AppHolmes: Detecting and Characterizing App Collusion among Third-Party Android Markets

Mengwei Xu[1], Yun Ma[1], Xuanzhe Liu[1]*, Felix Xiaozhu Lin[2], Yunxin Liu[3]
[1]Key Lab of High-Confidence Software Technology, MoE (Peking University). Beijing, China
[2]Purdue University. West Lafayette, USA, [3]Microsoft Research. Beijing, China
{xmw, mayun, xzl}@pku.edu.cn, xzl@purdue.edu, yunxin.liu@microsoft.com

ABSTRACT

Background activities on smartphones are essential to today's "always-on" mobile device experience. Yet, there lacks a clear understanding of the *cooperative behaviors* among background activities as well as a quantification of the consequences. In this paper, we present the first in-depth study of app collusion, in which one app surreptitiously launches others in the background without user's awareness. To enable the study, we develop AppHolmes, a static analysis tool for detecting app collusion by examining the app binaries. By analyzing 10,000 apps from top third-party app markets, we found that i) covert, cooperative behaviors in background app launch are surprisingly pervasive, ii) most collusion is caused by shared services, libraries, or common interest among apps, and iii) collusion has serious impact on performance, efficiency, and security. Overall, our work presents a strong implication on future mobile system design.

Keywords

Program Analysis; Mobile Computing; Community Detection

1. INTRODUCTION

Modern mobile apps often run background activities periodically to poll sensor data, maintain cloud activity, or update their local state [31]. While such background activities are essential to the "always-on" experience desired by mobile users today, they often have incurred substantial overhead. For example, a recent study [19] examined 800 apps running over 1,520 devices, and the results surprisingly indicated that background activities can account for more than 50% of the total energy for 22.7% of the apps, with an average of 27.1% across all 800 apps.

Fundamentally, the intention of background execution in Android or iOS is to enable an app – directly launched by the user – to remain alive and get executed from time to time. However, the mechanism also allows one app to programmatically launch other apps in the background, even though the users have never interacted with, or even not aware of the launched apps.

In this paper, we use *app collusion* to denote such **user-unaware cross-app launch**[1]. Compared to the background activities that are initiated by explicit user input or configuration, app collusion is covert and often more sophisticated. As a result, the incurred system overhead and security risk are not only substantial but also often hidden. What is even worse, such overhead and risk will grow as one user installs more apps on her mobile device.

Motivated by anecdotes [14, 16, 15] about serious app collusion on third-party app markets, we present the first large-scale, systematic study of app collusion that focuses on three aspects: (1) characterizing app collusion across a massive set of mobile apps; (2) revealing the root causes of app collusion; (3) quantifying the consequence of app collusion. To enable the study, we build ***AppHolmes***, a static analyzer that not only retrofits existing techniques for analyzing Inter-Component Communications (ICC) [36, 28, 37, 39, 29, 35] but also infuses domain knowledge for identifying important collusion categories, which would have been omitted by existing tools.

Equipped with the AppHolmes tool, we have studied over 10,000 Android apps collected from three leading third-party app markets in China, i.e., Baidu[2], Tencent[3], and Wandoujia[4]. These markets are the major channels for distributing Android apps in mainland China, each of which has hundreds of millions of users [11, 12]. To our surprise, we find that the app collusion prevails among Android apps – in particular the popular ones. Our further investigation reveals the major root causes of app collusion as *third-party push services*, *functional SDKs*, and *shared resources*. Finally, we quantify the hidden cost and security risk of app collusion over typical apps. Our major findings are as follows:

• **App collusion prevails in third-party Android markets.** Among the top 1,000 apps from Wandoujia, 822 of them exhibit collusion behaviors; on average, each app col-

*Corresponding author: xzl@pku.edu.cn.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052645

[1]Note that we use the word "collusion" to describe end user's experience, despite that not all such launches are intentionally planned by app developers, as will be discussed in Section 6.
[2]`http://shouji.baidu.com/`
[3]`http://android.myapp.com/`
[4]`http://www.wandoujia.com/`

ludes with 76 other apps. We have similar observations on the Tencent and Baidu markets.

• **Popular apps are more likely to be collusive.** While each of the top 1,000 apps colludes with other 76 apps on average, the same metric decreases to 51 for apps ranked between top 1,001 $\sim$ 2,000, and further decreases to 37 for apps ranked between top 2,001 $\sim$ 3,000.

• **One app may exhibit different collusion behaviors when downloaded from different markets.** The common reason is that app developers intentionally adapt their apps in order to conform to the market-specific policies.

• **Different categories of apps behave differently in collusion.** For instance, among the top 1,000 apps from Wandoujia market, each shopping app colludes with 174 other apps on average, almost 7 times more than game apps.

• **Asymmetry in app collusion.** Popular apps are more likely to be launched by other apps rather the other way around. On average, for a top-50 app A in the Wandoujia market, the apps that may launch A in the background are twice as many as the apps that A may launch in the background.

• **Sharing third-party push service is a primary cause of app collusion.** We have discovered that *all* the 8 most popular third-party push services can cause collusive relations, which constitute 77% of the total app collusion discovered by us.

• **Even app developers may be unaware of their app collusion.** This is because the collusion behaviors may root in the third-party SDKs or services that are shipped with poor documentation, used by app developers without much understanding, and integrated as black boxes.

• **App collusion leads to significant hidden cost**. Through experiments with 40 popular apps, we show that for every single foreground app, the background collusion can lead to additional **202 MB** memory use, extra **8.9%** CPU usage, and **16** additional high-risk permissions on average. With app collusion present, the user perceives **2**× latency in launching foreground apps and the off-screen battery life is reduced by 57%.

We have made the following contributions:

- We present the first large-scale study of collusive behaviors of apps and demonstrate the prevalence of app collusion among third-party Android markets.
- We reveal the root causes of app collusion, which explain why and how apps are involved in collusive relations.
- We quantify the hidden cost and security risks introduced by app collusion.

