batch is then scored and the top \( w \)% ranked samples are given to the analyst for a deeper analysis. We measure how many samples (in \%) that final block of samples passed on to the analyst are malware. We repeated this process 900 times, obtaining one percentage each time. For each workload \( w \), the distribution of values is given in the boxplots shown in Fig. 5. We repeated the process for both TriF\(w\)ow and RSS [34]. We also compared how both systems behave against a random ordering of the batch of samples. The square (□) symbol in each plot of Fig. 5 denotes the average ratio of malware samples given to the human analyst after using a random prioritization policy, while the diamond (◇) symbol denotes the value given by the triage system.

For our experiments we set \( N = 10 \), though the particular value is irrelevant for our analysis as it only constitutes a scale factor.

For each workload, we prepare a batch of samples containing randomly chosen samples from the joint goodware and malware datasets (recall that the malware-to-goodware ratio for testint is 1:5, so on average there will be 5 times more goodware than malware in each batch). Each sample in the batch is then scored and the top \( w \)% ranked samples are given to the analyst for a deeper analysis. We measure how many samples (in \%) that final block of samples passed on to the analyst are malware. We repeated this process 900 times, obtaining one percentage each time. For each workload \( w \), the distribution of values is given in the boxplots shown in Fig. 5. We repeated the process for both TriF\(w\)ow and RSS [34]. We also compared how both systems behave against a random ordering of the batch of samples. The square (□) symbol in each plot of Fig. 5 denotes the average ratio of malware samples given to the human analyst after using a random prioritization policy, while the diamond (◇) symbol denotes the value given by the triage system.

For each workload, we prepare a batch of samples containing randomly chosen samples from the joint goodware and malware datasets (recall that the malware-to-goodware ratio for testint is 1:5, so on average there will be 5 times more goodware than malware in each batch). Each sample in the batch is then scored and the top \( w \)% ranked samples are given to the analyst for a deeper analysis. We measure how many samples (in \%) that final block of samples passed on to the analyst are malware. We repeated this process 900 times, obtaining one percentage each time. For each workload \( w \), the distribution of values is given in the boxplots shown in Fig. 5. We repeated the process for both TriF\(w\)ow and RSS [34]. We also compared how both systems behave against a random ordering of the batch of samples. The square (□) symbol in each plot of Fig. 5 denotes the average ratio of malware samples given to the human analyst after using a random prioritization policy, while the diamond (◇) symbol denotes the value given by the triage system. The red arrow joining them represents the difference between both numbers, i.e., the time saved after triaging the batch. Our results show that in all cases TriF\(w\)ow can prioritize more malware samples per batch (see upper quartiles in Figure 5b) than RSS for every single workload value. Although not shown in the

