# DAPANDA: Detecting Aggressive Push Notifications in Android Apps

Tianming Liu[1*], Haoyu Wang[1*✉], Li Li[2], Guangdong Bai[3], Yao Guo[4], and Guoai Xu[1]
[1] Beijing University of Posts and Telecommunications, Beijing, China
[2] Monash University, Australia [3] The University of Queensland, Australia
[4] Key Laboratory of High-Confidence Software Technologies (MOE), Peking University, Beijing, China

***Abstract*—Mobile push notifications have been widely used in mobile platforms to deliver all sorts of information to app users. Although it offers great convenience for both app developers and mobile users, this feature was frequently reported to serve malicious and aggressive purposes, such as delivering annoying push notification advertisement. However, to the best of our knowledge, this problem has not been studied by our research community so far. To fill the void, this paper presents the first study to detect aggressive push notifications and further characterize them in the global mobile app ecosystem on a large scale. To this end, we first provide a taxonomy of mobile push notifications and identify the aggressive ones using a crowdsourcing-based method. Then we propose DAPANDA, a novel hybrid approach, aiming at automatically detecting aggressive push notifications in Android apps. DAPANDA leverages a guided testing approach to systematically trigger and record push notifications. By instrumenting the Android framework, DAPANDA further collects all notification-relevant runtime information to flag the aggressive ones. Our experimental results show that DAPANDA is capable of detecting different types of aggressive push notifications effectively in an automated way. By applying DAPANDA to 20,000 Android apps from different app markets, it yields over 1,000 aggressive notifications, which have been further confirmed as true positives. Our in-depth analysis further reveals that aggressive notifications are prevalent across different markets and could be manifested in all the phases in the lifecycle of push notifications. It is hence urgent for our community to take actions to detect and mitigate apps involving aggressive push notifications.**

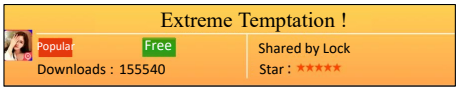
***Index Terms*—Push notification, dynamic analysis, advertisement, Android, mobile app**

## I. INTRODUCTION

Since the mobile push notification service was introduced by Apple in 2008 [17], it has been widely adopted in various mobile platforms including Android [7]. In essence, it provides a mechanism to display messages outside of the normal interface of a mobile app (usually in the status bar at the top of the screen). Push notifications are generally used by app developers to deliver various kinds of information, such as timely reminders and up-to-date messages (e.g., location-based messages and new content available in news).

Push notifications could be delivered without a specific request from the app, which means that the app does not have to be relaunched during the process. Users can directly react to the notification by simply tapping on it, providing timely information and allowing quick and easy responses for app users. The notification interfaces are further allowed to be customized by developers to provide flexibility and better user experiences. Thanks to these benefits, push notifications have been favored by both mobile users and app developers and hence been extensively integrated into modern mobile apps.


(a) An example of anonymous and ad-related APNs




(b) An example of compulsive and malicious APNs

Fig. 1. Motivating examples of aggressive and malicious push notifications.

Because notifications can effectively "*push*" an app into the user's attention, app developers are encouraged to utilize push notifications for re-engaging mobile app users. However, despite users are generally in favor of push notifications, the abuse of such notifications can still annoy app users and likely result in user complaints. For example, a number of reports revealed that app developers have been abusing push notifications for various purposes [4], [8], [11], [13], [14].

In addition to the way notifications are pushed, recent studies reveal that app users are likely to also complain about the contents delivered in the notifications [9]. For example, Bell argues that Facebook has a notification problem and enumerated 13 most annoying push notifications from Facebook [12]. Most of these notifications are considered as annoying because their contents are considered inappropriate by users. Specifically, app users especially hate notifications that contain advertisements [1], [3]. Actually, both Google [2] and Apple [16] have released strict policies to regulate the use of mobile push notifications: *Push Notifications must not be required for the app to function, and should not be used for advertising, promotions, or direct marketing purposes, or sending sensitive personal or confidential information.*

Unfortunately, despite that mobile push notifications are explicitly disallowed to deliver ads and promotions by app markets, many app developers appear to be still enticed to such practices, and even employing more sophisticated methods so as to avoid being caught during app vetting. As shown in Fig. 1

*The names of the first two authors are in alphabetical order. Haoyu Wang is the corresponding author.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00017

(a), it is an ad-related push notification, while the interface of this notification has been customized such that it does not explicitly mention which app it belongs to. The most annoying part is that it tricked the user into clicking the notification and the link will be redirected to a page full of ads. This suggests that, even though both Google and Apple have declared strict policies, it is still difficult for app markets to automatically regulate the apps that violate the policy, such that anomalous apps are still able to sneak into app markets and mobile devices. Consequently, annoying notifications are frequently pushed to mobile app users in the real-world.

More seriously, push notifications could also be exploited for malicious purposes. Fig. 1 (b) shows a push notification triggered from the app `com.keeeweee.lockscreen`. The annoying part of this notification is that it cannot be closed by the users, i.e., mobile users have to click it. Even worse, once the user clicks it, an app downloading process will be triggered immediately (specifically app *upyel.patyzbg.vbxsoef.kncp.cbsmk.rnnbs* will be downloaded) and the installation UI will pop out. When we uploaded this downloaded app to VirusTotal, over 30 antivirus engines flagged it as malicious [23]. This example suggests that push notifications could be used as a new covert channel to spread malware, which has been largely overlooked by our community.

In this work, we refer to such annoying and even malicious notifications as **aggressive push notifications (APNs for short)**. To the best of our knowledge, except for some sporadic news reports discussing specific instances of APNs, our research community has not studied this problem systematically [44], [47] and hence no research tools have been proposed to detect and mitigate the occurrences of APNs.

This paper seeks to develop an automated approach to detect APNs, and further dissect mobile push notifications and characterize their behaviors in large-scale. To that end, we first provide a taxonomy that characterizes a variety of APNs based on a comprehensive user survey and a summary of market policies. Towards the automated detection of APNs, we aim to address several key challenges:

- **How to automatically trigger push notifications efficiently?** As push notifications are displayed outside of the normal UIs of a given app, no existing tools explicitly support automated testing of push notifications (e.g., identify the notification views). Besides, APNs could be triggered in either the foreground or background. Thus, we need to develop a new approach that is scalable enough and also ensures good coverage of APNs.
- **How to accurately identify the content and network traffic from push notifications?** As push notifications are running within the hosting app, the network traffic triggered by push notifications would be mixed together with the traffic generated by other parts of the app (e.g., banner ads or the contents in the main activities). As we seek to characterize the malicious contents delivered by push notifications, it is important to pinpoint the corresponding content accurately.
- **How to trace the origins of APNs?** Mobile push notifications could be implemented by the app developers, or third-party libraries (e.g., Google Cloud Messaging). In addition to detecting APNs, we also seek to trace back to the origin of APNs, i.e., analyzing who should be responsible for the aggressive behaviors (e.g., app developers or ad networks).

To address the aforementioned challenges, we propose DAPANDA (**D**etecting **A**ggressive **P**ush Notification in **And**roid **A**pps), a novel hybrid approach that supports accurate detection of APNs. DAPANDA mainly relies on two key techniques to characterize the behaviors of mobile push notifications at runtime. To trigger push notifications efficiently, we have proposed an *app queuing* approach to enforce automated exploration of push notifications. To accurately pinpoint the information related to each notification, we have implemented an *instrumentation* method that integrates call stacks and inter-component tracing to record all necessary information to detect APNs. Finally, we use a manually crafted benchmark set to demonstrate the effectiveness of DAPANDA.

To further characterize the presence of APNs in the wild, we have applied DAPANDA to 20,000 Android apps crawled from 8 popular app markets including Google Play. We have identified over 1,329 APNs from 1,036 apps, which accounts for over 5% of the apps studied in this work. With further inspection, we have also found that a large portion of these APNs were originated from aggressive third-party libraries.

This paper makes the following main contributions:

- We have created a taxonomy of APNs in a systematic way. To the best of our knowledge, this is the first work that is focused on detecting and characterizing aggressive/malicious push notifications.
- We have implemented DAPANDA, a new approach that is able to expose push notifications with an automated exploration strategy, and then characterize the behaviors of APNs accurately.
- We have performed a large-scale measurement study by applying DAPANDA to 20,000 apps, seeking to measure the phenomenon of APNs in the wild. We have revealed the severity of APNs in the mobile app ecosystem, and further investigated the underlying working mechanisms behind APNs.