The remainder of the paper is organized as follows. We present the background knowledge in Section 2. We formally define the problem in Section 3. We describe AppHolmes, our static analyzer for detecting app collusion in Section 4. We present the analysis of collusion of popular apps from three major third-party Android markets, reveal their root causes, and quantify the resultant hidden cost in Section 5, Section 6, and Section 7, respectively. We discuss the implications and limitations of our work in Section 8, discuss prior work in Section 9, and conclude in Section 10.

2. BACKGROUND

To make the collusion behaviors clearly presented, we first introduce some background knowledge in this section.

• **App Components.** In the Android programming model, an app is mainly composed of four types of components:

- ***Activities*** represent user interfaces, which dictate the UI and handle the user interaction to the phone screen.
- ***Services*** handle background processing associated with an application.
- ***Broadcast receivers*** can receive broadcasts from Android OS or other applications.
- ***Content provider*** allows sharing of structured data across different apps.

An app is launched (or woken up), as we say, if at least one of its component is loaded into the memory and running.

• **Cross-app launch APIs.** Android provides several IPC mechanisms to enable inter-app communications, among which some APIs can even wake up the target app by launching one of its component. These APIs include: 1) *startActivity(Intent), startActivityForResult(Intent),* etc, which can switch to another UI page. If the target Activity exists but has not been launched, the system will launch it first. 2) *startService(Intent)* and *bindService(Intent).* Similarly, such APIs will launch the target Service if it has not been launched. 3) *sendBroadcast(Intent, permission).* The system will resolve the broadcast to find out all matched target Broadcast Receivers. The common pattern of Broadcast Receiver, is starting a service and dispatching the events to this service to be handled. 4) *getContentResolver().query(URI).* This API is to access the open data from Content Provider of external app.

• **Resolve Intents.** An *Intent* is an abstract description of an operation to be performed, which contains fields like *action*, *data*, etc. These fields are used to decide whether *Intents* can be matched to *IntentFilter* that are declared in manifest files. There are two primary forms of intents: 1) **Explicit Intents** specify a component by providing the exact class name of the target component. 2) **Implicit Intents** do not specify a component; instead, they must include enough information for the system to determine which components to run for that intent.

3. PROBLEM STATEMENT

Intuitively, app collusion is a directed relationship among two apps indicating that one can launch the other without users' notice. We formally define the app collusion as follows.

An app $app = \{com_i\}$ can be abstracted as a set of components com_i. Components can communicate with each other to realize the functionality of an app, denoted as *Inter-Component Communication* (ICC). Generally, ICC can occur between components either within the same app or from different apps. To study the app-collusion behavior, we define a special case for ICC named Inter-App Collusion (IAC) as $\langle com_i, com_j \rangle$, where three conditions are satisfied: 1) com_i can launch com_j, 2) com_i and com_j come from different apps, and 3) com_j is not an activity. It should be particularly noticed that the third condition constrains no existence of UI switch, so that IAC is invisible and un-perceived to users. At app level, app collusion is a binary relationship among apps, app collusion $= \langle app_i, app_j \rangle \subset APP \times APP$, where $\exists com_i \in app_i, com_j \in app_j, \langle com_i, com_j \rangle \in IAC$.

The following code shows a simple implementation of app collusion. When `methodA` is invoked, a new `Intent` is initialized to target at a service from another app (line 4∼5), and

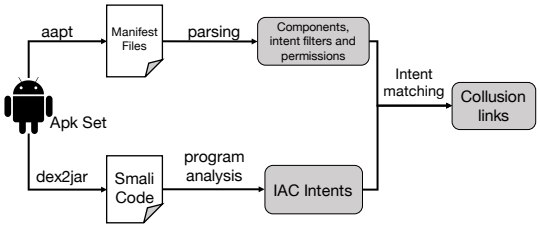


Figure 1: The workflow of AppHolmes.

sent to the system to wake up that service (line 7). Since a service has no UI elements, users won't notice it.

Listing 1: A code sample of app collusion

```
1 public class classA{
2     public void methodA(Context context) {
3         // com.example.target is not the current package;
4         Intent intent = new Intent();
5         intent.setClassName("com.example.target",
              "com.example.target.otherService");
6         intent.setAction("some_action");
7         context.startService(intent);
8     }
9 }
```

In addition, we need to describe the collusion behavior for a set of apps, and how one app behaves in this context of app set: given a set of apps K, we define $Collusion(K)$ as the collection of all the app collusion pairs $\langle app_i, app_j \rangle$ among K. For each app $app_i \in K$, we define:

$$In\text{-}Collusion(app_i) = \{app_j \mid \langle app_j, app_i \rangle \in Collusion(K)\}$$
$$Out\text{-}Collusion(app_i) = \{app_j \mid \langle app_i, app_j \rangle \in Collusion(K)\}$$

Intuitively, *In-Collusion* describes for one app, how many other apps can wake up it, and *Out-Collusion* describes how many it can wake up.

4. APPHOLMES: COLLUSION DETECTOR

To enable large-scale analysis on Android apps, we design and implement a static analysis tool named AppHolmes. As shown in Figure 1, AppHolmes takes three steps to detect app collusion among a given set of apps (K).

• **Extraction.** AppHolmes first extracts two kinds of information from apks: 1) manifest files via Android Asset Packaging Tool (aapt) [1] and 2) smali code via dex2jar [3]. The manifest files declare one app's components, intent filters and permissions; the smali code, which is a disassembled version of the DEX binary used by Android's Davik VM, enables us to carry out static program analysis.

• **Program Analysis.** Next, AppHolmes performs static program analysis on the extracted smali code and identifies possible *Intent* values from all callsites of *startService(), bindService()* and *sendBroadcast()*. These callsites are potential candidates to trigger collusion behaviors with the corresponding *Intents*.

• **Match Intents.** Finally, AppHolmes matches all *Intents* detected in Step 2 and *IntentFilters* extracted from manifest files in Step 1. If an *Intent* from app A matches to an *IntentFilter* from app B, AppHolmes reports a potential app collusion as $\langle A, B \rangle \in Collusion(K)$.