---

\(^2\)http://apt.securelist.com

<table>
<thead>
<tr>
<th>Source Categories</th>
<th>Unique Identifier</th>
<th>DATABASE INFORMATION</th>
<th>LOG</th>
<th>FILE</th>
<th>NETWORK</th>
<th>SMS_MMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSerialNumber()</td>
<td>getDeviceId()</td>
<td>getConnectionId()</td>
<td>write()</td>
<td>openConnection()</td>
<td>sendTextMessage()</td>
<td></td>
</tr>
<tr>
<td>getSubscriberId()</td>
<td>getSimSerialNumber()</td>
<td>query()</td>
<td>dump()</td>
<td>selectUserName()</td>
<td>sendTo()</td>
<td></td>
</tr>
<tr>
<td>getNetworkOperator()</td>
<td>getLineId()</td>
<td>getSyncState()</td>
<td>bind()</td>
<td>disableNetwork()</td>
<td>recordSound()</td>
<td></td>
</tr>
<tr>
<td>getAllMessagesFromSIM()</td>
<td>getNumber()</td>
<td>getColumnsNames()</td>
<td>openFile()</td>
<td>disableWifiApEnabled()</td>
<td>sendData()</td>
<td></td>
</tr>
<tr>
<td>getHost()</td>
<td>getAllPermissionGroups()</td>
<td>getRowCount()</td>
<td>openFileInput()</td>
<td>setWifiApEnabled()</td>
<td>sendDataMessage()</td>
<td></td>
</tr>
<tr>
<td>getRemotePort()</td>
<td>getInstalledApplications()</td>
<td>sendData()</td>
<td>openFileOutput()</td>
<td>sendWifiApBackend()</td>
<td>setPremiumSmsPermission()</td>
<td></td>
</tr>
<tr>
<td>getRemoteAddress()</td>
<td>getLongitude()</td>
<td>v()</td>
<td>gameOver()</td>
<td>setNetworkPolicies()</td>
<td>dispatchMessage()</td>
<td></td>
</tr>
<tr>
<td>getLinkAddress()</td>
<td>getLatitude()</td>
<td>w()</td>
<td>startListening()</td>
<td>setServiceProviderName()</td>
<td>append()</td>
<td></td>
</tr>
<tr>
<td>getNetworkPolicies()</td>
<td>setDefault()</td>
<td>install()</td>
<td>storeFile()</td>
<td>setNetworkName()</td>
<td>setBandwidth()</td>
<td></td>
</tr>
<tr>
<td>getCellIdentity()</td>
<td>getNetworkPolicies()</td>
<td>notify()</td>
<td>sendTo()</td>
<td>setNetworkCountryIso()</td>
<td>disableGetBroadcast()</td>
<td></td>
</tr>
<tr>
<td>getLongitude()</td>
<td>getNetworkPolicies()</td>
<td>setUserName()</td>
<td>sendTo()</td>
<td>getNetworkOperator()</td>
<td>moveMessageToFolder()</td>
<td></td>
</tr>
<tr>
<td>getSampleId()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getNetworkCountryIso()</td>
<td>setTestVisibility()</td>
<td></td>
</tr>
<tr>
<td>getDeviceName()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>storeFile()</td>
<td>getSimCountryIso()</td>
<td>setAuthUserName()</td>
<td></td>
</tr>
<tr>
<td>getNumber()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getSubscriberId()</td>
<td>setScanMode()</td>
<td></td>
</tr>
<tr>
<td>getDeviceSerialNumber()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getPhoneNumber()</td>
<td>processMessage()</td>
<td></td>
</tr>
<tr>
<td>getNetworkCountry()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getNetworkOperator()</td>
<td>v()</td>
<td></td>
</tr>
<tr>
<td>getNetworkOperator()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getNetworkCountryIso()</td>
<td>w()</td>
<td></td>
</tr>
<tr>
<td>getAllMessages()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getNetworkOperator()</td>
<td>bind()</td>
<td></td>
</tr>
<tr>
<td>getNetworkOperator()</td>
<td>getNetworkPolicies()</td>
<td>getUser()</td>
<td>sendData()</td>
<td>getNetworkOperator()</td>
<td>write()</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Results of the triage evaluation. Each plot shows the distribution of the fraction of malware correctly prioritized (y-axis) when a market operator can only afford to analyze \( w \)% of the samples (x-axis) at each time interval (e.g., daily-basis). Results are given for both RSS (left) and TriF\(w\)ow (right). The red arrows within each plot represent the gain achieved by each scoring system with respect to a random prioritization policy.

Table 5: Most relevant sources and sinks from sensitive categories.
3.5 Efficiency

We now discuss the efficiency of TriFLOW measured as the time required to obtain the score for an app. The scoring process has two main steps: extracting the sources and sinks of the app to construct the set $\hat{F}$ of possible information flows, and then computing the score by adding up the product $\theta_f I(f)$ for each flow $f \in \hat{F}$. The first step requires identifying all existing sources and sinks, whereas the second depends on the size of $\hat{F}$, i.e., the number of sources times the number of sinks in the app. Fig. 7 shows both quantities for all the apps in our datasets (GooglePlay and Drebin). We consistently observe approximately twice the number of sources than sinks in each app, with an average of 290.10 and 176.81, respectively. The average size, measured as the number of potential flows, is 77,187.

Fig. 8 shows the overall time required to obtain the score for each app as a function of its number of potential flows. On average it takes 56 seconds to triage the entire app. The minimum and maximum scoring time for an app in our dataset is 0.01 seconds and 76.63 minutes, respectively. Approximately 50% of the apps require less than 31 s; 80% of the apps require less than 103 s; and 90% of the apps require less than 155 s. On average, TriFLOW requires 2.3 ms per potential flow in the app. Execution times are not constant for a given size because not all potential flows will have a non-null probability of occurrence. The higher the number of flows with $\theta_f > 0$, the higher the number of risk terms that have to be added to the total score. This process is largely non-optimized in our prototype, hence the substantial variability observed in Fig. 8.

