

**Detecting, Characterizing, and Mitigating App Collusion on
Android Appstores**

| | |
|------------------|---|
| Journal: | <i>Transactions on Mobile Computing</i> |
| Manuscript ID | TMC-2017-07-0437 |
| Manuscript Type: | Regular |
| Keywords: | H.3.4.d Performance evaluation < H.3.4 Systems and Software < H.3 Information Storage and Retrieval < H Information Technology a, J.9 Mobile Applications < J Computer Applications |
| | |

SCHOLARONE™
Manuscripts

Review Only



1
2
3
4
5
6
7
8
9
10 Dear EIC, AE, and reviewers,

11
12 Please kindly find our manuscript enclosed, entitled "**Detecting, Characterizing, and**
13 **Mitigating App Collusion on Android Appstores**".
14

15 This manuscript is an extended version of our paper published in the proceedings of
16 WWW 2017, entitled "**Detecting and Characterizing App Collusion among**
17 **Third-Party Android Markets. Proceedings of the 26th International Conference on**
18 **World Wide Web, WWW 2017, pp.143-152**". On top of the conference version, we
19 made substantial extensions and revisions in this manuscript. The details include:
20
21

- 22
- 23 ● We apply our AppHolmes analysis tool over apps from Google Play and find that
24 the collusion behaviors are not so significant compared to those on Wandoujia,
25 Tencent, and Baidu (Section 5.1). Such a finding evidences that there are various
26 potential risks when downloading apps from third-party Android appstores.
 - 27 ● We reorganize the characterization of app collusion from five aspects: appstore
28 effect, popularity effect, category effect, asymmetry effect, and version effect
29 (Section 5.1-Section 5.5). We believe that such a characterization could provide
30 more comprehensive insights of app collusion.
 - 31 ● Third, besides the controlled study of simulating app collusion in the conference
32 version, we further evaluate the hidden cost caused by app collusion with a field
33 study that is based on the three-month real smartphone usage logs from 78
34 in-school volunteers (Section 7.2). The results confirm that the
35 collusion-introduced hidden cost is non-trivial and thus leads to undesirable poor
36 or harmful user experiences.
 - 37 ● Last but not the least, based on the analysis results, we propose a practical
38 approach, namely AppConan, to eliminating the collusion-introduced overhead
39 by balancing the list of apps that will be used by users and the system-wide
40 resource consumption (Section 8). More specifically, AppConan proposes a
41 time-aware, resource-consumption-wise, and personalized algorithm that can
42 accurately predict the app to be used by a user in a given time window, by
43 learning the app usage from the collected data in the field study. Hence,
44 AppConan can intelligently maintain the list of collusive apps by optimizing the
45 overall system resource consumption. The results demonstrate the effectiveness
46 and efficiency of AppConan.

47
48 The preceding extensions confirm that more than 30% of the submitted manuscript
49 consists of new content compared to the conference version.
50
51
52
53
54
55
56
57
58
59
60

Indeed, the current version of this manuscript is 15 pages, with an extra page compared to the 14-page policy of TMC. We are fully aware of such a policy but would like to request the peer review. We would like to pay for any additional page charge by complying with the TMC's publication policy.

Thank you for your upcoming efforts on reviewing our work!

Best regards,

Sincerely,

Xuanzhe, Mengwei, Boyuan, Yun, Xiaozhu, Yunxin, and Ziniu

Detecting, Characterizing, and Mitigating Collusion of Android Apps

Xuanzhe Liu *Member, IEEE*, Mengwei Xu, Boyuan Yang, Yun Ma *Student Member, IEEE*,
Felix Xiaozhu Lin, Yunxin Liu *Member, IEEE*, Ziniu Hu

Abstract—It is well-known that many apps can drain smartphones' compute resources at background and thus affect user experience. However, it yet lacks a clear understanding of the *cooperative behaviors* among background activities as well as a quantification of the consequences. In this article, we build tools to detect, characterize, and mitigate the app collusion. We first make an in-depth analysis of the so-called app collusion behaviors, i.e., 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 Google Play and three leading third-party Android appstores, we find that i) covert, cooperative behaviors in background app launch are surprisingly pervasive especially on third-party appstores, ii) most collusions are caused by shared services, libraries, or common interest among apps, and iii) collusion has serious impact on performance, efficiency, and security. We then develop a runtime scheduler namely AppConan that can learn the user interaction behaviors over apps, monitor the app collusion online, and automatically suppresses those colluded apps affecting user-interaction experiences.

Index Terms—Program Analysis; Mobile Computing; Community Detection

1 INTRODUCTION

Modern mobile applications (a.k.a., apps) are indispensable in our daily life. It is well-known that the apps often run background activities periodically to poll sensor data, maintain cloud activity, or update their local state [1]. While such background activities could be essential to the “always-on” interaction experience desired by users, they often have incurred substantial overhead which may be undesirable. For example, a recent study [2] 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.

Essentially, 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 never interact with or intend to use such apps. More seriously, the users may be never aware of the launched apps.

In this article, we use the term, *app collusion*, to denote such **user-unaware cross-app compositional launch**¹. Compared to the background activities that are triggered 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 can grow as one user installs more apps on her mobile device.

Motivated by anecdotes [3] [4] [5] about serious app collusion on appstores, we present a systematic study which focuses on four 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. (4) mitigating the undesired user experiences caused by app collusion.

To enable the preceding goals, we first build *AppHolmes*, a static analyzer that not only retrofits existing techniques for analyzing Inter-Component Communications (ICC) [6] [7] [8] [9] [10] [11] but also infuses domain knowledge for identifying important collusion categories, which would have been omitted by existing tools.

With the AppHolmes tool, we have studied over 10,000 Android apps collected from Google Play along with other three leading third-party Android appstores in China, i.e., Baidu², Tencent³, and Wandoujia⁴. These appstores are the major channels for distributing Android apps in mainland China, each of which has hundreds of millions of users. While the app collusion is not so significant on Google Play, to our surprise, we find that the app collusion prevails among Android apps from all three third-party appstores – 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*. Furthermore, we quantify the hidden cost and security risk of app collusion over typical apps.

Finally, we propose a runtime system-wide approach, namely *AppConan*, which can help mitigate the undesired

• Xuanzhe Liu, Mengwei Xu, Yun Ma, and Ziniu Hu are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871. Email: {liuxuanzhe, xumengwei, mayun}@pku.edu.cn.

• Felix Xiaozhu Lin is with Purdue University. Email: xzl@purdue.com.

• Yunxin Liu is with Microsoft Research. Email: yunxin.liu@microsoft.com.

• Boyuan Yang is with Beijing University of Technology. Email: bigwhite-beer@gmail.com.

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 §6.

2. <http://shouji.baidu.com/>

3. <http://android.myapp.com/>

4. <http://www.wandoujia.com/>

1 user experiences caused by app collusion, by learning the
 2 user interaction behaviors over apps and establishing a
 3 personalized decision-making model to efficiently control
 4 the app collusion. Our major findings are as follows:

- 5 • We design the AppHolmes tool and present the first
 6 large-scale study of collusive behaviors of apps and
 7 demonstrate the prevalence of app collusion among
 8 third-party Android appstores. On average, each app
 9 colludes with 76 other apps on Wandoujia appstore.
 10 We have similar observations on the Tencent and
 11 Baidu appstores.
- 12 • We characterize the collusion behaviors among apps
 13 from five effects. (1) **Appstore effect**: an app may
 14 exhibit different collusion behaviors when downloaded
 15 from different appstores, and the common reason is
 16 that app developers intentionally adapt their apps
 17 in order to conform to the appstore-specific policies;
 18 (2) **Popularity effect**: popular apps are more likely
 19 to collude with more apps; (3) **Category effect**: apps
 20 from different categories can behave quite differently
 21 in colluding other apps, e.g., among the top 1,000
 22 apps from Wandoujia appstore, each shopping app
 23 colludes with 174 other apps on average, almost 7
 24 times more than game apps; (4) **Asymmetry effect**:
 25 popular apps are more likely to be launched by other
 26 apps rather the other way around. On average, for a
 27 top-50 app A in the Wandoujia appstore, the apps
 28 that may launch A in the background are twice as
 29 many as the apps that A may launch in the back-
 30 ground; (5) **Version effect**: for a specific app, the
 31 number of collusion keeps increasing with its version
 32 evolves.
- 33 • We reveal the root causes of app collusion. Sharing
 34 third-party push service contributes primarily to the
 35 app collusion. We have discovered that *all* the 8 most
 36 popular third-party push services can cause collusive
 37 relations, which constitute 77% of the total app col-
 38 lusion discovered by us. In addition, app developers
 39 may be never unaware of the possible collusion. This
 40 is because the collusion behaviors may root in the
 41 third-party SDKs or services that are shipped with
 42 poor documentation, used by app developers without
 43 much understanding, and integrated as black
 44 boxes.
- 45 • We quantify the non-trivial hidden cost and security
 46 risks introduced by app collusion. A controlled ex-
 47 periment with 40 popular apps indicates that for ev-
 48 ery single foreground app, the background collusion
 49 can lead to additional 202 MB memory use, extra
 50 8.9% CPU usage, and 16 additional high-risk per-
 51 missions on average. With app collusion present, the
 52 user perceives 2× latency in launching foreground
 53 apps and the off-screen battery life is reduced by
 54 57%. A field study over the two-month app-usage
 55 logs collected from 78 volunteer students also evi-
 56 dences the overhead caused by app collusion, which
 57 can reach 9.2% CPU and 288 MB memory daily on
 58 average.
- 59 • We design the AppConan tool along with a machine
 60 learning algorithm that can help eliminate the app
 61 collusion-introduced hidden cost by deriving the
 62 user interaction behaviors on their mobile devices.
 63 Our algorithm can accurately predict those apps

that an user is likely to interact with in a specific time window, and thus can decide whether the apps colluding with current active apps should be launched or not by estimating the potential system-wide overhead. We evaluate the approach over the same collected user behaviors, and the results show that AppConan can save about 8.2% CPU usage, 201 MB memory usage, and 8.4 MB daily network usage per day on average.

Part of this work has been published on our previous WWW paper [12]. Compared to the conference-version work, we make various significant extensions. First, we apply our AppHolmes over apps from Google Play and find that the collusion behaviors are not so significant compared to those on Wandoujia, Tencent, and Baidu (§5). Such a finding evidences that there are various potential risks when downloading apps from third-party Android appstores. Second, we evaluate the hidden cost caused by app collusion not only by a controlled study, but further demonstrate the cost with a field study of 78 real-world users (§7.2). Last but not the least, we propose the practical AppConan approach to eliminating the cost introduced by app collusion via balancing the list of apps that will be used by users and the system-wide resource consumption (§8).

The remainder of the article is organized as follows. We present the background knowledge in §2. We formally define the problem in §3. We describe AppHolmes, our static analyzer for detecting app collusion in §4. We present the analysis of collusion of popular apps from three major third-party Android appstores, reveal their root causes, and quantify the resultant hidden cost in §5, §6, and §7, respectively. We discuss the implications and limitations of our work in §9, discuss prior work in §10, and conclude the work with future outlook in §11.

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 *IntentFilters* that are declared in manifest files. The primary pieces of information in an *Intent*, is **action** that specifies the general action to be performed and **data** to be operated on. Other additional attributes include **category**, **type**, **component** and **extras**.⁴ 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 information⁵ for the system to determine which components to run for that intent. Developers need to declare all components and *IntentFilters* which specify the *Intents* each component can receive in manifest file.

A match between an *Intent* and an *IntentFilter* enforces the following conditions to be valid: the action and the category must match, and the data (both the data type and data scheme+authority+path if specified) must match as well. An *Intent* could have multiple categories, but can have only one action. If the filter specifies no actions, it will match only the Intents that do not contain any action. Extra categories in the filter that are not in the Intent can not make the match fail.

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, $app \text{ collusion} = \langle app_i, app_j \rangle \in APP \times APP$, 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 sent to the system to wake up that service (line 7). Since a service has no UI elements, users have no idea of its launch.

Listing 1: A code sample of app collusion

```
public class classA{
    public void methodA(Context context) {
        // com.example.target is not the
        // current package;
```

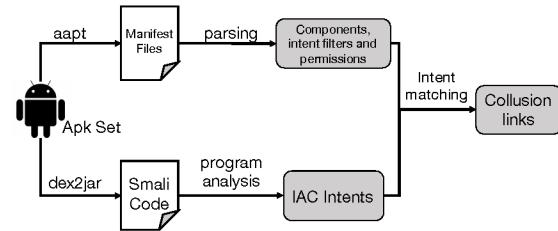


Fig. 1: The workflow of AppHolmes.

```

Intent intent = new Intent();
intent.setClassName("com.example.target",
    "com.example.target.otherService");
intent.setAction("some_action");
context.startService(intent);
}
  
```

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:

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

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

Intuitively, *In-Collusion* describes how many other apps can wake up an app, and *Out-Collusion* describes how many apps this app can wake up.

4 DETECTING COLLUSION WITH APPHOLMES

To enable a 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) [13], and 2) smali code via dex2jar [14]. 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 learn the lessons from the ideas of IC3 [11], 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 for only 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 SDK⁵ 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 accurate results. However, PRIMO [15], which is a probabilistic model to accurately match ICCs, points out that a conventional matching strategy can cause lots of false positives (95.6%). Learning 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 if and 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 if and 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 the ground truth by developing an Xposed [16] 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 can reach 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 §4, we then want to figure out which apps are likely to exhibit collusion behaviors. To this end, we perform a large-scale analysis of popular apps from Google Play and other three leading third-party Android appstores in China: Wandoujia, Tencent, and Baidu. More than 2 million apps are available on each of these third-party appstores. For each app, the appstore provides the binary .apk file and the number of downloads. We crawled top 10,000 apps from these appstores during July 2016 - August 2016.

5. A popular third-party push service. More details in §6.

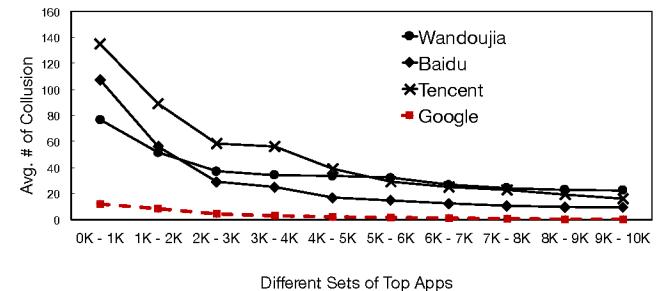


Fig. 2: The average count of apps that are collusive with a given app on different Android appstores. Each set includes 1,000 apps, sorted by app downloads.

5.1 Appstore Effect

We first report the overall results by applying AppHolmes over the top apps from these four appstores, as shown in Figure 2. The first and the most surprising observation is that the app collusion behavior is very significantly different between the apps from Google Play and those from other third-party Android appstores. It is observed that the app collusion number is much higher in other third-party appstores. For example, Google Play has only 11 app collusion among the top 1,000 apps averagely, while Baidu appstore has 107 on average, almost 10 times more than Google Play. Additionally, the app collusion is a common phenomenon of apps that are released on third-party Android appstores. 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. Such a finding is quite interesting and we explore the root causes that lead to the difference later in § 6.

Next, the overall trend of collusion is the same in all three third-party appstores. However, the divergence of average collusion number of top 1,000 apps is quite large, as Tencent appstore is 138 while Wandoujia is only 77. Such a divergence draws our attention and raises the interest to interpret the potential reasons. We identify two major factors. First, app rankings can be different on different appstores. We find that there are only 512 common apps among top 1,000 apps from both Baidu and Tencent appstores. For example, it is observed that many apps developed by Qihoo and Tencent companies are ranked high on Tencent appstore but not available on Baidu appstore. Second, even the same set of apps can behave differently on different appstores, since developers may customize their apps to fit some policies made by different appstores.