The program analysis techniques of AppHolmes are based on the ideas of IC3 [35], which infers all possible values of complex objects in an interprocedural, flow, and context-sensitive manner. However, AppHolmes is different from IC3 in the following aspects: 1) AppHolmes focuses on IAC among different apps, which is only a small subset of ICC. Therefore, we can filter most string values of intra-app communications and give a performance boost. 2) AppHolmes tackles more Android APIs that may affect the Intent values. For example, we observe that many apps use `queryIntentServices(Intent, flags)` to retrieve all services that can match a given intent. Then they wake up these services via the return values of `queryIntentServices`. These APIs are used only for inter-app communications, therefore are not considered by most ICC tools like IC3. 3) AppHolmes is tailored for some special cases that cannot be handled by universal static analysis techniques. For instance, we observe that apps using the Getui SDK[5] will load from files a list of app names for launching in the background. These runtime-generated strings are almost impossible to reveal in generic static analysis. AppHolmes, by contrast, tracks such files once it notices that the app being analyzed integrates the Getui SDK, according to the app's manifest. Such ad-hoc strategy may not be generalized in other cases but helps us explore more app collusion.

Achieving Both High Precision and Recall. AppHolmes must produce reliable and practical results. However, PRIMO [34], which is a probabilistic model to accurately match ICCs, points out that a conventional matching strategy can cause lots of false positives (95.6%). Learned from this model and our empirical observations, we regard that package name and action value are the dominate fields for matching explicit intents and implicit intents, respectively. Therefore, AppHolmes adopts a new strategy to keep both precision and recall: 1) explicit *Intents* match only if the package name matches. If other fields are successfully parsed and not empty, they also need to be matched. But if they are not successfully parsed or parsed to be empty, it will not cause the match to fail. 2) implicit *Intents* match only if the action matches. Similarly, other fields also need to match if they are not empty.

Tool Evaluation. To evaluate the accuracy of AppHolmes, we collect ground truth by developing an Xposed [10] module, which can trace all app collusion at runtime. We set up a test device with 30 popular apps installed on it, and record the actual occurred collusion by using the device for 12 hours. Comparing the static analysis results reported by AppHolmes with the runtime trace, we find that the recall and precision are 97% and 71%, respectively. This result shows that AppHolmes has a high coverage of collusion that actually occurs during execution, and only incurs a relatively small number of false positives. Note that 71% is actually the lower bound of precision, since dynamic running is limited by the user behaviors and time duration, therefore cannot make sure to go through all possible code blocks inside each app.

5. CHARACTERIZING APP COLLUSION

Based on AppHolmes presented in Section 4, we perform a large-scale analysis of popular Android apps from three popular Android markets in China: Wandoujia, Tencent and Baidu markets. More than 2 million apps are available on each of these markets. For each app, the market provides the

[5]A popular third-party push service. More details in Section 6.

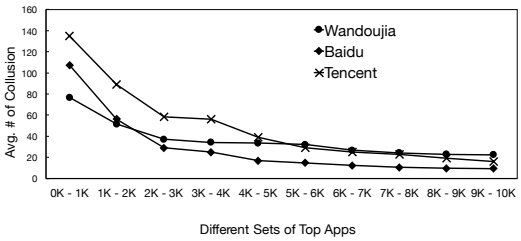


Figure 2: The average count of apps that are collusive with a given app on different Android markets. Each set includes 1,000 apps, sorted by app downloads.

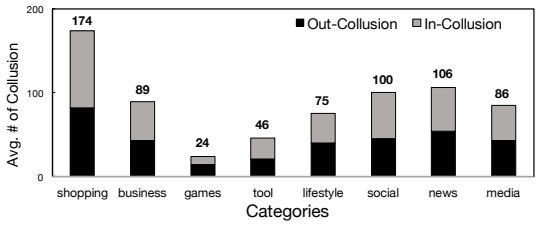


Figure 3: The average count of collusive apps for a given app from different categories (Wandoujia top 1,000).

binary apk file and the number of downloads. We crawled top 10,000 apps from these markets during July 2016 - August 2016. In this paper, we mainly present the results of Wandoujia, and the complete analysis results are available online[6].

Results Overview. Our key observation is that app collusion is a common phenomenon in third-party Android markets. For example, as shown in Figure 2, among top 1,000 apps from Tencent, 894 of them show collusion behaviors and on average, each app colludes with 138 other apps. We will dig into the root causes of this phenomenon in Section 6.

Collusion vs Popularity. Another interesting observation from Figure 2 is that popular apps are more likely to be collusive. Taking Wandoujia apps as an example, the average number of collusive relations is 76 for top 1,000 apps. The number goes down to 51 for top 1,000∼2,000 and 37 for top 2,000∼3,000.

There are two major reasons for such correlation. First, the richer features in popular apps are more likely to be invoked by other apps. For example, many apps allow users to login via their WeChat and QQ[7] accounts. To achieve this, developers need to integrate WeChat and QQ SDKs into their apps, and these SDKs are responsible for introducing app collusion into these apps. Second, most popular apps are developed by a small number of big companies, e.g., at least 11 apps out of top 100 apps from Wandoujia are from Tencent company. These apps owned by the same company are usually collusive with each other, to share services or user data.

Collusion vs Markets. The overall trend of collusion is the same in all three markets. However, the divergence of average collusion number of top 1,000 apps is quite large, as Tencent market is 138 while Wandoujia is only 77. To explain such divergence, we identify two causes: 1) app rankings can be different on different markets. We find that there are only 512 common apps among top 1,000 apps from both Baidu and Tencent markets. For example, it is observed that many apps developed by Qihoo and Tencent companies are ranked high on Tencent market but not available on Baidu market. 2) Even the same set of apps can behave differently on different markets, since developers may customize their apps to fit some policies made by different markets. [32]

We then dig into the second cause by studying the collusive behavior of an identical set of 50 apps (same version) across all the three markets. AppHolmes detects 65 app collusion for the apps from Wandoujia and Tencent markets, but 92 app collusion for the Baidu market. Our investigation reveals that the Baidu market *enforces* all apps to integrate the Baidu SDK, which leads to app collusion.

Collusion vs App Categories. Figure 3 shows the relation between the average number of collusion and app categories. As observed, shopping apps have the largest number of collusion (174). In contrast, game and tool apps have very few collusion (24 and 46).