To boost research along this direction, we have released the benchmark and our experiment results at:

https://github.com/DaPANDA2019/DaPANDA

## II. A TAXONOMY OF AGGRESSIVE PUSH NOTIFICATIONS

In order to automatically identify APNs, we seek to explore why push notifications were considered as aggressive and what types of APNs exist in the mobile app ecosystem. To this end, we resort to a straightforward approach to understanding push notifications manually. This manual process allows us to form a taxonomy of all possible types of push notifications (cf. II-A). We then leverage the taxonomy to confirm APNs using a crowdsourcing-based approach, i.e., following the opinions

of Android app users responded in a survey and the policies given by app markets (cf. II-B).

### A. A Taxonomy of Android Push Notifications

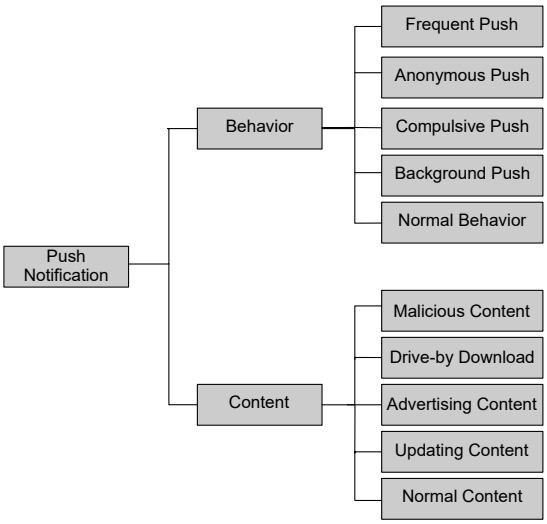



Fig. 2. A taxonomy of push notifications.

The first step is to understand the characteristics of Android push notifications and their types. We resort to various sources, including Android documentation, news reports, user comments about annoying push notifications on app markets, as well as running different Android apps ourselves. In the end, we have created a taxonomy of Android push notifications from two different aspects, as listed in Fig. 2.

The first aspect is concerned about the pushing *behavior*, based on the way notifications are pushed/displayed to mobile users. We found four specific types for this category and regard all remaining scenarios as the Normal Behavior type[1]. The specific types are explained as follows:

- **Background Push.** The notification would be triggered while the hosting app is running in the background.
- **Compulsive Push.** In general, a notification could be canceled or cleared by swiping it or clicking the `Clear button` provided by the system. However, some intrusive notifications cannot be canceled, i.e., users are forced to click it. Such kind of push notifications is caused by the misuse or malicious use of push notification configurations. Two flag fields *FLAG_ONGOING_EVENT* and *FLAG_NO_CLEAR* are related to this behavior. Aggressive/malicious developers might intentionally set the flag and force users to click these notifications.
- **Anonymous Push.** In general, push notifications should explicitly show the icons and names of their hosting apps. However, anonymous push may deliberately hide its hosting app from mobile users (cf. Fig. 1).
- **Frequent Push.** It refers to the situation where a number of notifications are pushed from the same hosting app during a short period of time (e.g., less than 2 minutes).

For the second aspect, we are concerned about the *contents* of the push notifications, including both the contents displayed and the redirected contents after clicking. We have also observed four specific types in this aspect as follows (putting the rest into the Normal Content type):

- **Advertising Content.** It refers to a notification that contains advertising content, which is explicitly disallowed by both Google and Apple. In general, it is non-trivial to detect whether the content is ad-related or not. Thus, in this paper, we regard the push notifications originated from advertising libraries as ad push, which is a reasonable assumption and we will further discuss it in Section V.
- **Updating Content.** It refers to a notification that serves as a reminder of updating or downloading app-related resources. For example, such push notifications would always remind users to update the app with contents like "new version found, need to update".
- **Drive-by Download.** This kind of notification may trigger unintentional downloads (e.g., of advertised APKs) when a user clicks the notification, without requiring user confirmation. Such behaviors often heavily impact user experience, and in most cases, drive-by downloads cannot even be easily canceled.
- **Malicious Content.** This category refers to such notifications that, after clicked, may jump to landing pages where malicious content is presented, or trigger the downloading of malicious files (e.g., apks).

With this taxonomy, we are able to classify each push notification into one or more types considering both their *behavior* and *content*. For example, a push notification that is frequent and with advertising content will be classified into the "frequent + advertising" type. Ideally, we could have as many as 25 different types (including the "normal behavior, normal content" type). However, since *frequent + updating* and *anonymous + updating* are not possible in practice, we did not take these two types into account. Finally, we have obtained 23 different notification types based on our taxonomy.

Note that while this taxonomy covers most common cases of push notifications, it is not completely orthogonal. The actual push notification may belong to more than one behavior type and more than one content type simultaneously. For example, the motivation example shown in Fig. 1 (b) belongs to *Compulsive* and *Anonymous* behavior types, and *Malicious*, *Drive-by Download* and *Ad* content types.

### B. User Survey

Although we are now able to classify mobile push notifications into different types based on their behavior and content aspects, we still do not know which types are considered to be aggressive, as no previous studies have characterized them. Instead of labeling them ourselves, we seek to adopt a crowdsourcing-based approach, i.e., assigning the level of

[1]Note that the Normal Behavior type may also contain some other types of abnormal behaviors, however, because we cannot assign them a specific type, we will consider them as Normal in this taxonomy. It is the same for the Normal Content type discussed later in this section

| | Malicious | Drive-by | Ad | Updating | Other |
|---|---|---|---|---|---|
| Frequent | 4.92 | 4.65 | 4.13 | — | 3.50 |
| Anonymous | 4.87 | 4.50 | 4.38 | — | 3.29 |
| Compulsive | 4.87 | 4.62 | 4.04 | 2.77 | 2.88 |
| Background | 4.81 | 4.52 | 3.54 | 2.62 | 2.69 |
| Other | 4.69 | 4.40 | 3.17 | 2.46 | 1.62 |

Fig. 3. The survey results. The *green* cell stands for benign and not aggressive at all, the *yellow* stands for disturbing to users but not aggressive, and the *red* ones stand for aggressive cases, and the deeper the color, the more aggressive it is from users' perspective. Overall, 17 ones (in red) are regarded as APNs.

aggressiveness of all possible types of the push notifications from the perspective of real Android app users, in order to confirm which notifications are more likely to be APNs.

**Survey Design.** Based on the taxonomy we created (cf. Fig. 2), we further embed it into a Likert-scale [54] online survey to measure the aggressiveness for each type of notifications. Participants in the survey, with the experience of using Android mobile devices and the basic understanding of mobile notifications, were provided with 23 types of push notifications together with example screenshots and their explanations. The details of the user survey could be found at the project on Github [20]. Participants are asked to grade each type of push notification from a level of 1 to 5 based on its aggressiveness, which is defined as follows:

- **1** stands for benign and not aggressive at all.
- **2** stands for disturbing to users but not aggressive.
- **3** stands for somewhat aggressive.
- **4** stands for aggressive.
- **5** stands for extremely aggressive.

To encourage users to respond to our survey, we pay 2 US dollars to the person who responds to our full survey. Eventually, our online survey receives 52 effective responses, which is a fairly representative number considering the difficulties to encourage people to answer online surveys [29].

**Survey Result.** The responding results are illustrated as a heatmap in Fig. 3. This heatmap is drawn based on the average scores by all respondents. Following the convention of Likert-scale, in this work, we define five aggressiveness levels: *Benign*, *Disturbing*, *Somewhat Aggressive*, *Aggressive*, and *Extremely Aggressive*, which are regarded as such if the average score of all the responses falls into ranges [1,2), [2,3), [3,3.5), [3.5,4.5), [4.5,5], respectively. In this work, *we consider such combinations that have an average score higher than three as APNs*. Eventually, as shown in Fig. 3, 17 combinations fall into this category and hence are regarded as APNs.