When processing a large dataset of apps, most of the computation time goes to the extraction of the information flows. Fig. 9 shows the comparison between the time taken by our approach and FlowDroid. We can observe that FlowDroid is computationally more intensive than TriFLOW. In particular, we observe an improvement of about two orders of magnitude for smaller set of apps and about one order of magnitude for larger sets. This is a natural advantage of using a probabilistic predictor with respect to a precise tainting analysis, though it should only be used as an estimation for fast risk analysis.

4. DISCUSSION

We next discuss a number of potential limitations of our approach related to its accuracy, the underlying risk notion, the validity of our results, and attacks against the scoring system.

Accuracy. A crucial step in TriFLOW is the accurate identification of the sources and sinks present in an app. Our approach to...
Information flow analysis in Android. Information flows provide meaningful traces that describe how data components are propagated amongst the variables (and components) of a program. Such flows can be used to represent the behavior of a given pro-

**Risk notion.** TriFlow scores apps according to the probable presence of interesting flows. In this paper, we have quantified how significant a flow is using the mechanism described in Section 3.3, which captures how useful the flow might be to identify malicious apps. While we believe this is a useful risk metric, we also acknowledge that its use might easily lead to misinterpretations. Specifically, apps that score high should not be thought of as “likely malware,” but simply as apps that possibly contain dangerous information flows (dangerous in the sense that they are more frequent in malicious than in benign apps). During our experiments we came across some benign apps that score higher than many malicious samples, including, for instance, three known antivirus products (McAfee Mobile Security, NQ Mobile Security, and Vodafone Protect).

Our flow weighting scheme could be easily extended to incorporate other relevant flow features, or simply replaced by another measure of significance provided by the analyst (e.g., different weights for different SuSi categories). More generally, TriFlow should be viewed just as a risk metric finer-grained than permissions, and in a real setting its use should be complemented with other risk metrics that consider features of an app other than permissions or information flows.

**Datasets.** The experimental results discussed in this paper might be affected by the number and representativeness of the apps in our datasets. While the exact coverage of our datasets cannot be known, we believe they are fairly representative in terms of different types of benign and malicious apps. For the latter we relied on the Drebin dataset, which extends the widely used Malgenome dataset and has been consistently used by most works in the Android malware area in the last two years. In the case of benign apps, we could only afford analyzing around 4000 applications, including 42 which are amongst the top most downloaded apps from Google Play in 2016. The limiting factor here is the extraction of information flows (with FLOWDROID, in our case), which requires a substantial amount of computational resources and, furthermore, fails for a large fraction of apps. This limitation is, in fact, one key motivation for our work. In any case, we did our best to avoid selection bias by choosing apps of different sizes and from different categories, prioritizing when possible those more popular (in terms of downloads) in the Google Play market.

**Evasion attacks.** A sensible goal for an adversary is to modify his app so that it receives the lowest possible score. Since the score is monotonically increasing in the number of flows, adding sources or sinks will never decrease the score. To reduce the overall score an adversary will need to remove the use of some sources or sinks (which may affect the app’s functionality), or just make them undetectable (e.g., as discussed above in the case of reflection). Alternatively, the adversary could try to replace current flows by others that use sources and sinks that are functionally equivalent to the original but receive a considerably lower weight. In our current implementation, this would only be possible by relying on methods rarely used by malware. We have not explored the extent of this limitation, and it is left for future work.

Our approach is vulnerable to collusion attacks since it does not consider information flows across apps (i.e., when the source is located in one app and the data is passed on to another app that access the sink). This can be seen as an extension to information flows of the classical permission redelegation attacks [13], and can only be solved by extending individual analysis to groups of apps (e.g., such as in [22, 39]).

## 5. RELATED WORK

**Information flow analysis in Android.** Information flows provide meaningful traces that describe how data components are propagated amongst the variables (and components) of a program [45].
gram, showing how and for what purpose programs are using specific pieces of information [6]. Any information flow is characterized by two main points defining the direction of the flow, known as the source and the sink. Sources are points within the program where sensitive data are obtained or stored in memory, while sinks are points where such data are leaked out of the program [44].

Unlike traditional desktop operating systems, apps in Android have their own life cycle and multiple execution entry points [19]. There are two types of information flows in Android applications. Explicit information flows analyze data-flow dependencies without considering the control-flow of the program. In contrast, implicit information flows analyze the control-flow dependencies between a source an a sink [31]. State-of-the-art analysis techniques (e.g. FlowDroid [5]) generally rely on explicit flows for two main reasons. First, implicit data flows can be tracked at a reasonable cost in most of the applications; and second, tracking such flows are unnecessary for many systems [31].