To account for this difference, we identify two possible explanations, i.e. the nature of apps and the developer demographics. First, shopping apps have more intention to keep running in the background so that it can notify some new products and discounts, which may attract users to place more orders. We find that 41 out of 51 shopping apps (80%) are integrated with popular third-party push services, which is the primary cause to app collusion as we will present in the next section. However, for game and tool categories, the percentage of apps integrated with push services is only 11% and 23%, respectively. This difference stems from the unique characters of different categories of apps.

Second, the most popular shopping apps are mostly from big companies, while game and tool apps are mostly from small companies or even individual developers. For example, we observe that out of the 51 shopping apps from Wandoujia top 1,000, 8 of them are owned by Alibaba. As we mention above, apps from the same company tend to be more collusive.

Asymmetry of Out- and In-Collusion. Though the total number of Out and In-Collusion for a given set of apps is the same, we find that popular apps tend to have more In-Collusion than Out-Collusion. In other words, popular apps are more likely to be launched in the background rather than launching other apps. We calculate the detailed Out and In-Collusion number for top 50 apps from Wandoujia (among top 1,000 apps). The average In-Collusion for these 50 apps is 75, almost two times as many as Out-Collusion (35). This disparity can be extremely large for some apps like Sina Weibo, which has 374 In-Collusion but only 2 Out-Collusion.

Such asymmetry is interesting yet not unexpected: popular apps have richer features or data that other apps can utilize. For instance, Wechat (com.tencent.mm) has the largest In-Collusion (540), since it provides many functionalities in its SDK such as login, paying, sharing to other apps. To utilize these features, third-party apps need to wake up WeChat first. We will discuss this phenomenon in detail in Section 6.

6. ROOT CAUSES OF APP COLLUSION

Next we present the major causes of app collusion and

[6] `http://www.mobisaas.org/projects/appholmes/`

[7] Wechat and QQ are the most popular social apps in China.

Table 1: A list of popular third-party push services in Chinese markets and how they contribute to app collusion.

Push Service	# in top 1K	Having Collusion?	Mechanism for background launch
MiPush	161	Yes	Explicit broadcast
Getui	104	Yes	Start service
Umeng	85	Yes	Bind service
Baidu	61	Yes	Implicit broadcast
tPush	57	Yes	Implicit broadcast
jPush	45	Yes	Start service
huawei	28	Yes	Explicit broadcast
leancloud	7	Yes	Start service

describe how apps can be clustered in different communities based on their collusive behaviors.

6.1 Classifications of App Collusion

To have better understanding into why apps are collusive, we break down the overall 1,710 collusive relations among Wandoujia top 100 apps into four categories: push service, functional SDK, shared resource, and miscellaneous.

Third-party Push Service (77%). Nowadays, many mobile apps rely on push service to timely receive updates from the cloud. On Android platforms, Google provides an official push service framework, called Google Cloud Messaging (GCM) [13]. To implement GCM, the Android OS runs a separated system process that receives messages from the cloud and delivers them to corresponding apps.

In countries or regions where Google service is unavailable, there arise many third-party push services to achieve the same goal. To use one of these push services, app developers need to integrate the corresponding push library into their applications. Different from GCM that lives in a system process, these third-party push services have to live in app processes which are subject to termination by users or OS, e.g. when the system runs short of memory.

We further dig into how these push services behave by disassembling 8 push libraries widely used in China. We find that to keep push services alive, third-party push libraries request the enclosing apps to launch other "companions" in the background: if more than one app integrated with the same push library are installed on the same device, each of them launches the others from time to time. As shown in Table 1, all investigated push libraries can lead to app collusion. As the major cause to app collusion, third-party push services contribute 77% to the total 1,710 collusive relations among top 100 apps from Wandoujia.

Though such "one for all, all for one" feature increases the chance of immediately receiving messages from cloud, it also introduces hidden cost to system and developers. Besides the significant system cost, which will be quantified in Section 7, it also puts extra burden on developers and bloats the app binaries. To run on devices manufactured by certain vendors, e.g. Xiaomi or Huawei, apps need to use the vendor's own push libraries to gain the best reliability or lowest delay. To fit a variety of devices, many apps are packaged with multiple push libraries, and use one of them depending on the user's actual device. For example, the Toutiao app[8] is shipped with three push libraries: Umeng, MiPush and Huawei Push; it uses MiPush on Xiaomi devices, uses Huawei Push on Huawei devices, and uses Umeng on others.

Functional SDK (15%). Many apps expose some of their own functionalities – in the form of SDK – to other apps for invocation. For example, the Facebook SDK allows third-party apps to login via Facebook accounts, share messages, get Facebook's social graph, etc. Such interactions will inevitably lead to app collusion. This contributes around 15% to the overall collusion.

While the collusion due to functional SDK use sometimes is well justified (the app exposing the functionalities have to run anyway), in our study, we found that excessive, unjustified collusion emerges due to abuse of functional SDK. Take the WeChat SDK as an example. Prior to using WeChat's rich social features, third-party apps need to register themselves with WeChat by invoking the latter's SDK, which will launch WeChat in the background through a specific broadcast. A reasonable way of performing registration, as one may expect, should be on demand, i.e. when the requesting app is indeed using WeChat's features. However, many apps do so aggressively, immediately invoking the WeChat SDK as soon as they are launched, and therefore cause unnecessary app collusion. During the tool evaluation mentioned in Section 4, 24 out of 30 installed apps are integrated with WeChat SDKs, and 9 of them exhibit such an abuse.

Shared Resource (5%). Apps from the same company are often collusive to each other, which contributes around 5% to the overall number. Taking Baidu as an example, we notice that almost every Baidu app has a registered service called `com.baidu.sapi2.share.ShareService`. Our collected log by running these Baidu apps on one device shows that they share this service with each other via `bindService()`, e.g., using users' search keywords from a search-engine app can help recommend products on a shopping app. Such cooperation may help provide more powerful features by combining user data generated in different apps, but it also inevitably causes unnecessary resource consuming as demonstrated in Section 7.