### C. Market Policies

After identifying the types of APNs, we go one step deeper to check if some of the notification types, which are considered to be aggressive by app users, have been explicitly restricted by market policies. App markets have responsibilities and incentives to regulate app behaviors that may lead to dissatisfaction of users. When using apps with APNs, users may not only complain about the app itself but also complain about the market where the app is downloaded from. To this

TABLE I
APN-RELATED POLICIES DECLARED BY APP MARKETS.

| | Behavior | | | | Content | | | |
|---|---|---|---|---|---|---|---|---|
| | Freq | Anonymous | Compulsive | BKG | Malicious | Drive-by | Ad | Update |
| GPlay | | | | | √ | | √ | |
| Huawei | | √ | | | √ | | √ | |
| Tencent | | | √ | | √ | | √ | |

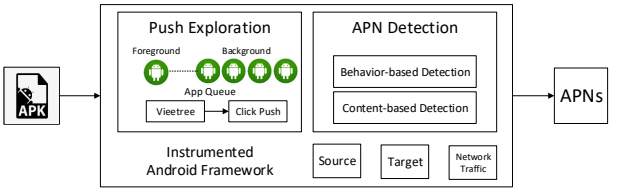



Fig. 4. Overview of DAPANDA.

end, we crawl the policy descriptions from several Android app markets (including Google Play) and manually go through them to check if the policy has explicitly mentioned that certain types of push notifications are not allowed by the apps submitted to its market. As illustrated in Table I, market policies have explicitly disallowed *malicious pushes*, *ad-related pushes*, and *anonymous/compulsive pushes*, which are generally in line with the choice of users from our survey, providing concrete evidence to confirm the correctness of our survey results. Note that the market polices are generally coarse-grained, while our survey results have extended the policies with more detailed combinations.

## III. APPROACH

Aiming at systematically detecting APNs in Android apps, we propose a dynamic analysis approach called DAPANDA for automatically exploring and characterizing push notifications. Fig. 4 illustrates the working process of DAPANDA, which is mainly made up of three modules: (1) **Automated exploration of push notifications.** This module leverages automatic Android GUI traversal techniques for triggering the appearance of push notifications and clicking subsequently the pushed notifications. (2) **Framework Instrumentation.** This module aims at hooking relevant methods for capturing runtime information of push notifications (e.g., how is a push notification triggered and consumed?). (3) **Aggressive Push Notification Detection.** This module follows the pre-defined definition of APNs to identify the aggressive ones from all the triggered notifications, utilizing the information collected in the instrumented framework after the completion of automated GUI exploration.

### A. Automated Exploration of Push Notifications

The general idea of this module is to identify and understand the layout of push notifications by constructing the corresponding view trees of each UI page and click the notifications by simulating the touch events at runtime. Since push notifications could be triggered in both the foreground and background, existing UI exploration tools focused on a single app becomes ineffective, thus we propose a new exploration strategy called

*App Queuing*, aiming at exploring as many push notifications as possible.

*1) App Queuing:* The key idea of *app queuing* is that, instead of quitting the app directly after running it in the foreground, which has been done by almost all the state-of-the-art app testing approaches [19], [52], we will put the app into the background, allowing silent notifications to be still pushed. Additionally, the *app queuing* approach can also improve exploration efficiency. In practice, although only one app runs in the foreground, more than one app could run in the background, which allows us to test multiple apps at the same time. Specifically, our exploration strategy maintains an app queue structure with three principal operations: `insert`, which adds an app to the app queue and automatically tests it in the foreground; `downgrade`, which puts the app into the background; and `remove`, which removes an app from the queue based on their arriving order (FIFO).

During exploration, we can configure the capacity of the queue $n$ and the foreground execution time $t_f$ for each app. Specifically, (1) $n$ apps are allowed to be concurrently tested in our system and (2) each app is running in the foreground for $t_f$ seconds. When experimenting on a large set of apps, we follow the FIFO principle as in the queue structure, i.e., once an app runs in the foreground for time $t_f$, we will put it into the background. Once the app queue reaches its capacity $n$, the app at the rear of the queue (who enters the queue earliest) would be removed and further be closed and uninstalled, making room for new apps. Note that, each app runs in the foreground for $t_f$ time, and in the background for $(n\text{-}1)*t_f$ time, thus $n*t_f$ time in total, while the expected average execution time for each app is still $t_f$.

To ensure each app strictly follows this strategy and prevent exceptional cases (e.g., app crash or interference of concurrent running apps), our system monitors the execution states (e.g., use adb shell command *dumpsys activity top* to query the foreground activity) at runtime for every 10 seconds. Once exceptional cases are found, our system will either revoke the app back to the foreground or restart the app (e.g., use adb shell command *am start* with the launcher activity of the app).

*2) View Tree based UI Exploration:* For each app, we split the UI exploration into two phases: *(1) exploration of in-app UIs*, to trigger push notifications; *(2) exploration of notification UIs*, to identify push notification views and click them. Thus we can not only trigger the corresponding behaviors, but also harvest their distribution contents.

**Exploration of In-app UIs.** In this work, we plan to trigger push notifications by exploring apps with randomly generated UI-focused test inputs, as the occurrence of push notification is unpredictable, without the knowledge of predictable trigger points. However, some apps present welcome pages or user agreements on their first run during the experiment, which may stop us from triggering the main app functionality. Therefore, we take advantage of a model-based UI input generation method here. During the exploration of the in-app UI, we propose to analyze the view tree of each UI state, and then apply the DFS (depth-first search) algorithm to generate the possible input events, in order to trigger the functionality of the app. Fig. 5(a) shows an example of the view tree we constructed during in-app UI exploration.

**Exploration of Notification UIs.** In order to fully exploit the app queuing mechanism, we decide to explore the notification related UIs when the testing app switches its state, from running in the foreground to running in the background. During this interval, before the next app runs in the foreground, we first simulate the *Swipe Down* action on the status bar to open the *notification drawer*, where we can view more details and take actions with the notification. We then get the view trees of the current state (notification drawer), based on Google Accessibility [15]. Fig. 5(b) shows an example of the view tree we constructed for a notification drawer. Three FrameLayout nodes are laid at the bottom, and each of them corresponds to a notification view, which is also a tree-like structure, as shown in Fig. 5(c). We can retrieve the notification related information from the view tree, including its coordinates, text messages, resource id, the package name of the original app.

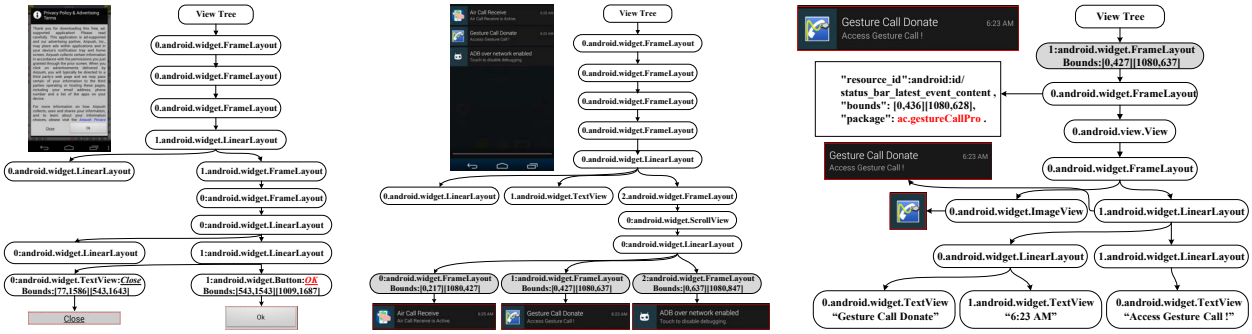
Finally, with the retrieved coordinate information, we click on the notification view accordingly by simulating a click event at the center of the view. This process would be repeated several times if we found more than one notification views. Note that we only click once for each unique push notification.

### B. Framework Instrumentation

The objective of this phase is to collect all the necessary information relevant to push notifications, mostly the ones that could be useful for characterizing APNs. Specifically, in order to accurately identify APNs, we seek to collect three types of runtime information. As shown in Fig. 6, which illustrates the typical working scenario of push notifications, the following three types of runtime information are needed: (1) The **source** where the notification is pushed to the system, (2) The **target** where the execution will jump to after the notification is clicked, and (3) the **network traffic** triggered by the consumption of the notification.