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

As the collusion is a typical phenomenon of third-party Android appstores, the rest of this article mainly presents the results of Wandoujia due to page limit. It should be mentioned that the results from other two appstores are quite similar to those from Wandoujia. The complete analy-

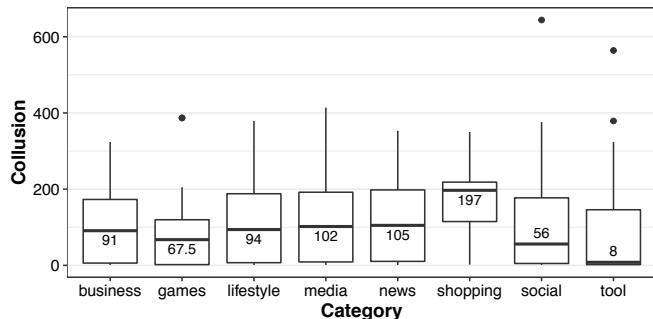


Fig. 3: The number of collusive apps for a given app from different categories (Wandoujia top 1,000).

sis results have been open and available online⁶.

5.2 Popularity Effect

Another interesting observation from Figure 2 is that those popular apps are more likely to be collusive ones. 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 QQ⁷ 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. These apps owned by the same company are usually collusive with each other, to share services or user data.

5.3 Category Effect

We then want to know whether an app's functionalities could be a factor affecting the collusion. To this end, we cluster the apps according to their categories. Since Wandoujia, Baidu, and Tencent share the similar categorization systems on their appstores, we report only the results of Wandoujia.

Figure 3 implies the relation between the number of collusion and app categories. As observed, shopping apps have the largest number of collusion (197). In contrast, tool apps have very few collusion (8).

To account for this difference, we identify two possible reasons, 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 the tool category, the percentage of apps integrated with push services is only 11%. This difference stems from the unique characters of different categories of apps.

Second, the most popular shopping apps are mostly from those large vendors, while game and tool apps are mostly from small and medium vendors or even individual developers. For example, we observe that out of the 51

6. <http://www.mobisaas.org/projects/appholmes/>

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

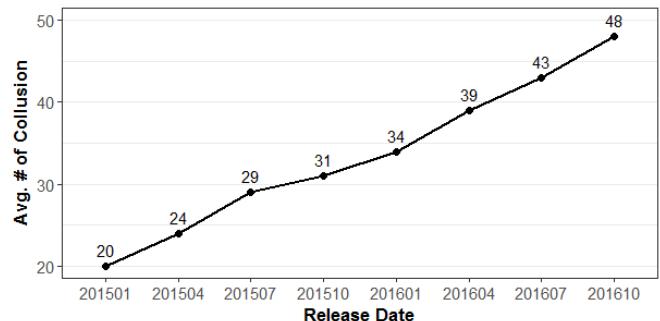


Fig. 4: The relation between the collusion number and app version.

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.

5.4 Asymmetry Effect

Although 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, payment, 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 §6.

5.5 Version Effect

Finally, we also study how the collusion behavior evolves as the apps are updated. We crawl all the history apk files for top 100 apps among 10 categories (thus 1,000 in total), and study how the app collusion number changes during two recent years (2015 - 2016). Figure 4 shows the analysis results for 798 apps, as we filter some apps that were first released after 2015. Apparently, the number of app collusion keeps increasing as the apps are updated. For instance, one app colludes with other 20 apps in Jan. 2015 averagely, and the number is doubled since Jul. 2016 and still becomes larger with time. Such an observation can somewhat attribute to the fact that developers often enrich the features and functionalities of their apps in new versions, so that the opportunities of app collusion increase accordingly.

6 ROOT CAUSES OF APP COLLUSION

Next we present the major causes of app collusion and describe how apps can be clustered in different communities based on their collusive behaviors.

| 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 |

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

6.1 Classifications of App Collusion

To have better understanding on 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) [17]. To implement GCM, the Android OS runs a separated system process that receives messages from the cloud and delivers them to corresponding apps. Then, apps can deal with these messages as developers wish. Using GCM to receive push messages is recommended as “best practice” [18], since it saves memory by having only one push process running, and avoids apps unnecessarily running in background to wait for messages.

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 out 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. Also, most of these collusive relations are forced by SDKs, with only one exception, jPush. For apps using jPush, developers can choose whether they want their apps to be involved with the collusive relation with other jPush-enabled apps. 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.

Although such “one for all, all for one” feature increases the chance of immediately receiving messages from the cloud, it also introduces hidden cost to system and developers. Besides the significant system cost, which will be quantified in §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 app⁸ 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 and unjustified collusion emerges due to the 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 to 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 exactly 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 §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 one another, which contributes around 5% to the overall collusion. 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 indicates that they share this service with one another via `bindService()`, e.g., using users’ search keywords from a search-engine app can help recommend products on a shopping app. Such a cooperation could help provide more powerful features by combining user data generated in different apps, but it also inevitably causes unnecessary or even undesired resource consuming as demonstrated in §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 reader app (`com.ophone.reader.ui`), we find that 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 collusion network of top 1,000 apps from the preceding three third-party appstores. As shown in Figure 5, 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 [19] algorithm to layout the graph, and use Modularity [20] to divide all nodes into different communities (denoted by the node colors).

As we can see in these graphs, the overall network graph is a bit complicated, and is primarily clustered into

8. A popular news client in China

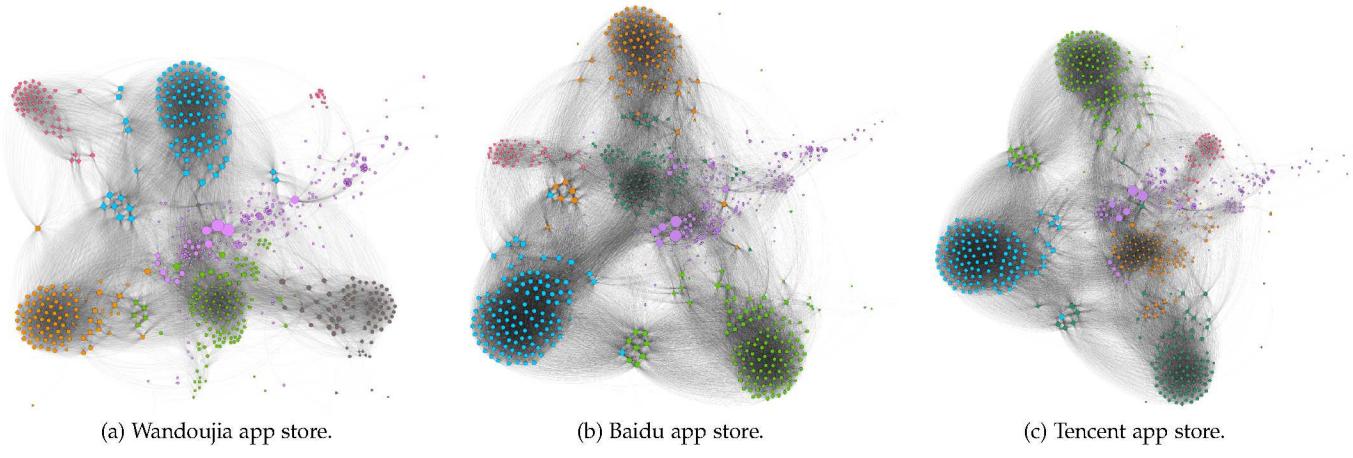


Fig. 5: App communities based on detected collusion among top 1,000 apps from three third-party appstores. Each node represents an app while each edge represents the collusive relations. Different colors represent different communities. Different node sizes represent different number of In-Collusion.

six communities with 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. Take the graph of Wandoujia appstore as an 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* 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 a gravity is caused by the functional SDKs.

Even within a given community, apps may establish 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

We demonstrate the hidden cost of app collusion via both controlled experiments over a set of popular apps and a field study from real-world users. The results show that app collusion has significant impacts on user-perceived latency, energy consumption, security risk, as well as system resources including CPU and memory.