Miscellaneous (3%). We also found interesting cases that fall out of any category above. The purposes of such cooperation can be diverse, e.g., exchanging user data for advertisements, retrieving news information, or just simply keeping apps alive. For instance, in the code of a reader app (`com.ophone.reader.ui`), we find it hard-codes a target app name (`com.qihoo360.mobilesafe`) in order to wake up the latter.

6.2 Clustering Apps via App Collusion

To understand how apps are organized into *communities* according to their collusive relations, we visualize the network graph of top 1,000 apps from Wandoujia. As shown in Figure 4, a node represents an app while the directed edges represent the collusion among two apps. The node size is proportional to the in-degree of this node. We perform ForceAtlas2 [4] algorithm to layout the graph, and use Modularity [8] to divide all nodes into different communities (denoted by the node colors).

As we can see, the overall network graph is complicated, and is primarily clustered into six communities by different colors. Given that the use of third-party push services is a major reason of collusion, apps sharing the same push service are naturally clustered into the same community. For example, the group of red nodes (left top) uses tPush, the group of green nodes (middle bottom) uses Getui and the group of blue nodes (middle top) uses Umeng. Interestingly, the geographical-central community (purple) are apps *not*

[8]A popular news client in China

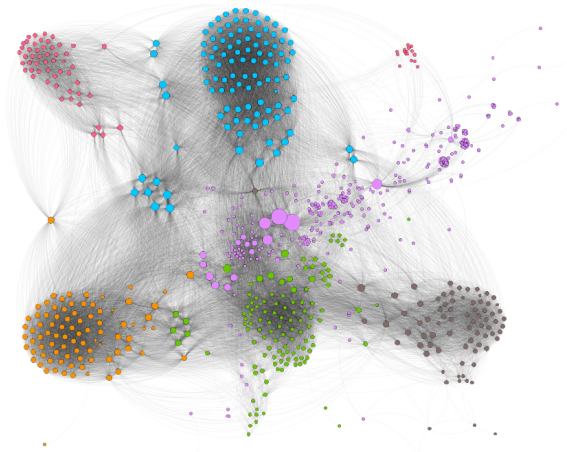


Figure 4: App communities based on detected collusion among top 1,000 apps from the Wandoujia market. Each node represents an app while each edge represents the collusive relations. Different colors represent different communities.

using the common push libraries listed in Table 1. One possible reason of these apps' concentrating into the same group rather than sparsely scattered, is that they are all "attracted" by a relatively small set of popular apps. These popular apps, like WeChat and Weibo, reside in the core of purple community, and have large "gravity" to others. Such gravity is caused by the functional SDKs.

Even within a given community, apps may form even smaller subgroups. For instance, the blue community (middle top of the figure) represents apps using Umeng SDK; it is further partitioned into two subgroups, one of which is closer to the purple community and has larger in-degrees. This stems from the fact that some apps are integrated with not only Umeng SDK but also others like Xiaomi SDK. These apps are drawn to the other communities and therefore separated from the remaining apps in the same community.

7. HIDDEN COST OF APP COLLUSION

To demonstrate the hidden cost of app collusion, we carry out a set of experiments over a set of popular apps. The results show that app collusion has significant impacts on user-perceived latency, energy consumption, security risk, as well as system resource including CPU and memory.

7.1 Experiment Methodology

App Selection. According to a study of millions of Android users [27], there are around 40 apps installed at the same time for one device (mean: 43.6, median: 37.0). To make our experiment results representative, we randomly select 40 apps from top-100 apps (by the number of downloads) of Wandoujia, and then split them into two sets: 10 as Foreground apps and 30 as Background apps. We list the app information in Table 2. Foreground apps are those that we directly interact with, while Background apps are never launched manually and can be woken up only via app collusion.

Experiment Design. To have in-depth and comprehensive understandings of how app collusion can impact user experience, we design three experiments. All these experiments are repeated for 20 times on Nexus 6 with Android 6.0. We use monkey [17] tool to automatically run over Foreground apps.

Table 2: The Foreground and Background apps used in our experiments, selected from the Wandoujia top 100.

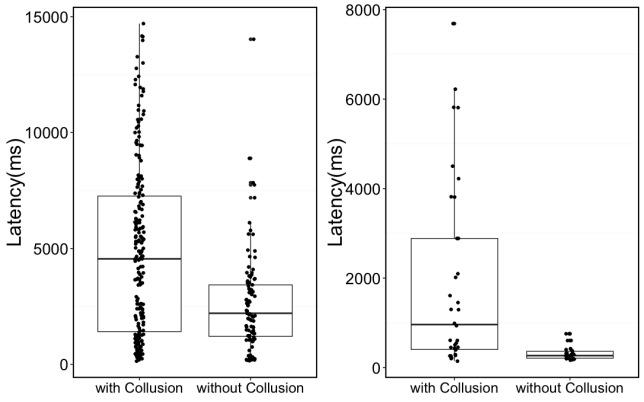
	App Name	Category	# of Downloads
Foreground apps	QQ	Social	560 M
	Taobao	Shopping	460 M
	Sina Weibo	Socail	210 M
	Baidu Map	Map	200 M
	iQiyi	Video	190 M
	Baidu Searchbox	Tool	180 M
	Baidu Browser	Browser	65 M
	PPS TV	Video	56 M
	Shuqi Reader	Books	40 M
	Nuomi	Shopping	18 M
Background apps	WeChat	Social	530 M
	QQ Browser	Browser	440 M
	WiFi Locating	Tool	400 M
	QQ Zone	Social	380 M
	Tmall	Shopping	270 M
	Tudou Video	Video	260 M
	Palm Reading iReader	Books	250 M
	Where Travel	Travel	240 M
	UC Mobile	Browser	240 M
	Sogou Map	Map	220 M
	Alipay	Finance	200 M
	Youku Video	Video	170 M
	Gaode Map	Map	110 M
	Meituan	Shopping	89 M
	STORM	Video	76 M
	Today's Headlines	News	76 M
	Sohu Video	Video	73 M
	ES File Explorer	Tool	67 M
	Jingdong	Shopping	62 M
	PPTV Video	Video	58 M
	Baidu Cloud	Tool	55 M
	Baidu Video	Video	49 M
	Wangyi News	News	49 M
	WoStore	Tool	39 M
	Baidu Tieba	Social	38 M
	Changba	Music	38 M
	hao123 Navigation	Browser	26 M
	Himalayan FM	News	19 M
	Sina News	News	16 M
	Beauty Shot	Image	15 M