Unfortunately, it is not straightforward to collect some of the aforementioned information. For example, *it is difficult to track the source where the notification is pushed*. Furthermore, although it is relatively easy to collect all network traffic after a notification is clicked (e.g., via *Tcpdump*), *it is still difficult to locate the app that has actually generated those traffic, as there are always multiple apps running at the same time*. To this end, we propose an instrumentation-based approach, in which we leverage the Xposed framework [6] to hook all the notification-relevant methods to collect the app execution logs on demand. The Xposed framework allows us to collect the runtime information of tested Android apps without actually instrumenting the APK. We only need to set up the framework once and it works for all the apps to be tested.

Table II summarizes the key methods hooked by the instrumentation module in order to extract runtime information that our approach is interested in (i.e., the runtime information involved in the lifecycle of push notifications). Listing 1 further




(a) The view tree of a user agreement. (b) The view tree of a notification drawer. (c) The view tree of a typical push notification.

Fig. 5. Constructing the view tree of push notifications.

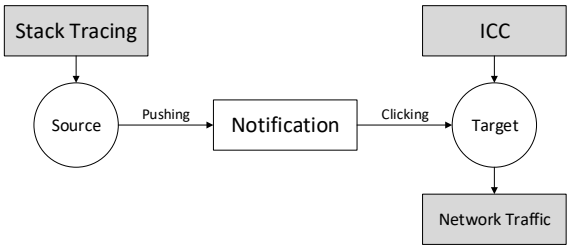



Fig. 6. The life-cycle of a standard mobile push notification and the corresponding information we collected (in gray).

TABLE II
KEY INFORMATION EXTRACTED WITH FRAMEWORK INSTRUMENTATION.

| Category | Specific Info | Method |
|---|---|---|
| Source | PackageName | API Hooking: NotificationManager.notify() /Service.startForeground() |
| | Title&Texts | |
| | Icons&Images | |
| | Flags | |
| | ResourceID | |
| | NotificationID | |
| | Source ClassName | Call stack tracing |
| Target | Intent | API Hooking: PendingIntent.getActivity() /getActivities()/getBroadcast()/getService() |
| | Target ClassName | |
| Network Traffic | ClassName of Url | API Hooking and trace call stacks in network module |
| | PackageName of Url | |
| | Drive-by Download | API Hooking: execStartActivity() |
| | PackageName&Text | Extracted from Viewtree |
| | URLs | Extracted from PCAP file |
| | Downloaded Files | |

illustrates the detailed runtime information we could collect with the instrumentation module.

*1) Notification Source:* We mainly retrieve two types of information for the notification source. The first is necessary configuration information, the other one is the origin of the push notification (e.g., the class that issues it).

During the implementation of push notifications, the developers would need to specify the detailed configurations. Some of them could be obtained during runtime from the view trees (e.g., text and icon), while some others cannot (e.g., flags). Thus, we have instrumented a list of APIs that can push notifications including *android.app.NotificationManager.notify()* and *android.app.Service.startForeground()* to get the instances of *Notification*, and then we further get the configuration data by checking the corresponding properties, as show in Table II.

```
1  //Source
2  Source ClassName: com.appquanta.dll.cookiemanager.ag
3  PackageName: cn.happyeclub.tjraduyy
4  Title&Texts:: Come and Buy!&Click to see details.
5  Icons&Images: 17301651.jpg
6  Flags: FLAG_ONGOING_EVENT
7  ResourceID of templates:
       17367140/notification_template_base
8  NotificationID: 25503184
9
10 //Target
11 Intent: {
12 flg=0x18800000
13 cmp=cn.happyeclub.tjraduyy/com.appquanta.wk.MainActivity
14   (has extras) } (from API getActivity())
15 Target ClassName: com.appquanta.wk.MainActivity
16
17 //Network Traffic
18 {Url: http://c1.apkads.com/get/w/20190411/3d991cf
19   380d8422ab581e69f8cdc0a3c.142368472.21321705.apk
20 ClassName of Url: com.appquanta.dll.cookiemanager.be
21 PackageName of Url: cn.happyeclub.tjraduyy}
22 {Url: http://alog.umeng.com/app_logs
23 ClassName of Url: com.umeng.analytics.g
24 PackageName of Url: cn.happyeclub.tjraduyy}
25 Drive-by Download
       Apk:file:///storage/emulated/0/download/
26   com.yiqimmm.apps.android-8627.zip
27
28 //Below are From PCAP
29 PackageName&Text Messages: cn.happyeclub.tjraduyy, Come
       and Buy!&Click to see details.
30 URL:c1.apkads.com/get/w/20190411/3d991cf380
31   d8422ab581e69f8cdc0a3c.142368472.21321705.apk
32 alog.umeng.com/app_logs
33 Downloaded Files:3d991cf380
34   d8422ab581e69f8cdc0a3c.142368472.21321705.apk
```

Listing 1. An example of obtained runtime information.

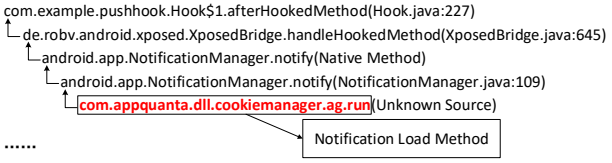



Fig. 7. An example of stack traces.

To trace the origin of push notifications, we apply a call stack based method. From the *Notification* instance we located via instrumentation, we log its call stack, and further pinpoint the package and class issuing this notification. Fig. 7 shows an example of a call stack we harvested at runtime. In the example, the notification is implemented by an ad library called

Quanta [10], where we can trace to its load method from the call stack. Note that, to further assist our source tracing, our system also integrates a third-party static library detection tool, which could help us flag the possible ad libraries and push libraries, even if in the form of naming obfuscation (e.g., com.a.b.c). In this work, during our implementation, we have embedded LibRadar++ [57], [70], and based on which we have labeled over 60 ad libraries and 20 push libraries.

*2) Notification Target:* The actual target of each push notification, i.e., the component it connects to, is essential for us to identify the actual traffic triggered by the corresponding push notification. The target component is set via a *PendingIntent*, a special *Intent* that can take an action in the future. After the notification is clicked, the pending *Intent* will be sent to the system. Following the inter-component communication mechanism [46], the appropriate target component will be activated to execute in the foreground. Thus, our goal is to demystify the corresponding *Intent*.

In Android, there are two forms of intents: *Explicit Intents* and *Implicit Intents*. Explicit Intents specify the target component (i.e., via the *cmp* attribute), which can be directly inferred. However, for implicit intents, the value of *cmp* is not directly set but via several special attributes such as *action*, *category*, etc. These attributes will be leveraged by the system to locate the target components, which should have declared an *Intent Filter* with the same attribute values.

By instrumenting a series of APIs *getActivity()/getActivities()/getBroadcast()/getService()* under *android.app.PendingIntent*, we can acquire the value of *Intent* concerning three types of components in Android – Activity, Service and Broadcast Receiver. We check the *cmp* attribute value to directly locate target components for explicit Intents (e.g., line 2 in Listing 2). For implicit Intents, we resort to the Intent and Intent Filter matching mechanism to pinpoint the target component. Normally, the Intent Filter contents can be extracted from the manifest configuration file of Android apps (e.g., lines 6-13 in Listing 2). However, this is not always true for Broadcast Receivers, in which dynamic Intent Filters can be registered without mentioning in the manifest file. To this end, we additionally hook method *android.app.ContextImpl.registerReceiverInternal*, the underlying implementation of API *registerReceiver()*, to further include dynamically registered Intent Filters (e.g., lines 16-20 in Listing 2).

*3) Network Traffic:* For each push notification clicked, a PCAP file is generated using Tcpdump [21] to record its network traffic. We also gather the package name of the source app that pushed the notification via the View Tree. This package name will be used to check whether the notification is triggered by the app running in the foreground. We then analyze the PCAP files using Bro [22], where several scripts are further introduced to extract contents from the traffic.