7.1 Controlled Experiments

We first describe the methodology and results of our controlled experiments, from which we quantify the cost from app collusion of popular apps.

| | App Name | Category | # of Downloads |
|-----------------|----------------------|----------|----------------|
| Foreground apps | QQ | Social | 560 M |
| | Taobao | Shopping | 460 M |
| | Sina Weibo | Social | 210 M |
| | Baidu Map | Map | 200 M |
| | iQiyi | Video | 190 M |
| | Baidu Searchbox | Tool | 180 M |
| | Baidu Brower | Browser | 65 M |
| | PPS TV | Video | 56 M |
| | Shuqi Reader | Books | 40 M |
| | Nuomi | Shopping | 18 M |
| Background apps | WeChat | Social | 530 M |
| | QQ Brower | 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 |

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

7.1.1 Experiment Methodology

App Selection. According to a study of millions of Android users [21], 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

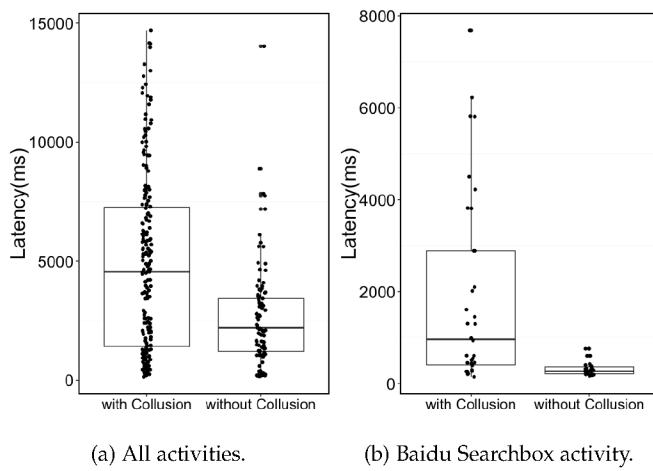


Fig. 6: Launch latency of Foreground apps with and without the presence of app collusion.

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 [22] tool to automatically run over Foreground apps.

- **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 app¹⁰.

- **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 perceptible to users, and *Exp I* to gain more comprehensive and system-wide detailed results. 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 collusion, we enforcedly stop all Background apps in advance so that these apps cannot be awoken by system broadcasts.

7.1.2 Experiment Results

Overall, *Exp I* shows that 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

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.

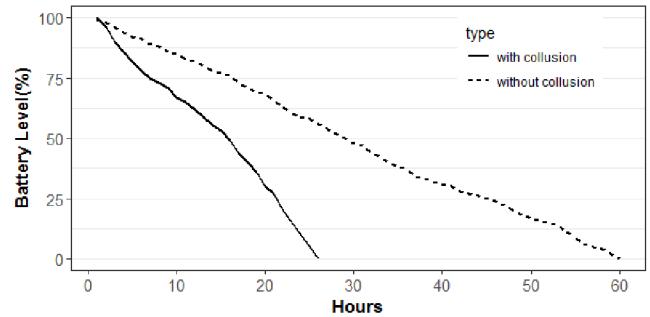


Fig. 7: The impact of app collusion on battery life, estimated when the screen is always kept off.

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. We report the results as follows.

- **2x user-perceived latency.** As illustrated in Figure 6(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 6(b) shows more detailed results for a search page activity from Baidu Searchbox. Obviously, the launch time is quite stable around 400 ms without app collusion. When app collusion occurs, 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 re-launch the app, and such an action 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 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 7, 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 awoken by app collusion. Such a significant impact is consistent with prior knowledge: Google already determines that for every second of "active" use on a typical phone, the standby time is reduced by two minutes [23]. 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 high-risk permissions required.** Android permission strategy has been studied and criticized for a long period of time [24] [25]. Excessively granting permissions to apps can lead to potential privacy leak, especially for those officially marked "dangerous" permissions by Android OS [26], e.g., contacts, SMS, and camera. In our experiment, we quantify the number of dangerous permissions owned by Background apps that are woken

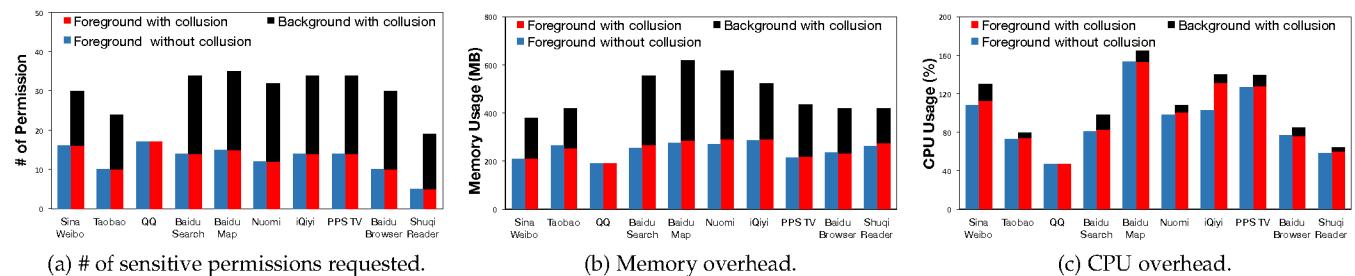


Fig. 8: A breakdown of extra system resource usage, without and with the presence of app collusion.

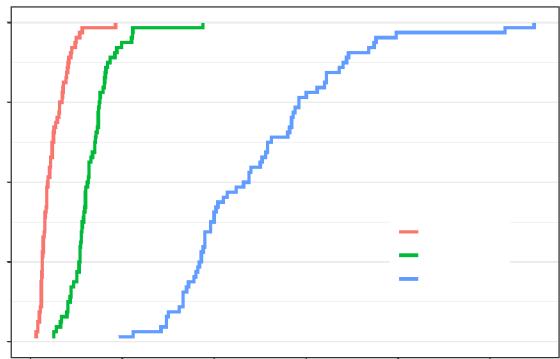


Fig. 9: Distribution of three numbers among different users: number of apps *used* each day on average, number of apps *launched* in background each day on average, and number of all *installed* apps (excluding all pre-installed apps).

up via app collusion. As shown in Figure 8(a), background apps that are woken up via app collusion require many extra dangerous permissions (16 averagely), e.g., Shuqi Reader asks only 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 at all. For instance, `android.permission.ACCESS_COARSE_LOCATION` is used to access GPS for location information, which is not necessarily required by Shuqi Reader, but required by the Background apps colluding with Shuqi Reader. It implies that app collusion can increase the probability of privacy leak, and also causes more energy consumption since it turns on more sensors on (GPS).

• 202 MB memory usage overhead. Figure 8(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 the collusion-introduced memory overhead is non-negligible. The average memory overhead caused by app collusion is 202 MB, and the worst case, Baidu Map, can wake up 13 Background apps which consume 334 MB memory. Such an overhead is even larger than the memory usage of the Foreground app itself (286 MB).

- 8.9% CPU usage overhead.** Similarly, we report the CPU overhead in Figure 8(c). The average CPU overhead caused by 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).

7.2 User Study

The preceding control study can evidence the possible hidden cost from collusion. We then figure out how app collusion can affect real-world user experiences by collecting app launch and interaction logs from volunteers.

7.2.1 Data Collection

We develop a system-wide tool to monitor the device's running status and record the following logs. First, the tool records the app that a user interacts with in foreground. To this end, the tool probes the Android stack every 1 second. Second, the tool records the resources usage of all background apps, including memory, CPU, and network, and the frequency every 30 seconds. Third, the tool records the full list of installed apps on the device. Fourth, the tool records the system events that indicate the change of interaction context, including network connection (cellular/Wi-Fi/OFF), timestamp, screen status (ON/OFF), battery charging status (ON/OFF), and battery level (0 to 100).

We recruited college student volunteers for the data collection via an internal social network site in Peking University. To protect user privacy, all involved users are anonymized and assigned with a unique ID. We finally had 78 student volunteers who agreed with the collection statements, and we asked them to install our monitoring tool on their Android smartphones. To keep the collected data clean, we help the volunteers disable all installed apps that can receive system broadcasts [27]. Such a goal can be easily done through the functionality provided by some third-party Android management apps [28] [29]. As a result, apps can be launched only by explicit users' operation or via app collusion. The data collection process lasted for three months from Sept. 2016 to Nov. 2016, and we finally collected 914,189 records with data size of 7.1 GB. The data collection and analysis process was conducted with the IRB approval by the Research Ethic Committee of Institute of Software, Peking University. We plan to release the collected data along with this work is published.