- ***Exp I.*** We run each Foreground app one by one for 30 seconds. Meanwhile, we log the CPU and memory usage for each Foreground and Background app via `dumpsys`[9].
- ***Exp II.*** We run 10 Foreground apps at the same time to simulate how real users use devices. When switching among different Foreground apps, we do not manually kill the Background apps. We log the user-perceived latency of each activity launching for each Foreground app[10].
- ***Exp III.*** We first run each Foreground app once before killing them manually. We then turn off the screen and keep the phone silent. We log the battery drop from fully charged status to the phone automatically turned off.

We design ***Exp II*** and ***Exp III*** to study direct consequences from app collusion that are perceivable to users, and ***Exp I*** to gain more comprehensive and detailed results in system aspect. Note that we feed the Monkey tool with the same seeds when running with and without Background app collusion. To make sure the overhead is caused by app collu-

[9] A system tool that runs on the device and provides information about the status of system services including memory and CPU usage, battery level, etc.

[10] Note that more than one activity will be launched for each Foreground app and we record the latency for all these activities rather than only the app launch.



(a) All activities. **(b)** Baidu Searchbox activity.

Figure 5: Launch latency of Foreground apps with and without the presence of app collusion.

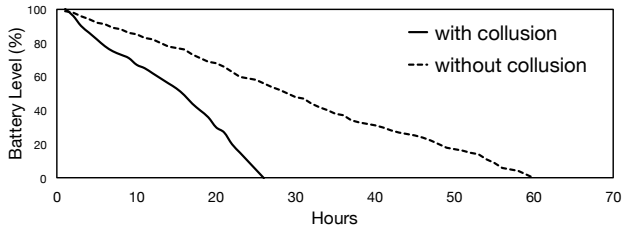


Figure 6: The impact of app collusion on battery life, estimated when the screen is always kept off.

sion, we enforcedly stop all Background apps beforehand so that these apps cannot be woken up by system broadcasts.

7.2 Experiment Results

In overall, ***Exp I*** shows 9 out of 10 Foreground apps have at least one Out-Collusion among these 30 Background apps, and each one colludes to 6.4 apps averagely. Next we report the cost of app collusion from different aspects, among which the user-perceived latency (***Exp II***) and energy consumption (***Exp III***) can be directly perceived by users, while CPU, memory and extra permissions (***Exp I***) can help understand why latency and energy are severely impacted.

2x user-perceived latency. As illustrated in Figure 5(a), the overall latency is severely lengthened by app collusion, as the median number is almost two times as much as the situation without app collusion (4,556ms vs 2,208ms). Figure 5(b) shows more detailed results for a search page activity from Baidu Searchbox. Obviously, the launch time is quite stable around 400ms without app collusion. When app collusion is introduced, half of the launch time go beyond 1s, and even more than 5s in some cases. Such significant overhead means sluggish response from device screens, and therefore affects user experience when interacting with their devices.

Two key reasons are expected to cause such increased latency: higher memory pressure and higher CPU usage. For example, we observe that Foreground apps are much more likely to be squeezed out from memory with app collusion. For frequently used apps, being removed out of memory should be avoided, otherwise a cold start is needed to relaunch the app, which usually takes much more time than a warm start. Among 200 app launches (20 times for each Foreground app) in our experiment, only 34 is cold start when there's no app collusion. But with app collusion, 97 cold starts occur, which is almost 2 times more. This is one of the reasons explaining why app performs more smoothly without app collusion as mentioned in last paragraph. We will further discuss the memory and CPU overhead later.

2.3x energy consumption. As shown in Figure 6, our experimental device lasts for 59 hours without app collusion in the background. This standby duration is cut down to 26 hours while there exists many background apps woken up by app collusion. Such significant impact is consistent with prior knowledge, as Google already determined that for every second of "active" use on a typical phone, standby time is reduced by two minutes [5]. For real device users, whose devices are usually put in pocket or on office table, this energy issue can be extremely bothering since the battery runs out quickly with little interacting usage.

16 additional high-risk permissions required. Android permission strategy has been studied and criticized for long period of time [20] [23]. Excessively granting permissions to apps can lead to potential privacy leak, especially for those marked as "dangerous" permissions by Android OS [2], e.g., contacts, SMS and camera. In our experiment, we quantify the number of dangerous permissions owned by Background apps that are woken up via app collusion.

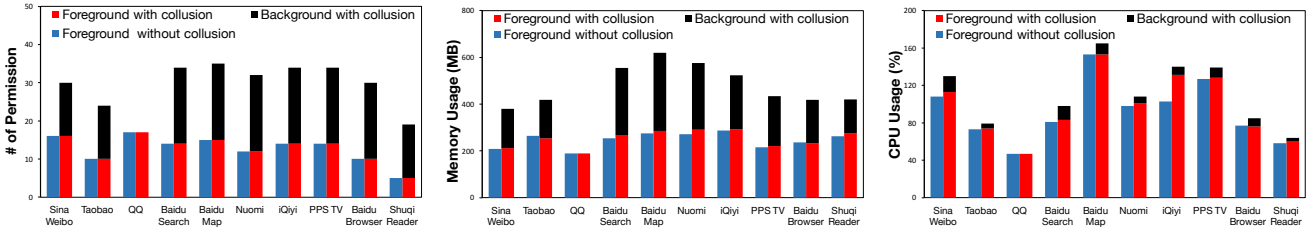
As shown in Figure 7(a), Background apps that are woken up via app collusion require many extra dangerous permissions (16 averagely), e.g., Shuqi Reader only asks 5 dangerous permissions, but the Background apps woken up by Shuqi Reader ask for 14 dangerous permissions in total. Even worse, Background apps may require additional permissions that the Foreground app does not need. For instance, `android.permission.ACCESS_COARSE_LOCATION` is used to access GPS for location information, which is not required by Shuqi Reader, but required by the Background apps colluding with Shuqi Reader. It means app collusion increases the probability of privacy leak, and also causes more energy consumption since it keeps more sensors on (GPS).