To further pinpoint the notification traffic, we have instrumented a list of network APIs in "HttpClient", "HttpUrlConnection" and "OkHttp", which are widely used networking modules. By hooking and tracing the call stacks of these key APIs, we

```
1  //Explict Intent
2  Intent{flg=0x24000000 cmp=be.ppareit.swiftp_free
3      /be.ppareit.swiftp.gui.FsPreferenceActivity}
4
5  //Implict Intent targetting statically-registered
       component
6  Intent={act=com.zhiyoo.UPDATE_CLICK (has extras)}
7  AndroidManifest.xml (Registration Info):
8          <receiver
       android:name="com.zhiyoo.app.BBSReceiver">
9              <intent-filter>
10                 <action
       android:name="com.zhiyoo.UPDATE_CLICK"/>
11                 ......
12             </intent-filter>
13         </receiver>
14
15 //Implict Intent targeting dynamically-registered
       Broadcast Receiver
16 Intent{act=com.unipay.secservice.action.SYNC (has
       extras)}
17 Instrument API
       android.app.ContextImpl.registerReceiverInternal :
18 IntentFilter.mActions: com.unipay.secservice.action.SYNC
19 BroadcastReceiver:com.unipay.xiaowo.pluginmgr.plugin1
20                     .MyBroadcastReceiver$1
```

Listing 2. Three types of intents and their target components.

can acquire the URLs in the network traffic, and the origin package that triggered the URLs (cf. Line 18-24 in Listing 1). The extracted information will help identify network traffic introduced by the corresponding push notification. For example, as shown in Listing 1, the traffic of Umeng Analytics does not belong to the push notifications, as its origin package (cf. Line 23) does not equal to the package of notification target (cf. Line 15). Note that, as drive-by-download notifications would trigger app downloading first, and then pop up an installation activity (interface provided by the system), we further instrument the API *execStartActivity()* to capture this behavior.

### C. Aggressive Push Notification Detection

As demonstrated in Fig. 2, push notifications are categorized from two aspects: (1) runtime behavior and (2) notification content. With the runtime information collected, we are now able to detect APNs based on our taxonomy.

**Behavior-based detection.** It is quite straightforward to characterize a given push notification to the specific types of notifications based on the behaviors specified in Fig. 2. For example, we regard a notification as a *background push* if it is not pushed by the foreground app, *compulsory push* if *FLAG_ONGOING_EVENT* or *FLAG_NO_CLEAR* flags are enabled, and *frequent push* if three or more notifications with different notification id are pushed from the same app within two minutes. Regarding the *anonymous push*, since Android 7.0, the system forces notifications implemented with *system templates* to display its source app name [5]. In order to keep pushing anonymous notifications, app developers are required to implement customized templates. We are able to extract all possible system templates (seven kinds in total, based on the ResourceID). Therefore, we regard a notification as an *anonymous push* if system templates are not used while neither the title nor icon is matched between the notification and the tested apps.

**Content-based detection.** Content-based notification types are also quite easy to classify, once the relevant runtime

information is collected. We can identify *updating contents* based on keyword matching (e.g., download, update and new version, a total of 9 keywords), while *drive-by download* based on if an APK file is downloaded after each notification is clicked, since no other interaction will be introduced. For *malicious content*, VirusTotal will be leveraged to scan every URL and files recognized from the network traffic (collected after the notification is clicked). We consider a notification as containing malicious content as long as VirusTotal flags its content as such. For *Advertising content*, we will check if the source method, which pushed the notification, belongs to ad libraries.

Finally, the remaining push notifications that cannot be classified into the above types will be considered as Normal Behavior/Content types.

## IV. Evaluation

To evaluate the performance of DAPANDA, we consider answering the following three main research questions.

- **RQ1:** Can DAPANDA effectively and accurately identify APNs in Android apps?
- **RQ2:** What is the percentage of apps with APNs in the wild? What is the distribution across different app markets?
- **RQ3:** How are the underlying working mechanisms of APNs manifested in the lifecycle of push notifications?

### A. Experimental Setup

To effectively answer the above research questions, we will conduct both in-the-lab and in-the-wild experiments. The in-the-lab experiment aims to provide reliable indications on the performance of our approach and, at the same time, identify appropriate parameters for setting up the in-the-wild experiment, which subsequently is applied to evaluate the performance of our approach for a large number of apps in real-world settings.

**Setup for RQ1 (in-the-lab).** To evaluate the effectiveness of our tool, we need to build a benchmark to support in-the-lab experiments. Unfortunately, to the best of our knowledge, there are no publicly available benchmarks on mobile push notifications in our community. Therefore, we resort to user comments on app markets (Google Play in particular) to manually construct such a benchmark. If a given app receives at least two comments complaining about the annoying behaviors of its notifications, the app has a high probability to push aggressive notifications and hence is a good candidate to be included in our benchmark. We first use a keyword-based (e.g., push notification, notification bar) method to filter relevant user comments, and then we manually went over the reviews and randomly selected 100 such apps to form our benchmark set. Note that during our selection of benchmark apps, we cannot figure out the type of APNs accurately, as some user comments are vague and hard to infer their corresponding behaviors.

As for the parameters in the app queuing exploration strategy, we further set the capacity "$n$" of our app queue to four different scales, from 1 to 7. For $n = 1$, representing that our system also supports running only one app each time, the app would be explored fully in the foreground state. We set the maximum capacity as 7 in our experiment, as the testing phone we used (Nexus 5) is unable to host more apps running at the same time due to its hardware constraints. We set the exploration time per app "$t$" to 11 different scales, from 5 seconds to 1,800 seconds. Note that the upper-bound was set dynamically during our experiment, based on whether we could trigger more APNs.

**Setup for RQ2 and RQ3 (in-the-wild).** For RQ2 and RQ3, we rely on real-world Android apps to support the in-the-wild experiments. From August 2017 to December 2018, we had crawled and collected over 3 million Android apps from 8 markets including Google Play. To perform an efficient study, we seek to focus on those apps that are likely to invoke APIs related to notifications delivery (e.g., *notify()*). To this end, we have incorporated our system into a static analyzer to identify the invocations of those APIs in the apps. By doing so, we have managed to obtain 20,000 apps (without considering the markets at this point) as our dataset.

In the large-scale experiments, we launch DAPANDA on actual smartphone devices, i.e., Nexus 5 smartphones with Android 4.4 (or API level 19). We do not use emulators since apps may embed evasion techniques to avoid running on emulator environments [65]. We use four Nexus 5 smartphones running in parallel for testing. It takes roughly 42 hours to explore all 20K apps, with the app queue size $n = 5$ such that 5 apps were running at the same time, and the exploration time $t = 600s$, where each app would be running in the foreground for 120s and in the background for 480s.[2]

### B. RQ1: Effectiveness of DAPANDA

Table III shows the overall result of our evaluation on the crafted benchmark under different configurations. In general, our approach could achieve a high recall rate (84 APN-triggering apps at most out of 100 labeled apps). We manually confirmed and categorized those apps into our taxonomy, as shown in Table IV. To further explore the reasons why our exploration cannot recall all labeled apps, we conducted a manual analysis. We installed and ran the remaining apps for a long time, and we found that the notifications could not be triggered even manually. There could be multiple explanations on this. First, the apps were released years ago, and the notification services could be invalid, or we did not get the appropriate app version as users complained. Second, it may require the right combinations and configurations in order for the APNs to appear. Finally, it is also possible that some user comments might not be accurate at all.

From our experiment result shown in Table III, we also identified the appropriate parameters for the large-scale study.

(1) In general, the number of APN-triggering apps is positively correlated with the exploration time. However, the number reaches its peak at $t = 600s$ or $t = 900s$ in most cases, and increasing the exploration time further would not significantly improve the results.

(2) With the exploration time growing, it is interesting to see that, strategies with app queuing ($n > 1$) achieve better

[2]This configuration is selected because it achieves the best performance in the study of RQ1.