7.2.2 Results Analysis

For every single user, we calculate the average number of used apps per day, the average number of apps launched in

| resource type | consumption | percentage |
|-----------------------|-------------|------------|
| CPU usage | 9.2% | 15.3% |
| Memory usage | 288 MB | 18.9% |
| Cellular Data Traffic | 11.5 MB | 32.6% |

TABLE 3: The resource consumed by the background apps that are launched via app collusion. **Percentage** presents the proportional consumption to the resource consumed by all foreground and background apps. The network usage only include the data transferred under 4G or LTE (excluding WiFi).

background, and the number of all installed apps (excluding all pre-installed apps whose package name starts with `com.google`, `com.samsung`, etc). We apply our AppHolmes to match the background apps that are launched by collusion. We plot the distribution of the information in Figure 9. On average, a user interacts with only 9.5 apps per day, accounting for a very small portion (8.2%) of the number of all installed apps (116) on device. However, as observed in Figure 9, there are 31 apps launched by app collusion in background per day, almost 3 times more than the number of apps that are actually used by users.

We then calculate the system resource consumed by those apps that are infrequently used by users but launched in background via app collusion. As shown in Table 3, around 9.2% CPU cycles and 288 MB RAM are used by the collusive apps, which contributes 15.3% and 18.9% to the overall CPU/memory usage at system level, respectively. Such a non-trivial undesired additional overhead is even more serious compared to results reported from the control study. In other words, we can clearly state that collusion cannot be neglected for real-world smartphone usage, since the system resource overhead can further lead to poor user experience, e.g., UI lagging and battery drain as shown in previous subsection. In particular, the average cellular data traffic consumed by the collusive apps can reach 11.5 MB per day. Given the limited data plan held by college students in China, such a waste of data traffic can be a significant issue.

8 APPCONAN: ELIMINATING COLLUSION-RELATED HIDDEN COST

Motivated by the high prevalence (§5) and non-trivial hidden cost (§7) of app collusion, we aim to further propose an online optimization approach, namely AppConan, to eliminating the undesired overhead caused by those app collusion.

The major challenge of our goal is that we don't want to interfere too much with the original apps' normal behavior when suppressing the collusion behavior. Indeed, app collusion can have positive aspects sometimes, among which the probably to-be-used apps can be pre-launched by the currently used app in the background, so that users can quickly open and interact with this app later (we call it "warm launch" in this article). In other words, simply cutting-off all app collusion cannot be directly adopted since it can compromise the user experience. For instance, as explained in §6, disabling all app collusion can make apps fail to receive notifications from the cloud. As a result, users could miss some important messages which they want to subscribe from some apps. Therefore, we need a more intelligent and moderate mechanism to identify those "useless" collusive apps that do not positively contribute to user experience, so that we can disable or delay their launch.

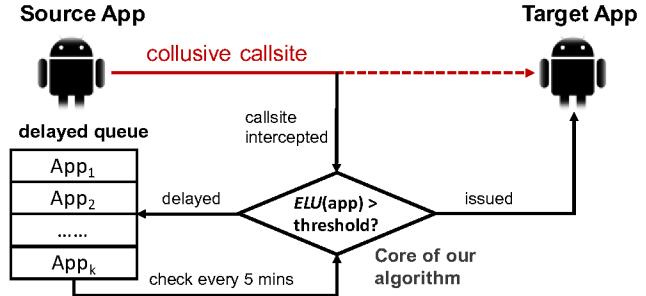


Fig. 10: Workflow of AppConan.

8.1 AppConan Design

Based on the preceding analysis, the design principles of the online optimization approach proposed in this article are as follows.

- **Time-Aware.** If a collusive app is unlikely to be interacted by user within a given time window (e.g., 5 minutes), it should not be launched in background via app collusion. On the contrary, if the collusive app is likely to be used by the user, we view such a collusion as "useful" and allow its running in background. This principle is based on the intuitive observation: after being launched in the background, an app can prepare its data in advance and response to user action more quickly and smoothly.
- **Resource-Consumption-Wise.** The more resources an app consumes in background, the more probably it shall be disabled or delayed from being launched by app collusion. Indeed, this principle requires the trade-off against the preceding principle, i.e., keeping more apps alive and saving system resources.
- **Personalized.** The runtime optimization should be highly personalized for different users and configurable for different hardware device specifications. Such a flexibility is necessary as users have different usage patterns of apps and devices have different amount of available system resources.

To address the preceding three principles, we design AppConan as shown in Figure 10. The core component of AppConan is a run-time decision maker, which determines whether an app collusion should be delayed or issued when it happens. The decision is made by calculating the ELU value of the app to be launched, and compares it to a configurable threshold \mathcal{T} that can be defined by either developers or users. The ELU value indicates how much the app x is *eager to be launched by the user*, and is the key indicator of AppConan. It is directly derived from a probability model \mathcal{P} and a resource weighting model \mathcal{R} in the following formula. We will explain the definition and the design details of \mathcal{P} and \mathcal{R} later in this section.

$$ELU(x, context) = \frac{\mathcal{P}(x, context)}{\mathcal{R}(x)}$$

If the ELU of an app is higher, we enable the collusion to launch the app. Otherwise, we delay this app in a queue that maintains a list of apps to launch. In this article, we update the ELU of all apps in the queue for every 5 minutes, and enable the collusion if the updated ELU is higher compared to the threshold. We then explain how the two models are defined.

8.1.1 Probability Model

$\mathcal{P}(x, \text{context})$ means the probability of app x being used by users in the foreground within a given time window τ from now on given the current context. A lower $\mathcal{P}(x)$ value indicates less necessity of x being launched in the background since it is unlikely to be used by users. Therefore, we can delay this app to save more resources.

Intuitively, such a prediction problem can be treated as a binary-classification problem, i.e., given the current context information and a particular app, whether this app will be used or not within a given time window τ . A straightforward approach is recording the actual user interaction after app collusion as positive instances and the non-happened interaction as negative instances to train a classifier. However, due to the sparsity of the positive instances in the whole data span (any context at an arbitrary time point combined with any possible app that can be an observed instance), it is hardly known which app will never be used given a particular context.

Theoretically, such a problem is known as a typical One-Class Classification problem [30]. Here we adopt a state-of-the-art algorithm PU proposed by Liu et al. [31] [32] to solve the problem. The basic idea of the algorithm is to use the positive data to identify a set of informative negative samples from the whole data span, in order to keep the positive and negative samples balanced.

The overall classification process is described in Algorithm 8.1. At the beginning, the algorithm generates the whole data span U by enumerating every combination of target apps a and context. Then the algorithm identifies the informative negative samples according to the positive training data. For prediction, given the current context c , we can calculate the probability of any candidate target app ta that will be used by users within a given time window τ .

```

34 Input: PU Method  $\mathcal{PU}$  , Positive dataset  $P$ , Candidate
35   App Set  $S = (a_1, a_2, \dots, a_N)$ 
36 Output: Recommendation List  $L$ 
37 1  $U \leftarrow \text{generate\_whole\_data\_span}()$ 
38 2  $N \leftarrow \text{extract\_negative}(U, P)$  according to [31]
39 3  $\mathcal{PU}.\text{classifier}.fit(P \cup N)$ 
40 4 foreach  $ta \in S$  do
41   5 |  $\text{prob}[tp] = \mathcal{PU}.\text{classifier}.predict\_probability((c, tp))$ 
42   6 end
43   7  $L \leftarrow \text{sort\_by\_probability}(P, \text{prob})$ 
44 8 return  $L$ 
```

Algorithm 8.1: One-class classification algorithm for prediction.

To achieve effective prediction results, we carefully choose some meaningful context features, including information of the previously launched app (id, elapsed time, and category), network condition, battery level, and local system time. We adopt a variant of the PU algorithm, namely SPY+SVM, to accomplish the classification task.

8.1.2 Resources Weighing Model

$\mathcal{R}(x)$ measures how many system resources app x can consume in background. A high value of $\mathcal{R}(x)$ indicates that we should more aggressively disable or delay x from being launched since it consumes more resources than other apps. In detail, it is calculated as follows,

$$\mathcal{R}(x) = \sum_{i=0}^n \mathcal{W}_i \cdot \frac{r_i(x)}{R_i}$$

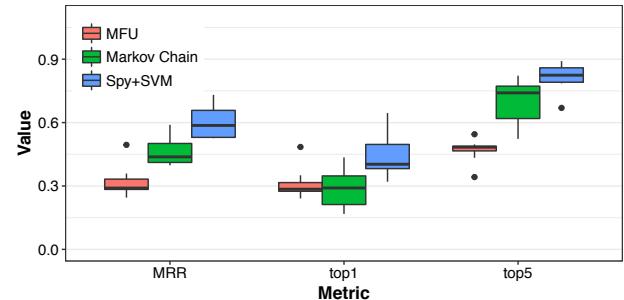


Fig. 11: Prediction results with different algorithms.

where $r_i(x)$ is the consumption of the i -th resource type for app x , R_i is the average consumption of the i -th resources type for all installed apps on the device, and \mathcal{W}_i is the importance weight of the i -th resources type. In this article, we consider three resource types: CPU, memory, and network usage.