From another point of view, information flows are categorized as either inter-app or intra-app depending on the type of communication. Inter-app communication, and, as a result, inter-app information flows are established between components of two different applications [9, 12]. On the opposite side, intra-app data flows are those established between different components of the same application [35]. In addition, information flows are usually tracked using—static or dynamic—taint analysis [21]. Static taint analysis aims at detecting privacy leaks before the execution of the application by constructing a control flow graph, while dynamic taint analysis tries to keep track of such leaks in run-time or in a customized execution environment [18].

There are several recent information flow analysis frameworks for Android (see Table 6). Static taint analysis tools such as FlowDroid [5], DroidSafe [16], FlowMine [36], CHEX [24], LeakMiner [43], and AndroidLeaks [15] have a relatively low run-time overhead with respect to other information flow frameworks. However, suffer from some critical issues that cannot be overlooked. On the one hand, they are imprecise as they need to simulate run-time behaviors [5], and, as a result, suffer from a high false positive rate [6]. On the other hand, some of these frameworks do not scale well with the number of applications [16]. Finally, applications could use advanced obfuscation techniques to hinder the extraction of information flows (e.g. [38]).

Similar to our approach, authors in MUDFLOW [6] use information flow analysis to study how malicious and benign apps treat sensitive data. MUDFLOW is able to establish a profile based on sensitive flows that allows them to characterize potential risks that are typically observed in malware. Our system, in a way, is motivated by these findings and by the fact that flow extraction involves a non-negligible amount of resources. In this paper, instead of simply analyzing the abnormal usage of sensitive information, we use speculative information flows to further triage Android apps.

Dynamic taint analysis systems such as TaintDroid [10] and DroidScope [42] generally compensate for the lack of precision of static tools. However, these frameworks inherit the limitations of dynamic analysis systems, i.e., they may miss data flows from parts of the code not explicitly exercised [6, 16]. Furthermore, apart from the fact that they impose a high run-time overhead [43], a malicious app could potentially fingerprint a given dynamic monitoring system to evade detection [5].

Tainting analysis frameworks are generally based on sensitive API calls tracking. Thus, it is paramount that this tracking considers the way apps interact with the system services. In Android, this interaction is stateless. This means that the taint analysis system has to take into account the life-cycle of applications and model all possible entry points and callbacks defined by the developer. Furthermore, sensitive API calls can also be declared in a native library outside of the main Dalvik Executable (DEX) and should also be modeled. Table 6 summarizes the most relevant information flow analysis frameworks discussed in each of the aforementioned categories together with the type of components modeled from the Android OS. Note that FlowDroid and DroidSafe are the only two static tainting frameworks that consider all modeling assumptions simultaneously.

Permission-based risk metrics for Android apps. The development of metrics and systems to assess risk in Android apps is an area that has received much attention in the last years. Works in this area have generally relied on metadata obtained from the app’s package, such as requested permissions, and from the market, including the number of downloads, number of views, or the developer’s reputation. Permission-based risk scores have been by far the most commonly explored because of two key advantages: permissions are relatively easy to understand by users and are compatible with the risk communication mechanism currently used in Android. Furthermore, app developers can reduce risk by avoiding the use of unnecessary permissions [14].

One of the seminal works in this area is [11], in which the authors propose a system based on a number of rules that represent risky permissions to flag apps. More recent contributions introducing permission-based risk metrics include DroidRanger [46], DroidRisk [41], MAST [7], WHYPER [28], RiskMon [20], MADAM [33], and the works of [27] and [23]. The risk metric proposed in DroidRisk [41] is based on the frequency and number of permissions an application request. In MAST [7], a risk signal is created based on the declared indicators of the app’s functionality, such as permissions, intent filters, and the presence of native code. The intuition behind this idea is that apps which are stronger in terms of finding relations between these indicators impose a higher magnitude of risk and, thus, should be flagged as malicious. WHYPER [28] uses natural language processing techniques to reveal why an app may need a specific permission, paying attention to permissions’ purposes. MADAM [33] relies mainly on metadata from the market, including the developer’s reputation and market provenance. Finally, RiskMon and the work in [23] consider API traces as well, since some of them are critical and do not require any permissions. Finally, [14], [34], and [29] assign high risk scores to permissions or combination of permissions that are critical and rarely requested by the apps in the same category.