TABLE III
THE NUMBER OF APN-TRIGGERING APPS WITH DIFFERENT PARAMETER SETTINGS FOR THE 100 APPS IN OUR BENCHMARK.

| Time/Strategy | App Queue | | | |
|---|---|---|---|---|
| | $n = 1$ (foreground) | $n = 3$ | $n = 5$ | $n = 7$ |
| 5s | 37 | - | - | - |
| 30s | 48 | - | - | - |
| 60s | 54 | - | - | - |
| 180s | 63 | 60 | 55 | 36 |
| 300s | 71 | 69 | 67 | 49 |
| 450s | 74 | 77 | 78 | 58 |
| 600s | 76 | 82 | 84 | 69 |
| 900s | 76 | 84 | 84 | 75 |
| 1200s | 76 | 84 | 84 | 74 |
| 1500s | 76 | 84 | 84 | 78 |
| 1800s | 76 | 84 | 84 | 76 |

TABLE IV
THE DISTRIBUTION OF DIFFERENT TYPES OF APNS IN OUR BENCHMARK. WE ONLY SHOW THE 84 ONES THAT WERE TRIGGERED WITH APNS AND FURTHER CONFIRMED BY US.

| Behavior\Content | Malicious | Drive-by | Ad | Updating | Other | Total |
|---|---|---|---|---|---|---|
| Frequent | **10** | **10** | **12** | - | **2** | 16 |
| Anonymous | **5** | **2** | **9** | - | **22** | 37 |
| Compulsive | **31** | **28** | **28** | - | - | 55 |
| Background | **16** | **12** | **24** | - | - | 33 |
| Other | **3** | **5** | **5** | - | - | 5 |
| Total | 41 | 36 | 53 | - | 24 | 84 |

results than the strategy with fully foreground exploration ($n = 1$). The underlying reason is that a number of APNs were triggered when the apps were running in the background, which is the advantage of our app queuing strategy. Note that with very limited time ($t < 60$, cf. Table III), we did not perform exploration based on app queuing. This is because with multiple apps running simultaneously, the app install/uninstall process may take longer than the foreground running time, which may cause conflicting issues.

(3) It is interesting to observe that, the strategy with "$n = 5$" is slightly better than "$n = 3$", and both of them achieve better results than "$n = 7$". We seek to investigate the reasons and found that with "$n = 5$", more background cases could be triggered. However, with "$n = 7$", due to the hardware limitations of Nexus 5, the smartphone would be lagging and some apps cannot work properly.

As a result, the best configurations for the following large-scale study are: the app queue size $n = 5$ and the exploration time $t = 600s$.

**Findings #1:** DAPANDA *is able to effectively and accurately detect APNs in our manually crafted benchmark set. Among 100 Google Play apps received complaints about their annoying notification behavior,* DAPANDA *can automatically flag 84 of them (a recall of 84%).*

### C. RQ2: The distribution of APNs in the wild

We then show results of RQ2, to understand how many apps with APNs exist in the wild. For the selected 20,000 market apps, we have successfully triggered 2,446 unique push notifications from 2,052 apps, which accounts for over 10% of the apps in our dataset. The distribution among different notification types is shown in Table V. Note that, although all the apps selected in our dataset have been found incorporating the related APIs, not all of them were identified with push notifications during our experiment, mainly due to two reasons. On one hand, the push notification related APIs would never be executed by the app, and checking statically whether an API is reached is an instance of the (undecidable) halting problem [37]. On the other hand, for *non-aggressive* push notifications, most of them were implemented based on third-party services (e.g., Google Cloud Messaging), and fully controlled by the developers (e.g., pushing messages at a certain time of the day), with strict regulations by the service providers (e.g., Google GCM regulates that developers cannot push repetitive push notifications in a single day [18]). For the identified 2,446 push notifications, 1,329 of them (54%) are considered to be APNs, based on the results of our user survey (cf. Section II-A). The 1,329 APNs were found to be pushed from 1036 apps, taking up 5.18% of our dataset.

TABLE V
THE OVERALL RESULT.

| Behavior\Content | Malicious | Drive-by | Ad | Updating | Other | Total |
|---|---|---|---|---|---|---|
| Frequent | **65(232)** | **58(210)** | **63(226)** | - | **17(62)** | 93(331) |
| Anonymous | **64(71)** | **56(63)** | **34(37)** | - | **205(229)** | 328(359) |
| Compulsive | **196(362)** | **141(294)** | **135(288)** | 132(144) | 653(675) | 978(1180) |
| Background | **324(397)** | **187(247)** | **325(404)** | 95(116) | 212(233) | 675(805) |
| Other | **120(152)** | **148(186)** | **73(87)** | 245(284) | 155(164) | 490(553) |
| Total | 608(839) | 471(694) | 509(717) | 432(512) | 963(1023) | 2052(2446) |

**Distribution across Markets.** Table VI shows the distribution of our dataset and identified apps with APNs across market[3]. Over 1.98% to 7.52% of app candidates in the studied markets were flagged as apps with aggressive notifications. Although each market has declared strict developer policies to regulate the APNs, we still find a number of aggressive cases in these markets. **This result suggests that it is challenging to perform automated regulation of APNs, thus both the app markets and our research community should pay more attention to this issue.**

**Findings #2:** *APNs are prevalent across all the app markets we studied, i.e., covering over 5% of the apps in our dataset. It is urgent for app markets to adopt techniques like* DAPANDA *to identify and remove apps with aggressive notification behaviors.*

### D. RQ3: Understanding the lifecycle of APNs

We further characterize the push notifications triggered in the large-scale experiment from different phases in their lifecycle, including (1) the origin of the push notification (including its implementation template), (2) the reflected runtime behaviors, (3) the triggered contents, and (4) the corresponding actions after the notifications are consumed.

[3]Note that one APK may correspond to several markets, as different markets have overlapped apps.

TABLE VI
THE DISTRIBUTION OF OUR DATASET AND IDENTIFIED AGGRESSIVE APPS.

| Market | #App | # Aggressive | % Aggressive |
|---|---|---|---|
| Google Play | 1,265 | 25 | 1.98% |
| HUAWEI Market | 1,499 | 104 | 6.94% |
| Tencent Myapp | 5,332 | 186 | 3.49% |
| PP Helper | 7,834 | 276 | 3.52% |
| Wandoujia | 5,967 | 241 | 4.04% |
| HiMarket | 3,795 | 240 | 6.32% |
| OPPO Mraket | 3,503 | 109 | 3.11% |
| Anzhi Market | 2,820 | 212 | 7.52% |
| Total | 20,000 | 1036 | 5.18% |

TABLE VII
DISTRIBUTION OF PUSH NOTIFICATIONS FROM DIFFERENT LIBRARIES.

| Ad Library | | | Push Library | | |
|---|---|---|---|---|---|
| Library Name | # Push | # App | Library Name | # Push | # App |
| Airpush | 173 | 170 | Tencent Bugly | 25 | 18 |
| Moxiu | 136 | 133 | Jpush | 25 | 7 |
| Daoyoudao | 121 | 72 | Umeng Message Push | 6 | 3 |
| JYPush | 67 | 28 | Tencent Tpush (Xinge) | 4 | 2 |
| Wostore_UNIPAY | 47 | 20 | GCM/FCM | 4 | 4 |
| PandaAd | 44 | 18 | Baidu Push Services | 4 | 3 |
| Feiwo | 31 | 15 | Rongyun Push | 2 | 1 |
| Kuguo | 23 | 16 | | | |
| Migu SDK | 15 | 13 | | | |
| Mipush | 13 | 12 | | | |
| Other | 47 | 12 | | | |
| **Total** | **717** | **509** | **Total** | **70** | **38** |

*1) Origin of the Push Notifications.:* Based on "ResourceID of templates" obtained from Framework Instrumentation, we observed that most push notifications were implemented using system templates, while over 26% of the apps (536 in numbers) and over 24% of the push notifications (584 in numbers) triggered were implemented using customized templates. In these 584 notifications, over 85% were labeled as APNs (498 in numbers). We further analyze the origin of the triggered push notifications, taking advantage of the call-stack based approach we proposed in Section III-B. Over 32% of the notifications were triggered by third-party libraries, including ad libraries and push libraries. We listed the ad libraries and push libraries with the number of triggered notifications in Table VII. While notifications originated from ad libraries were all considered to be APNs, taking up 30% of all notifications and 54% of APNs, some popular push notification services, including Google GCM/FCM and Baidu Push Services, were only identified with a few cases in our experiment, and no sensitive ones. As we mentioned earlier, these notification services have strict regulations to keep away APNs. For example, Google GCM/FCM does not allow developers to use customized push notifications, which prevents anonymous pushes completely.

*2) Runtime Behaviors:* We then provide a detailed characterization of their runtime behaviors based on the taxonomy we summarized in Section II-A.

**Frequent Push.** We have identified 93 apps with frequent push notifications, i.e., pushing 3+ messages in less than 2 minutes. For the 331 push notifications, over 232 of them were considered to be malicious, leading users to malicious URLs or downloading malware. Over 210 of them were also drive-by download pushes, and 226 of them push ads frequently.