8.2 Choosing Parameters

The online optimization approach has three sets of configurable parameters: resources importance weight \mathcal{W}_i , threshold \mathcal{T} , and time window τ .

Importance Weight. \mathcal{W}_i indicates how significant the i -th resources is under the current conditions of the user's device. For instance, if the foreground app is very CPU-intensive, the weight of CPU resources should be assigned with a higher value. If the app is running out of network data, the weight of network needs to be higher accordingly.

Threshold and Time Window. The parameters \mathcal{T} and τ are designed for the flexible trade-off between keeping more apps alive and delaying more apps to be launched to save system resource. A higher threshold \mathcal{T} and a smaller time window τ indicates more likelihood to disable or delay the app collusion.

8.3 Evaluation

To evaluate how well AppConan performs, we still use the collected data (§7.2) to train our probability model \mathcal{P} and evaluate its accuracy. Then, we demonstrate how much system resources AppConan can save along with very small impacts on the user experience.

8.3.1 Accuracy of Probability Model

We first evaluate the performance of our proposed prediction model, i.e., whether it can precisely predict whether an app will be used. We use two metrics to measure the model's performance, i.e., the accuracy and the Mean Reciprocal Rank (MRR) score. The MRR of a recommendation list is the multiplicative inverse of the rank for the correct answer, which is a promising indicator in the ranking problem.

We compare the proposed model with two straightforward algorithms including the Most-Frequently-Used (MFU) model that ranks the apps according to their usage frequency, and the *Markov Chain* based model that ranks the apps by the static transition probability for a target app to be navigated from the current app.

We use 30% of the collected data as the test set, and the remaining 70% data as the training set. We train the model for every single user and evaluate the mean accuracy. Fig 11

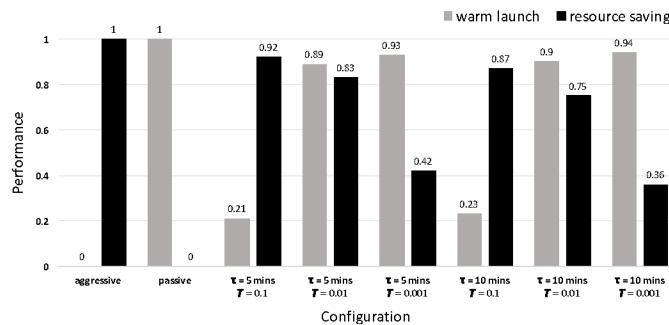


Fig. 12: Evaluation of our proposed algorithm in consideration of how it can keep apps launched by users quickly (warm launch) and can save the system resource. Both metrics are normalized. We have two ground truth: *aggressive* means disabling all app collusion, while *passive* means keeping the default strategy as Android and killing no app collusion.

illustrates that our proposed model outperforms the naive *MFU* and the *Markov Chain* algorithms, with its average top-1 accuracy of 0.43, top-3 accuracy of 0.74, and the average MRR score of 0.60. Such a result indicates that the actual most used app can be accurately predicted by our model.

Our proposed method is targeted for predicting whether an app will be used within a given time window. Indeed, there have been some efforts on app-usage prediction [33] [34] [35]. Our prediction result is comparable to these state-of-the-art methods. More specifically, *APPM* [33] achieves around 80.9% accuracy for top-5 prediction, while our model achieves around 81.7%. In the future, we plan to incorporate these existing prediction strategies into our model to further improve the accuracy.

8.3.2 Performance of AppConan

We then evaluate the overall performance of AppConan from two aspects: 1) How it affects the warm launch? Inaccurately delaying apps from being launched can lead to slow cold launch, which should be avoided. 2) How much resources AppConan can save? Here we simply present the saved resources for CPU, memory, and network data on average.

We emulate the app behavior by performing AppConan on the collected data. The result of warm launch can be simply obtained via looking up the background app list every time when user launches an app. To figure out how many resources are saved, we check the amount of resources consumed by the apps that are delayed from being launched, at every 5-minutes interval. For the configurable parameters, we simply set W_i ($i=1, 2, 3$) for all three types of resources equally as 0.33, and iterate the different values of time window τ and the threshold T .

As observed from Figure 12, when setting the parameters properly (e.g., $T=0.01$, $\tau=5$ mins), AppConan can significantly help save system resources (83%) consumed by the background apps that are launched via app collusion, while imposing quite little on the user experience (keeping 89% warm launch). In detail, this configuration can save about 8.2% CPU usage, 201 MB memory usage, and 8.4 MB cellular usage per day on average. Essentially, the key reason of such an improvement with little side-effect is due

to our probability model \mathcal{P} , which accurately helps identify “useless” app collusion at runtime.

Figure 12 also indicates that different configurations can lead to high divergence among warm launch and resource saving. Instead of fixing T and τ as constants, a more intelligent way is making the two parameters adaptive to user preferences and device status with better trade-offs. For example, for users who care more about system resource, or for devices that are running out of battery, we can dynamically assign higher T and lower τ .

In summary, we have demonstrated current AppConan is effective to optimize the undesirable overhead caused by collusion.

9 DISCUSSION

So far we have demonstrated the characteristics, root causes, hidden cost of app collusion and how the useless collusion can be mitigated. This section presents implications derived from app collusion, and discusses the limitations of our approach.

9.1 Implications

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 to uninstall apps that are seldom used, especially apps that have extensive collusive relations to other apps. 2) choose a proper Android appstore to download apps, since the same apps can behave differently on different appstores as aforementioned. 3) carefully disable the app collusion via some third-party apps like Greenify [36], LBE [28].

However, these strategies have two obvious defects. First, such all-or-nothing approach is not flexible since the necessity of app collusion can change dynamically. In some cases, app collusion can help achieve some app functionalities or preload the apps into memory as users wish. Simply disabling all app collusion may harm the user experiences in these situations. Second, many users lack the knowledge and patience to configure which kinds of app collusion should be disabled. In this way, adopting mechanisms such as AppConan can be helpful to end-users.

- **To OS vendors.** The module of inter-app communication in Android should be improved to mitigate the hidden cost caused by app collusion. Here we provide two suggestions. On the one hand, when app collusion is detected to launch third party apps without user awareness, 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 similar to AppConan for better managing 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 appstores.** Nowadays many Android appstores 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 appstores, 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.

9.2 Limitations

Although AppHolmes is carefully designed and implemented, it has some 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. Second, AppHolmes cannot handle cases where string values are generated at runtime. For example, we observe that apps using Getui will store their package names and some other metadata into a certain sdcard folder, therefore these apps are able to wake up each other via explicit *Intent* with package name. Such cases are almost impossible to be handled by classical static program-analysis techniques. We manually tackle some of these cases observed in our experiments, but it can't be applied universally.

Indeed, our current analysis focuses on third-party Android appstores only in China. The current results may not be well generalized to apps that are released on other appstores. For example, apps downloaded from Google Play indeed have less collusion behaviors. However, given the huge amount of apps and mobile users of the investigated third-party appstores, our study and results in this paper can still be meaningful.

Our proposed AppConan aims at eliminating the hidden cost of app collusion, and has various aspects to improve. First, the metric *ELU* used in our algorithm is not sufficient enough to capture the usefulness of the class of apps that are intended primarily to run in the background, e.g., data-backup apps or health monitoring apps. For instance, Dropbox is used by users to upload files to the server and such an operation shall happen on a scheduled frequency (e.g., every 5 minutes or whenever a file is changed), but the user may only infrequently access the Dropbox app when the screen is turned on. In this case, the *ELU* cannot be assigned to be low yet the background upload activities are supposed to be useful. For such apps, we plan to provide a feature where users can maintain a whitelist to bypass these apps. Second, our current algorithm can be further improved by including more features when constructing the predication model \mathcal{P} . For example, the information that an app is trying to launch other apps via collusion can indeed help justify the behavior, but our tool cannot catch such details as it requires the root privilege of a device.