As permission-based metrics are based on metadata of the app obtained through static analysis, they can be imprecise and prone to errors. Other metrics have tried to overcome this by looking into features other than permissions. For instance, RiskRanker [17] introduces a risk signal based on root exploits, while [23] proposes a risk score considering static metadata, dynamic information from intents, components, network usage, and the app’s behavior (e.g., whether an app launches other apps). Finally, the majority of metrics, except [20] and [27], do not take into account the security requirements or expectations of smartphone users. This is particularly important in practice, since risk ultimately depends on each user’s preferences and execution context.

Our approach is complementary to most of these works. While we share the goal of quantifying risk, our primary focus is not on malware detection, but on prioritizing information flow analysis. Furthermore, our flow-based scoring mechanism can be easily integrated with existing metrics based on other risk factors to provide a more comprehensive risk assessment.
6. CONCLUSION

In this paper, we designed and implemented a novel tool, called TriageFlow, that automatically scores Android apps based on a forecast of their information flows and their associated risk. Our approach relies on a probabilistic model for information flows and a measure of how significant each flow is. Both items are experimentally obtained from a dataset containing benign and malicious apps. After this training phase, the models are used by a fast mechanism to triage apps, thus providing a queuing discipline for the pool of apps waiting for a precise information flow analysis.

Our experimental results suggest that TriageFlow provides a sensible ordering based on the potential interest of the app. Given the huge amount of computational resources demanded by information flow analysis tools, we believe this could be very helpful to maximize the expected utility when dealing with large pools of apps. Additionally, TriageFlow could also be used as a standalone risk metric for Android apps, providing a complementary perspective to alternative risk assessment approaches based on permissions and other static features. Finally, to encourage further research in this area, we make our results and implementation available online.

Acknowledgments

This work was supported by the MINECO grants TIN2013-46469-R and TIN2016-79095-C2-2-R, and by the CAM grant S2013/ICE-3095. The authors would like to thank the anonymous reviewers for their valuable comments.

7. REFERENCES

A. Razeen, V. Pistol, A. Meijer, and L. P. Cox. Better

S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning

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 31st Annual Computer Security


A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli.
Madam: Effective and efficient behavior-based android
malware detection and prevention. IEEE Transactions on

B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru,
and I. Molloy. Android permissions: A perspective
combining risks and benefits. In 17th ACM Symposium on
Access Control Models and Technologies, SACMAT ‘12,

R. Schuster and E. Tromer. Droiddisintegrator:
Intra-application information flow control in android apps. In
11th ACM Asia Conference on Computer and

W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer.
Android taint flow analysis for app sets. In 3rd ACM
SIGPLAN International Workshop on the State of the Art in
Java Program Analysis, pages 1–6, 2014.

L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon.
Apkcombiner: Combining multiple android apps to support
inter-app analysis. In ICT Systems Security and Privacy
Protection: 30th IFIP TC 11 International Conference, SEC
2015, Hamburg, Germany, May 26-28, 2015, Proceedings,

S. Li, T. Tryfonas, G. Russell, and P. Andriotis. Risk
assessment for mobile systems through a multilayered
hierarchical bayesian network. IEEE Transactions on

vetting android apps for component hijacking vulnerabilities.
In ACM conference on Computer and communications

E. Mariconti, L. Onwuizure, P. Andriotis, E. De Cristofaro,
G. Ross, and G. Stringhini. Mamadroid: Detecting android
malware by building markov chains of behavioral models. In
The Network and Distributed System Security Symposium
(NDSS), 2017.


privacy risks in android: A user-centric approach. In Risk

Towards automating risk assessment of mobile applications.

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 ACM
Conference on Computer and Communications Security,

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

A. Razeen, V. Pistol, A. Meijer, and L. P. Cox. Better
performance through thread-local emulation. In 17th
International Workshop on Mobile Computing Systems and

S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou,
V. P. Ranganath, H. Li, and N. Guevara. Experimental study

of my market: Detecting malicious apps in official and
alternative android markets. In 31st Annual Computer Security

L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing
the os and dalvik semantic views for dynamic android
malware analysis. In 21st USENIX Security Symposium

Z. Yang and M. Yang. Leakminer: Detect information
leakage on android with static taint analysis. In Software

Pift: Predictive information-flow tracking. In 21st
International Conference on Architectural Support for
Programming Languages and Operating Systems, pages

W. You, B. Liang, J. Li, W. Shi, and X. Zhang. Android
implicit information flow demystified. In 10th ACM
Symposium on Information, Computer and Communications

of my market: Detecting malicious apps in official and