**Anonymous Push.** For the 584 push notifications that use customized templates, 359 of them were considered to be anonymous, i.e., hiding app name and app icon in any Android versions. Additionally, 492 push notifications could also be considered as anonymous in Android versions prior to V7.0, as they do not provide such information, but they implement the notifications based on system templates. The systems will force them to show app names in Android 7.0 and afterwards.

TABLE VIII
THE TOP 5 DOMAINS THAT HOST THE MOST NUMBER OF MALICIOUS URLS.

| Domain | # Number | # Aggressive | Aggressive% |
|---|---|---|---|
| api.airpush.com | 193 | 162 | 83.94% |
| mobile.eagla.com | 133 | 121 | 90.98% |
| ff.td68x.com | 64 | 64 | 100% |
| img.qycdn.daoyoudao.com | 39 | 39 | 100% |
| ei.nd.enjoyfinance.cn | 35 | 35 | 100% |

TABLE IX
THE TOP 5 DOWNLOADED MALICIOUS APPS.

| MD5 | # VT | Source app |
|---|---|---|
| b0490a5d8cce59616a12705adc546b61 | 40 | com.androidemu.harveshihun.alvinshihun |
| f93ec3d8490d583f425b0b5f312cb809 | 39 | com.budwbo |
| 7d8d182bf06d500217abca147ede9be1 | 36 | com.RunnerGames.game.Jesgtingche |
| fd23f172bb3633453cf154e769884dfe | 35 | com.june.sixteen.juejizhuti |
| 4a1417007cce3309e04b28f326953288 | 35 | com.suishouxie.yemdssfhgfekeji |

**Compulsive Push.** Over 48% of the push notifications we identified belong to the compulsive notification category. The most aggressive cases were that, 362 of them deliver malicious contents in this way, i.e., users cannot close the notifications and have to visit malicious URLs or download malware.

**Background Push.** Over 33% of the push notifications were triggered when the apps run in the background, which demonstrates the effectiveness of our app queuing strategy. Over half of the background notifications were malicious and advertisement related.

*3) Triggered Contents:* The APNs usually pose threats and spread sensitive contents including malicious contents. Then, we further analyzed the triggered malicious contents.

**URLs/Domains.** As we have recorded all traffic triggered by clicking push notifications, we are able to harvest 5584 distinct URLs, belonging to 997 different domains. We further analyzed the malicious URLs, i.e., the malicious or phishing pages introduced by clicking the push notifications. As reported by VirusTotal, 1,034 URLs from 194 domains were flagged as malicious. Table VIII shows the top 5 domains that host the most number of malicious URLs we identified.

**Drive-by-Download Apps.** During our exploration, we have collected 1,004 drive-by-download apps triggered by 471 apps with aggressive notifications. For the 1,004 apps we harvested, only 252 were unique, i.e., some apps were downloaded several times. We further send these apps to VirusTotal. The result suggested that 174 of them (69%) were flagged as positive, and 75 of them were flagged by 10 anti-virus engines, while 44 apps were flagged by 20 anti-virus engines. Table IX shows the top 5 downloaded malicious apps flagged by the most number of anti-virus engines.

*4) The Targets of Push Notifications:* We further categorized target components of push notifications. Over 42% of them invoke components within the app, while over 25% of them invoke third-party components that belong to ad libraries or

push libraries. Besides, over 15% of them invoke system components (e.g., *android.intent.action.VIEW*) to perform actions including opening files and visiting URLs.

**Findings #3:** *APNs could be manifested in all the phases in the lifecycle of push notifications, ranging from the origin of the notifications to their triggering behaviors, and from the contents to their target components, once the contents are consumed.*

## V. Threats to validity

To the best of our knowledge, this work is the first attempt in the community towards detecting APNs in Android. The implementation of DAPANDA, however, carries several limitations.

First, although we have created an effective app automation tool to trigger push notifications, the timing of push notification messages usually depends on the developers/advertisers. Our empirical study on the labeled benchmark suggested that most aggressive push notification behaviors could be triggered within 10 minutes of app running (cf. Section IV-B), however, malicious developers could use sophisticated ways to bypass our detection. Indeed, the classic principle of the unwinnable arms race between the attackers and defenders also applies to our work. There is hence a need to continuously improve our approach towards inventing advanced techniques for detecting aggressive push notifications in the long run. Second, we consider a push notification as advertisement-related by tracing whether it is originated from known ad libraries, as it is non-trivial for us to identify ads from the contents. However, there may exist exceptional cases where the notifications are pushed by app code to perform some in-app promotions. So far, in this case, we will still regard them as non-advertisement push notifications. Third, the contents (landing URLs or the downloaded APKs) in push notifications may vary due to factors such as time, location and user identifiers, etc, which unfortunately are ignored at the moment.

## VI. Related Work

**Mobile Push Notifications.** This paper is the first to detect aggressive push notifications for Android apps. Nevertheless, there are several studies [24], [31], [33], [45], [51], [53], [58]–[60], [72], [73] focused on analyzing mobile push notifications from other aspects. For example, Chen *et al.* [31] studied the security qualities of emerging push-messaging services and developed a tool to evaluate the security qualities of the service's SDKs and its integration within different apps. Ahmadi *et al.* [24] characterized the usage of Google Cloud Messaging (GCM) in Android malware, and proposed to trace the flows of GCM to improve malware detection. Lee *et al.* [45] have explored a new C&C channel for mobile botnets based on the push notification service of Android. These studies may have a correlation with part of our work, however, our work is the first to identify and characterize aggressive behaviors in mobile push notifications.

**Mobile Advertising.** Mobile advertising has been widely studied, including different techniques to detect third-party libraries (including ad libraries) [27], [50], [57], [66]–[68], analyzing the security and privacy behaviors of mobile ad libraries [30], [38], [49], [55], [63], mobile ad fraud detection [32], [34], [35], [56], and malicious contents distributed [61], [62]. Mobile push notification, which could also be used as a means for delivering mobile advertisement, has not been well studied in our research community. Nevertheless, some related techniques could be applied in our study.

**Malicious and Gray Behaviors of Mobile Apps.** Android malware detection is a more general research direction, with a large number of techniques and measurement studies [26], [28], [36], [37], [43], [48], [69]–[71], [74] proposed. Besides, a number of studies were focused on analyzing gray behaviors and aggressive/annoying behaviors in mobile apps. Andow *et al.* [25] proposed to design and implement heuristics for seven main categories of grayware, and then use these heuristics to simulate grayware triage on a large set of Android apps. Tang *et al.* [64] proposed a systematic and comprehensive empirical study on a large-scale set of fake apps. Hatada *et al.* [40] analyzed "potentially unwanted applications" (PUAs) in Android and proposed to classify them based on the similarity of DNS queries. A number studies were focused on fraudulent behaviors in mobile apps, e.g., promotion attack [39], [75], fake review [41] and new kinds of scams [42]. As APNs cover both malicious behaviors (e.g., spreading malware) and gray behaviors (e.g., compulsive or anonymous), our work is a complementary study of these existing efforts.

## VII. Conclusion

In this paper, we present the first work to demystify mobile push notifications and detect aggressive push notifications (APNs) automatically. In particular, we first create a comprehensive taxonomy, and then propose DAPANDA, a hybrid approach that leverages UI automation and framework instrumentation techniques to identify APNs. We have applied DAPANDA to 20K Android apps crawled from 8 app markets. Our experimental results show that APNs indeed exist in many Android apps. Among these aggressive notifications, some of them were found to be maliciously used to distribute malware and create annoying messages. Our results encourage our research community to invest more efforts into the detection and mitigation of APNs.

## Acknowledgment

We sincerely thank our shepherd Prof. Amin Alipour (University of Houston), and all the anonymous reviewers for their valuable suggestions and comments to improve this paper. This work is supported by the National Key Research and Development Program of China (grant No.2018YFB0803603), and the National Natural Science Foundation of China (grants No.61702045 and No.61772042).

## REFERENCES


[1] I, as a developer, HATE the idea of AirPush. Lets make a list of apps that use it, 2011. https://www.reddit.com/r/Android/comments/gzdz6/i_as_a_developer_hate_the_idea_of_airpush_lets/.

[2] Google Play Developer Programme Policies, 2013. https://play.google.com/intl/en-gb/about/index.html.

[3] No More Notification Ads and Icon Ads in Android Apps, 2013. https://googlesystem.blogspot.com/2013/08/no-more-notification-ads-and-icon-ads.html#gsc.tab=0.