10 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 [25], [37], [38], [39],

sensitive data theft [40], [41] and malicious data access [42]. 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 [6], [7], [8], [9], [10], [11], [15], [43] have been developed and applied on ICC analysis. PCLeaks [10] uses a static taint analysis technique to detect potential component leaks that could be exploited by other components. FUSE [8] provides a multi-app information flow analysis, and an evaluation engine to detect information flows that violate specified security policies. IccTA [7] improves the precision of the ICC analysis by propagating context information between components, and claims to outperform other existing tools. Eppic [6] reduces the discovery of ICC to an instance of the Inter-procedural Distributive Environment (IDE) problem, and provides a sound static analysis technique targeted to the Android platform. COAL [11] 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 [15] combines static analysis with probabilistic models to make static inter-application analysis more tractable, even at large scale. Our developed AppHolmes, is based on the similar static analysis technique and learns lessons from these tools. However, AppHolmes is distinguished 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 ($\geq 90\%$) to give a performance boost, which is quite necessary to carry out a large-scale analysis. 2) AppHolmes avoids a lot of false positives by a moderate-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 [44], [45], [46]. Background activities are identified as a critical factor that can impact app performance in consideration of system resource consumption. Chen *et al.* [2] 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.* [1] 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 provided a system mechanism, TAMER, to mitigate the impacts caused by background workloads based on monitoring, filtering, and rate-limiting all background wakeup events. In contrast, AppConan focuses on only the apps being launched via app collusion, hence we have less side-effects on the apps' original behavior. Some third-party apps such as Greenify [36] and LBE [28] provide the capability to control and cut off the collusive behavior among apps. However, such an all-or-nothing strategy is not always practical and feasible, and users are likely absent of the knowledge and patience to make proper configuration. Google also notices the issue of background services, so they introduce two features into Android 6.0 Marshmallow, called Doze and App Standby [18]. Doze reduces the battery consumption by deferring background CPU and network activity for apps when the device is unused for a long period 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. Doze kicks in only when the device is unplugged and stationary, while App Standby needs a period of time to trigger, and can be easily broken down when there is a notification generated. Additionally, these approaches are not personalized for specific users.

Prediction of App Usage. Predicting the apps to be used can facilitate the system to prepare the apps in advance and thus improve user experiences accordingly. Abhinav *et al.* [33] proposed a text-compression-alike algorithm that regards the usage history as sequential patterns and uses the preceding usage sequence to compute the conditional probability distribution for the next app. Yan *et al.* [34] proposed an algorithm for predicting the next app to be used based on user contexts such as location and temporal information. They built an app, called FALCON, which can pop up the predicted app to home screen for fast launching. Richardo *et al.* [35] proposed the Parallel-Tree-Augmented Naive Bayesian Network (PTAN) as the prediction model, and used a large scale of data from Aviate's log dataset to train their model. Our probability prediction model learns lessons from these previous efforts, but significantly differs from them. First, the prediction model in AppConan is time-aware, i.e., we not only take into account which app will be used, but also whether they will be used within a given time window. Second, AppConan effectively balances the background apps and the system-wide resource assumption.

11 CONCLUSIONS

In this article, 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 have revealed that app collusion is quite prevalent on third-party appstores, and is very related to apps' popularity, category, version, and so on. We have explored the root causes of such collusive behaviors, and classified them into various characteristic categories. We have quantified the undesired hidden cost caused by app collusion, which are non-trivial and can significantly impose user experience. Finally, we have proposed AppConan, a time-aware, resource-consumption-wise, and personalized approach to eliminating the collusion-introduced cost by accurately predicting the app to be used while minimizing the additional resource consumption.

REFERENCES

- [1] M. Martins, J. Cappos, and R. Fonseca, "Selectively taming background android apps to improve battery lifetime," in *Proc. of ATC'15*, 2015, pp. 563–575.
- [2] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vanithamby, "Smartphone energy drain in the wild: Analysis and implications," in *Proc. of SigMetrics'15*, 2015, pp. 151–164.
- [3] "Online discussion about App Collusion." <https://www.zhihu.com/question/38732700>, 2016.
- [4] "The astonishing facts about App Collusion." <https://www.zhihu.com/question/46614588>, 2016.
- [5] "Online discussion about App Collusion." <https://www.v2ex.com/t/314390>, 2016.
- [6] 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, 2013, pp. 543–558.
- [7] 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*, 2015, pp. 280–291.
- [8] 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*, 2014, pp. 4:1–4:10.
- [9] 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*, 2014, pp. 1329–1341.
- [10] L. Li, A. Bartel, J. Klein, and Y. L. Traon, "Automatically exploiting potential component leaks in android applications," in *Proc. of TrustCom'14*, 2014, pp. 388–397.
- [11] 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*, 2015, pp. 77–88.
- [12] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "Appholmes: Detecting and characterizing app collusion among third-party android markets," in *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3–7, 2017*, 2017, pp. 143–152.
- [13] "Android aapt," http://elinux.org/Android_aapt.
- [14] "dex2jar," <https://github.com/pbx1988/dex2jar>.
- [15] 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*, 2016, pp. 469–484.
- [16] "Xposed Module Repository." <http://repo.xposed.info/module/de.robbv.android.xposed.installer>.
- [17] "Google Cloud Messaging," <https://developers.google.com/cloud-messaging>, 2016.
- [18] "Optimizing for Doze and App Standby." <https://developer.android.com/training/monitoring-device-state/doze-standby.html>.
- [19] "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>.
- [20] "Modularity." [https://en.wikipedia.org/wiki/Modularity_\(networks\)](https://en.wikipedia.org/wiki/Modularity_(networks)).
- [21] 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*, 2015, pp. 459–472.
- [22] "UI/Application Exerciser Monkey," <https://developer.android.com/studio/test/monkey.html>, 2016.
- [23] "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>.
- [24] 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*, 2012, pp. 311–320.
- [25] 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, 2011, pp. 12–16.
- [26] "Android Permission Manifest." <https://developer.android.com/guide/topics/manifest/permission-element.html>.
- [27] "Android broadcasts." https://developer.android.com/guide/components/broadcasts.html#system_broadcasts.
- [28] "LBE Security master." <https://play.google.com/store/apps/details?id=com.lbe.security>.
- [29] "Baidu mobile security." <https://play.google.com/store/apps/details?id=com.baidu.mobilesecurity.intl>.
- [30] D. M. J. Tax, "One-class classification," *Applied Sciences*, 2001.
- [31] B. Liu, Y. Dai, X. Li, W. S. Lee, and P. S. Yu, "Building text classifiers using positive and unlabeled examples," in *Proc. of ICDM 2003*, 2003, pp. 179–188.
- [32] X. Li and B. Liu, "Learning to classify texts using positive and unlabeled data," in *Proc. of IJCAI 2003*, 2003, pp. 587–594.
- [33] A. Parate, M. Hmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proc. of UbiComp 2013*, 2013, pp. 275–284.
- [34] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. of MobiSys 2012*, 2012, pp. 113–126.
- [35] R. Baeza-Yates, D. Jiang, F. Silvestri, and B. Harrison, "Predicting the next app that you are going to use," in *Proc. of WSDM 2015*, 2015, pp. 285–294.
- [36] "Greenify." <https://play.google.com/store/apps/details?id=com.oasisfeng.greenify>.

- [37] P. P. F. Chan, L. C. K. Hui, and S. Yiu, "Droidchecker: analyzing android applications for capability leak," in *Proc. of WISEC'12*, 2012, pp. 125–136.
- [38] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proc. of NDSS'12*, 2012, p. 19.
- [39] 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*, 2012, pp. 229–240.
- [40] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proc. of MobiSys'11*, 2011, pp. 239–252.
- [41] W. Enck, D. Oeteau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. of USENIX Security Symposium 2011*, 2011, p. 2.
- [42] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proc. of NDSS'13*, 2013.
- [43] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Appscopy: semantics-based detection of android malware through static analysis," in *Proc. of FSE'14*, 2014, pp. 576–587.
- [44] 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*, pp. 287–300.
- [45] V. Gabale and D. Krishnaswamy, "Mobinsight: On improving the performance of mobile apps in cellular networks," in *Proc. of WWW'15*, 2015, pp. 355–365.
- [46] 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*, 2012, pp. 41–50.