202M memory usage overhead. Figure 7(b) shows the memory overhead caused by app collusion. For each app, the left bar shows its memory usage without app collusion. The right bar is combined by the memory usage of itself and the memory consumed by the Background apps it wakes up. Although background services usually consume less memory than foreground activities, since they ask no memory for UI rendering, our experiments results show that such memory overhead is non-negligible. The average memory overhead caused by app collusion is 202M, and the worst case, Baidu Map, can wake up 13 Background apps which consume 334M memory. Such overhead is even larger than the memory usage of the Foreground app itself (286M).

8.9% CPU usage overhead. Similarly, we report the CPU overhead in Figure 7(c). The average CPU overhead of app collusion is 8.9%, around 10% compared to the CPU usage of Foreground Apps (96.6%). Note that the usage percentage is calculated for one core so it may exceed 100% (Nexus 6 has 4 cores).

8. DISCUSSION

This section presents implications to mitigate the impacts



(a) # of sensitive permissions requested. **(b)** Memory Overhead. **(c)** CPU Overhead.

Figure 7: A breakdown of extra system resource usage, without and with the presence of app collusion.

from app collusion, and discusses the limitations of our tool and analysis work.

8.1 Mitigate Impacts of App Collusion

Given the common phenomenon and severe impacts of app collusion, we propose some suggestions to mitigate such impacts therefore improve user experience as well.

• **To end users.** There are three approaches for users to reduce the influence from app collusion: 1) try not to install apps that are seldom used, especially apps that have extensive collusive relations to other apps. 2) choose a proper Android market to download apps, since same apps can behave differently on different markets as afore-mentioned. 3) carefully disable the app collusion via some third-party apps like Greenify [6], LBE [7] or custom ROMs.

• **To OS vendors.** The module of inter-app communication in Android should be improved to alleviate the hidden cost brought by app collusion. Here we provide two suggestions. On one hand, when app collusion is detected to launch third party apps without the notice of users, the system can confirm with the users whether the app collusion is allowed or not. The potential cost of the collusion could be presented to the users to help them make the decision. On the other hand, Android can provide a scheduler for inter-app communications. Some communications that may not influence the functionality of apps could be deferred to execute.

• **To app developers.** We suggest developers to carefully design and implement apps to avoid unnecessary app collusion. For example, while integrating Push SDK is inevitable since implementing their own Push Service can be tedious and time-consuming, they can choose what SDK to use and whether they want to be involved in app collusion.

• **To app markets.** Nowadays many Android markets have their client-side app where users can directly find, download, and install the apps they want. Our analysis tool AppHolmes can facilitate them to notify users before downloading a new app that how many other installed apps are likely to collude to this app. With this knowledge from markets, users gain better insights into the consequences of installing a new app, and they may make better choice when there are many apps that can satisfy their demands.

8.2 Limitations

Though AppHolmes is carefully designed and implemented, it has certain limitations. First, it doesn't handle app collusion via *Content Provider*. The reason we leave out such case is that *Content Provider* is the most infrequently used component, and it uses *URI* rather than *Intent* to deliver messages. It requires more engineering efforts to tackle this issue since currently we focus on only *Intent*. We plan to add support for *Content Provider* later.

Indeed, our current analysis focuses on third-party Android markets only in China. The current results may not be well generalized to apps that are released on Google Play or other markets. It would be interesting to explore whether the similar observations can be also found on these markets and we plan to conduct the study as future work. However, given the huge amount of apps and mobile users of the investigated third-party markets, our study and results in this paper can still be meaningful.

9. RELATED WORK

In this section, we discuss existing literatures that relate to our work in this paper.

• **Misuse of ICC.** Inter-Component Communication (ICC) has been studied extensively in recent years. They focus on the security and privacy issues caused by the misuse of ICC, including privilege escalation attacks [23, 18, 26, 30], sensitive data theft [21, 22] and malicious data access [40]. App collusion, which is actually a subset of ICC, can lead to not only security problems but also long user-perceived latency and quick energy drain as we have demonstrated in our experiments.

• **ICC analysis.** Numerous tools [34, 24, 35, 36, 28, 37, 39, 29] have been developed and applied on ICC analysis. PCLeaks [29] uses a static taint analysis technique to detect potential component leaks that could be exploited by other components. FUSE [37] provides a multi-app information flow analysis, and an evaluation engine to detect information flows that violate specified security policies. IccTA [28] improves the precision of the ICC analysis by propagating context information between components, and claims to outperform other existing tools. Eppic [36] reduces the discovery of ICC to an instance of the Interprocedural Distributive Environment (IDE) problem, and provides a sound static analysis technique targeted to the Android platform. COAL [35] is the first generic solver that infers all possible values of complex objects in an interprocedural, flow and context-sensitive manner, taking field correlations into account. PRIMO [34] combines static analysis with probabilistic models to make static inter-application analysis more tractable, even at large scales. Our work, AppHolmes, is based on the same static analysis technique and learns from these tools. However, it is differentiated from these existing tools in two aspects: 1) AppHolmes focuses on inter-app communications, which is only a subset of ICC. Therefore, we can ignore most ICC cases ($> 90\%$) to give a perfor-

mance boost, which is quite necessary to carry out a large-scale analysis. 2) AppHolmes avoids much false positive by a middle-strategy on *Intent* matching. Existing tools try to find as many ICC as possible, but our work needs to give reasonable data analysis.