[4] How to Disable Notifications from Any App in Android, 2015. https://www.makeuseof.com/tag/stop-annoying-notifications-android/.

[5] Notifications in Android N, 2016. https://android-developers.googleblog.com/2016/06/notifications-in-android-n.html.

[6] Xposed Framework API, 2016. https://api.xposed.info/reference/packages.html.

[7] Android Cloud to Device Messaging, 2017. https://en.wikipedia.org/wiki/Android_Cloud_to_Device_Messaging.

[8] Find out which app is pushing ads in my notification bar?, 2017. https://android.stackexchange.com/questions/41045/find-out-which-app-is-pushing-ads-in-my-notification-bar.

[9] No More Notification Ads and Icon Ads in Android Apps, 2017. https://www.reddit.com/r/Android/comments/545x7k/theres_too_many_popular_apps_that_are_abusing/.

[10] Quanta Sky Inc., 2017. http://www.appquanta.com/index.html.

[11] Suspicious Push Notification on Android Phone, 2017. https://security.stackexchange.com/questions/186653/suspicious-push-notification-on-android-phone.

[12] The 13 most annoying Facebook notifications, ranked, 2017. https://mashable.com/2017/06/16/worst-facebook-notifications-ranked/.

[13] 'Ghost Push' Malware Threatens Android Users, 2017. https://www.pandasecurity.com/mediacenter/mobile-security/ghost-push-malware-android/.

[14] How to block spam notifications and rogue ads on Android smartphones, 2018. https://www.androidpolice.com/2018/05/16/track-block-rogue-ads-android/.

[15] Accessibility overview, 2019. https://developer.android.com/guide/topics/ui/accessibility.

[16] App Store Review Guidelines, 2019. https://developer.apple.com/app-store/review/guidelines/.

[17] Apple Push Notification service, 2019. https://en.wikipedia.org/wiki/Apple_Push_Notification_service.

[18] Firebase console, 2019. https://console.firebase.google.com/.

[19] Monkey, 2019. https://developer.android.com/studio/test/monkey.

[20] Survey results on Github, 2019. https://github.com/DaPANDA2019/DaPANDA.

[21] Tcpdump, 2019. http://www.tcpdump.org.

[22] The Zeek Network Security Monitor, 2019. https://www.bro.org/.

[23] Virustotal Detection Result, 2019. https://www.virustotal.com//#/file/56469bccccb788176564a03451e8879d5c8b70c6c65294fa254ea7cbe852cf90/detection.

[24] Mansour Ahmadi, Battista Biggio, Steven Arzt, Davide Ariu, and Giorgio Giacinto. Detecting misuse of google cloud messaging in android badware. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 103–112. ACM, 2016.

[25] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. A study of grayware on google play. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 224–233. IEEE, 2016.

[26] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE, 2015.

[27] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[28] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. Towards model checking android applications. *IEEE Trans. Software Eng.*, 44(6):595–612, 2018.

[29] Yehuda Baruch and Brooks C Holtom. Survey response rate levels and trends in organizational research. *Human relations*, 61(8):1139–1160, 2008.

[30] Theodore Book, Adam Pridgen, and Dan S Wallach. Longitudinal analysis of android ad library permissions. *arXiv preprint arXiv:1303.0857*, 2013.

[31] Yangyi Chen, Tongxin Li, XiaoFeng Wang, Kai Chen, and Xinhui Han. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 1260–1272. ACM, 2015.

[32] Jonathan Crussell, Ryan Stevens, and Hao Chen. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 123–134. ACM, 2014.

[33] Junhua Ding, Wei Song, and Dongmei Zhang. An approach for modeling and analyzing mobile push notification services. In *2014 IEEE International Conference on Services Computing*, pages 725–732. IEEE, 2014.

[34] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–268. ACM, 2018.

[35] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Guoai Xu, and Shaodong Zhang. How do mobile apps violate the behavioral policy of advertisement libraries? In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications (HotMobile '18)*, pages 75–80. ACM, 2018.

[36] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[37] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[38] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[39] Qian Guo, Haoyu Wang, Chenwei Zhang, Yao Guo, and Guoai Xu. Appnet: understanding app recommendation in google play. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics (WAMA '19)*, pages 19–25. ACM, 2019.

[40] Mitsuhiro Hatada and Tatsuya Mori. Detecting and classifying android puas by similarity of dns queries. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 590–595. IEEE, 2017.

[41] Yangyu Hu, Haoyu Wang, Li Li, Yao Guo, Guoai Xu, and Ren He. Want to earn a few extra bucks? a first look at money-making apps. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, pages 332–343. IEEE, 2019.

[42] Yangyu Hu, Haoyu Wang, Yajin Zhou, Yao Guo, Li Li, Bingxuan Luo, and Fangren Xu. Dating with scambots: Understanding the ecosystem of fraudulent dating applications. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[43] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[44] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2018.

[45] Hayoung Lee, Taeho Kang, Sangho Lee, Jong Kim, and Yoonho Kim. Punobot: Mobile botnet using push notification service in android. In *International workshop on information security applications*, pages 124–137. Springer, 2013.

[46] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering*, pages 280–291. IEEE, 2015.

[47] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[48] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

[49] Li Li, Timothée Riom, Tegawendé F Bissyandé, Haoyu Wang, Jacques Klein, and Yves Le Traon. Revisiting the impact of common libraries for android-related investigations. *Journal of Systems and Software*, 154:157–175, 2019.

[50] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE, 2017.

[51] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 978–989. ACM, 2014.

[52] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C 2017)*, pages 23–26. IEEE, 2017.

[53] Yuanchun Li, Ziyue Yang, Yao Guo, Xiangqun Chen, Yuvraj Agarwal, and Jason I Hong. Automated extraction of personal knowledge from smartphone push notifications. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 733–742. IEEE, 2018.

[54] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[55] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, pages 89–103. ACM, 2015.

[56] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. {DECAF}: Detecting and characterizing ad fraud in mobile apps. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 57–70, 2014.

[57] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th international conference on software engineering companion*, pages 653–656. ACM, 2016.

[58] Zhaotai Pan, Xiaoxing Liang, Yu Chen Zhou, Yi Ge, and Guo Tao Zhao. Intelligent push notification for converged mobile computing and internet of things. In *2015 IEEE International Conference on Web Services*, pages 655–662. IEEE, 2015.

[59] Martin Pielot, Karen Church, and Rodrigo De Oliveira. An in-situ study of mobile phone notifications. In *Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services*, pages 233–242. ACM, 2014.

[60] Martin Pielot, Amalia Vradi, and Souneil Park. Dismissed!: a detailed exploration of how mobile phone users handle push notifications. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, page 3. ACM, 2018.

[61] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS*, 2016.

[62] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *NDSS*, 2016.

[63] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, volume 10. Citeseer, 2012.

[64] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. A large-scale empirical study on industrial fake apps. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 183–192. IEEE, 2019.

[65] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[66] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 515–516. IEEE, 2017.

[67] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82. ACM, 2015.

[68] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Automated detection and classification of third-party libraries in large scale android apps. *Journal of Software*, 28(6):1373–1388, 2017.

[69] Haoyu Wang, Hao Li, and Yao Guo. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play. In *Proceedings of the 2019 World Wide Web Conference (WWW '19)*, pages 1988–1999. ACM, 2019.

[70] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the 2018 Internet Measurement Conference (IMC '18)*, pages 293–307. ACM, 2018.

[71] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. Rmvdroid: towards a reliable android malware dataset with app metadata. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, pages 404–408. IEEE, 2019.

[72] Ian Warren, Andrew Meads, Satish Srirama, Thiranjith Weerasinghe, and Carlos Paniagua. Push notification mechanisms for pervasive smartphone applications. *IEEE Pervasive Computing*, 13(2):61–71, 2014.

[73] Zhi Xu and Sencun Zhu. Abusing notification services on smartphones for phishing and spamming. In *WOOT*, pages 1–11, 2012.

[74] Wei Yang, Mukul Prasad, and Tao Xie. Enmobile: Entity-based characterization and analysis of mobile malware. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 384–394. IEEE, 2018.

[75] Hengshu Zhu, Hui Xiong, Yong Ge, and Enhong Chen. Discovery of ranking fraud for mobile apps. *IEEE Transactions on knowledge and data engineering*, 27(1):74–87, 2014.