Yun Ma is a Ph.D student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include services computing and web engineering.



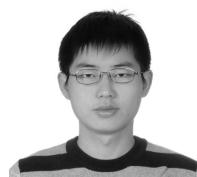
Felix Xiaozhu Lin is an assistant professor of Purdue University. He received his PhD in Computer Science at Rice (2014), M.S. in Computer Science (2008) and B.S. in Automation (2006) both from Tsinghua University, respectively. His research interests include operating system and runtime for programmability, energy efficiency, and performance.



Xuanzhe Liu is an associate professor in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His research interests are in the area of services computing, mobile computing, web-based systems, and big data analytic. He is the corresponding author of this work.



Yunxin Liu is a lead researcher in Microsoft Research Asia. He received his Ph.D. in computer science from Shanghai Jiao Tong University in 2011 (through the SJTU-MSRA joint PhD program). His current research interests include mobile systems and networking.



Mengwei Xu is a Ph.D candidate student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include mobile computing and operating system.



Ziniu Hu is an undergraduate student from Peking University majoring in computer science and technology. His research interests include data mining and software engineering.



Boyuan Yang is an undergraduate student in the School of Software Engineering (A National Pilot Software College) of Beijing University of Technology, Beijing, China. His research interests include software engineering and mobile computing.

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

Mengwei Xu¹, Yun Ma¹, Xuanzhe Liu^{1*}, Felix Xiaozhu Lin², Yunxin Liu³

¹Key Lab of High-Confidence Software Technology, MoE (Peking University). Beijing, China

²Purdue University. West Lafayette, USA, ³Microsoft 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

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

©2017 International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC BY 4.0 License. WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
<http://dx.doi.org/10.1145/3038912.3052645>



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., Baidu², Tencent³, and Wandoujia⁴. 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-

¹Note 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 ~ 2,000, and further decreases to 37 for apps ranked between top 2,001 ~ 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 than 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

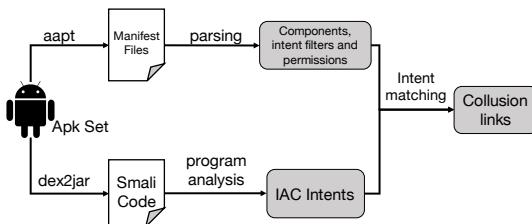


Figure 1: The workflow of AppHolmes.

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",
6             "com.example.target.otherService");
7         intent.setAction("some_action");
8         context.startService(intent);
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 $\text{Collusion}(K)$ 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:

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

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 Dalvik 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 $\langle A, B \rangle \in \text{Collusion}(K)$.

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 SDK⁵ 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

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

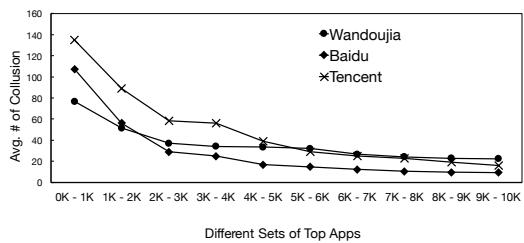


Figure 2: The average count of apps that are collusive with a given app on different Android markets. Each set includes 1,000 apps, sorted by app downloads.

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 online⁶.

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 QQ⁷ 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 col-

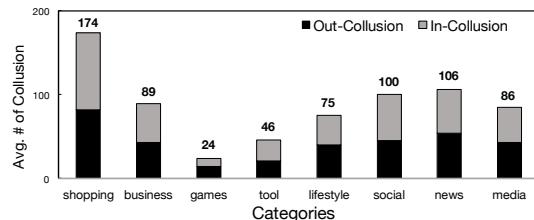


Figure 3: The average count of collusive apps for a given app from different categories (Wandoujia top 1,000).

lusion 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.

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/>

⁷Wechat 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 app⁸ 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

⁸A popular news client in China

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*

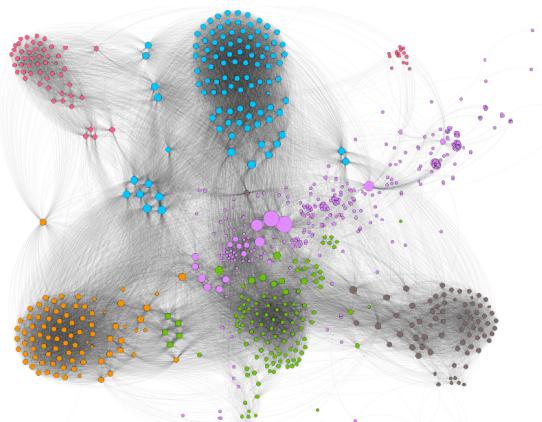


Figure 4: App communities based on detected collusion among top 1,000 apps from the Wandoujia market. Each node represents an app while each edge represents the collusive relations. Different colors represent different communities.

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 experi-

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 | Social | 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 |
| | Shugu Reader | Books | 40 M |
| | Nuomi | Shopping | 18 M |
| | 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 |
| Background apps | hao123 Navigation | Browser | 26 M |
| | Himalayan FM | News | 19 M |
| | Sina News | News | 16 M |
| | Beauty Shot | Image | 15 M |

ments are repeated for 20 times on Nexus 6 with Android 6.0. We use monkey [17] tool to automatically run over Foreground apps.

- **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 `dumppsys`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 app¹⁰.

- **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 collusion.

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

¹⁰Note 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.

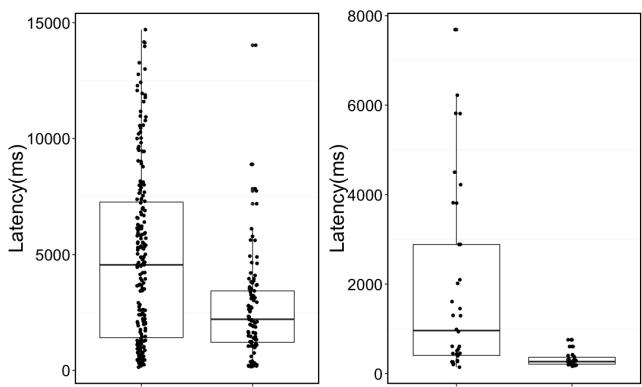


Figure 5: Launch latency of Foreground apps with and without the presence of app collusion.

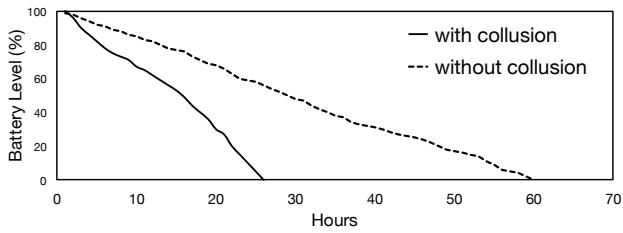


Figure 6: The impact of app collusion on battery life, estimated when the screen is always kept off.

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 re-

launch 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).

20M 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

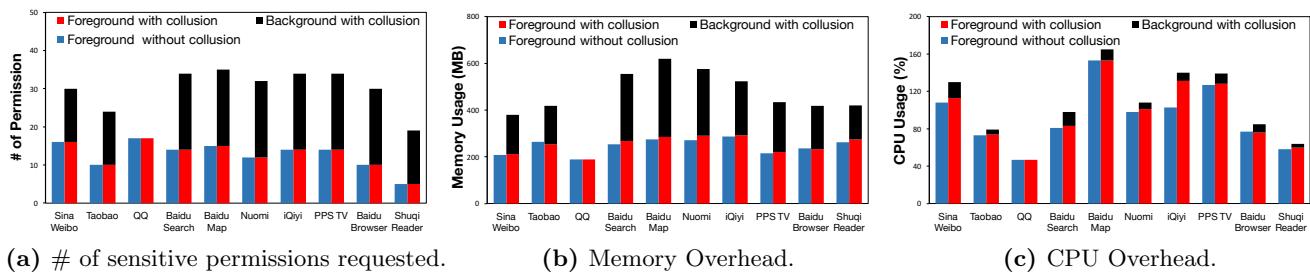


Figure 7: A breakdown of extra system resource usage, without and with the presence of app collusion.

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_\(networks\)](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 *Proc. 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. Appscopy: 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 *Proc. 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.