• **App performance.** Mobile app performance is essential to user experiences and has been studied broadly from many aspects [33, 25, 38]. Background activities are identified as a critical factor that can impact app performance in consideration of system resource consumption. Chen, *et al.* [19] studied 800 apps running on 1,520 devices, and showed that background energy can be significant for apps: accounting for more than 50% of the total energy for 22.5% of the apps, with an average of 27.1% across all 800 apps. Martins, *et al.* [31] performed a set of controlled experiments, characterizing how apps and core components use specific features to enable background computing, and how this computing significantly affects energy use. They also provide a system mechanism, *TAMER*, to mitigate the impacts caused by background workloads based on monitoring, filtering and rate-limiting all background wakeup events. Google also notices this issue, so they introduce two features into Android 6.0 Marshmallow, as called `Doze` and `App Standby` [9]. `Doze` reduces battery consumption by deferring background CPU and network activity for apps when the device is unused for long periods of time. `App Standby` defers background network activity for apps with which the user has not recently interacted. However, their power-saving capability is limited due to the rigor triggering conditions. Differently, our work focuses the source of background activities: how and why these background apps are launched. Our analysis results can give deep insights into how to solve this problem elegantly, but we leave this as future work.

10. CONCLUSIONS

In this paper, we have conducted a large-scale analysis on the collusive relationships among popular Android apps, based on our static programming analysis tool AppHolmes. Our analysis results show that app collusion is quite prevalent, and is very related to apps' popularity, category, etc. We then dig into the root causes of such collusive behavior, and classify them into three main categories. Among these categories, Push Service contributes around 77% to the total number. We also quantify the impacts caused by app collusion, which proves to be non-trivial and can do significant harm to user experience.

11. ACKNOWLEDGMENTS

This work was supported by the National Basic Research Program (973) of China under Grant No. 2014CB347701, the Natural Science Foundation of China (Grant No. 61370020, 61421091, 61528201), and the Microsoft-PKU Joint Research Program. Felix Xiaozhu Lin was supported in part by NSF Award #1464357 and a Google Faculty Award. Xuanzhe Liu acts as the corresponding author of this work. Mengwei Xu and Yun Ma contribute equally to this paper.

12. REFERENCES

[1] Android aapt. `http://elinux.org/Android_aapt`.

[2] Android Permission Manifest. `https://developer.android.com/guide/topics/manifest/permission-element.html`.

[3] dex2jar. `https://github.com/pxb1988/dex2jar`.

[4] ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. `http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0098679`.

[5] Google's killer Android L feature. `https://gigaom.com/2014/07/02/googles-killer-android-l-feature-up-to-36-more-battery-life-thanks-to-project-volta`.

[6] Greenify. `https://play.google.com/store/apps/details?id=com.oasisfeng.greenify`.

[7] LBE Security master. `https://play.google.com/store/apps/details?id=com.lbe.security`.

[8] Modularity. `https://en.wikipedia.org/wiki/Modularity_(networks)`.

[9] Optimizing for Doze and App Standby. `https://developer.android.com/training/monitoring-device-state/doze-standby.html`.

[10] Xposed Module Repository. `http://repo.xposed.info/module/de.robv.android.xposed.installer`.

[11] Statistics and facts about the smartphone market in China. `https://www.statista.com/topics/1416/smartphone-market-in-china/`, 2012.

[12] Chinese Android markets download volumn share. `http://www.digi-capital.com/news/2015/04/android-makes-more-money-than-ios-including-china/`, 2015.

[13] Google Cloud Messaging. `https://developers.google.com/cloud-messaging`, 2016.

[14] Online discussion about App Collusion. `https://www.zhihu.com/question/38732700`, 2016.

[15] Online discussion about App Collusion. `https://www.v2ex.com/t/314390`, 2016.

[16] The astonishing facts about App Collusion. `https://www.zhihu.com/question/46614588`, 2016.

[17] UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey.html`, 2016.

[18] P. P. F. Chan, L. C. K. Hui, and S. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proc. of WISEC'12*, pages 125–136, 2012.

[19] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *Proc. of SigMetrics'15*, pages 151–164, 2015.

[20] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this app safe?: A large scale study on application permissions and risk signals. In *Proc. of WWW'12*, pages 311–320, 2012.

[21] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proc. of MobiSys'11*, pages 239–252, 2011.

[22] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proce. of USENIX Security Symposium 2011*, page 2, 2011.

[23] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. of USENIX Security Symposium, 2011*, pages 12–16, 2011.

[24] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: semantics-based detection of android malware through static analysis. In *Proc. of FSE'14*, pages 576–587, 2014.

[25] V. Gabale and D. Krishnaswamy. Mobinsight: On improving the performance of mobile apps in cellular networks. In *Proc. of WWW'15*, pages 355–365, 2015.

[26] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proc. of NDSS'12*, page 19, 2012.

[27] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng. Characterizing smartphone usage patterns from millions of android users. In *Proc. of the IMC'15*, pages 459–472, 2015.

[28] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proc. of ICSE'15*, pages 280–291, 2015.

[29] L. Li, A. Bartel, J. Klein, and Y. L. Traon. Automatically exploiting potential component leaks in android applications. In *Proce. of TrustCom'14*, pages 388–397, 2014.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of CCS'12*, pages 229–240, 2012.

[31] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background android apps to improve battery lifetime. In *Proc. of ATC'15*, pages 563–575, 2015.

[32] Y. Ng, H. Zhou, Z. Ji, H. Luo, and Y. Dong. Which android app store can be trusted in china? In *Proc. of the COMPSAC 2014*, pages 509–518, 2014.

[33] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, Q. Yang, and M.-S. Hall. Reducing smartphone application delay through read/write isolation. In *Proc. of MobiSys'15*, pages 287–300.

[34] D. Octeau, S. Jha, M. Dering, P. D. McDaniel, A. Bartel, L. Li, J. Klein, and Y. L. Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proc. of POPL'16*, pages 469–484, 2016.

[35] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proc. of ICSE'15*, pages 77–88, 2015.

[36] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proc. of USENIX Security Symposium, 2013*, pages 543–558, 2013.

[37] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn. Multi-app security analysis with FUSE: statically detecting android app collusion. In *Proc. of PPREW'14*, pages 4:1–4:10, 2014.

[38] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who killed my battery?: analyzing mobile browser energy consumption. In *Proc. of WWW'12*, pages 41–50, 2012.

[39] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proc. of CCS'14*, pages 1329–1341, 2014.

[40] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proc. of NDSS'13*, 2013.