Devils in Your Apps: Vulnerabilities and User Privacy Exposure in Mobile Notification Systems

Jiadong Lou*, Xiaohan Zhang†, Yihe Zhang*, Xinghua Li†, Xu Yuan*, and Ning Zhang‡
*University of Louisiana at Lafayette, Lafayette, Louisiana, USA
†Xidian University, Xi'an, China
‡Washington University in St. Louis, St. Louis, Missouri, USA

***Abstract*—Witnessing the blooming adoption of push notifications on mobile devices, this new message delivery paradigm has become pervasive in diverse applications. Accompanying with its broad adoption, the potential security risks and privacy exposure issues raise public concerns regarding its great social impacts. This paper conducts the first attempt to exploit the mobile notification ecosystem. By dissecting its structural elements and implementation process, a comprehensive vulnerability analysis is conducted towards the complete flow of mobile notification from platform enrollment to messaging. Meanwhile, for privacy exposure, we first examine the implementation of privacy policy compliance by proposing a three-level inspection approach to guide our analysis. Then, our top-down methods from documentation analysis, application network traffic study, to static analysis expose the illicit data collection behaviors in released applications. In addition, we uncover the potential privacy inference resulted from the notification monitoring. To support our analysis, we conduct empirical studies on 12 most popular notification platforms and perform static analysis over 30,000+ applications. We discover: 1) six platforms either provide ambiguous KEY naming rules or offer vulnerable messaging APIs; 2) privacy policy compliance implementations are either stagnated at the documentation stages (8 of 12 platforms) or never implemented in apps, resulting in billions of users suffering from privacy exposure; and 3) some apps can stealthily monitor notification messages delivering to other apps, potentially incurring user privacy inference risks. Our study raises the urgent demand for better regulations of mobile notification deployment.**

***Index Terms*—mobile notification, vulnerability analysis, privacy exposure.**

I. Introduction

Mobile notification pushing pervasively exists, enabling app providers to send advertisements or other messages of interest to users. By delivering messages on devices' screens, the notification progressively becomes an effective way to quickly and deliberately propagate information to target users. A survey from *Bussiness of Apps* [41] has reported that over 50 billion mobile notifications have been sent to 900 million users during H1 2018, and each US user received 46 push notifications on average per day in 2019. Given such a mega-scale market, any misuse of the system by an adversary, either to spread misinformation and disinformation or to massively collect private user data without explicit permission from the users, is concerning. To ease the development of personalized mobile notifications, more than 50% (out of 30,000+ analyzed so far) of the apps are leveraging the mobile notification service in their interactions with customers. We analyze open-sourced notification services and discover that the mobile notification fulfills the message delivery and data collection by calling the third-party libraries (i.e., Android SDK) while messaging APIs are provided by the separate platforms to push the notifications. With such handfuls of platform services built into massive amounts of apps, it is of critical importance to understand the potential security and privacy issues.

For correspondence, please contact Prof. Xu Yuan (xu.yuan@louisiana.edu).

So far, the security of mobile applications has received significant attention in the past several years and various techniques were developed to analyze security properties [20], [15], [55], [32], [16]. In addition, the detection of vulnerabilities and privacy leakage toward third-party libraries has also attracted research community's attentions [36], [35], [21], [38], [40], [43], [57], [45], [46], [19], [24], [11], [29], [47]. Regarding mobile messaging services, many studies [34], [8], [39], [56], [31] have been conducted to detect the misuse/vulnerabilities of old Google Firebase Cloud Messaging system. Furthermore, a detection tool, *i.e.,* Seminal, was designed [17] to extract the semantic information of source codes to evaluate the integration of notification SDKs. However, a systematic security/privacy analysis toward the emerging mobile notifications ecosystem remains unexplored yet.

In this paper, we conduct the first systematic security and privacy exploration of the Android mobile notification services provided by popular emerging platforms. Through empirical study, we dissect the system designs of the mobile notification ecosystem into four stages, *i.e.,* platform enrollment and key distribution, notification SDK configuration, device tracking, and notification pushing through APIs. According to these stages, we identify four critical processes, *i.e.*, notification KEY configuration, application authentication, messaging API, and message verification. Then, following these processes, we analyze the potential vulnerabilities and viable attacks to exhibit real-world threats.

We further analyze the privacy exposure issues of mobile notification services. Our explorations are carried out from two aspects, *i.e.*, the data collection behavior in the notification SDK and the user privacy inference through notification monitoring. Regarding the former one, which ubiquitously exists without users' awareness, we first analyze the privacy policies and their compliance. In particular, we divide the privacy policy implementation into three levels and provide the related APIs inspection methods, including calling time

checking and user grant parameter tracking, to examine the compliance. Then, to reveal the data collection in apps, we start from the documentation analysis for identifying the data collection APIs, and then conduct a testing app network traffic study for exploiting different types of user data collection. After that, we conduct the taint analysis to extract sensitive data flow in released apps while proposing two new sensitive data sources *i.e.*, user in-app event and Android geofencing event. Beyond the code-level inspections on notification SDKs, we also discover a new side-channel attack for privacy inference through notification monitoring, which is blamed on the sharing access to the notification bar of the Android system. That is, a malicious app with notification listening permission can stealthily monitor mobile notification messages delivered to other apps, so as to infer the user's private information.

We conduct large-scale empirical studies on the 12 most popular notification platforms and collect 30,000+ apps in markets. NotiLeak, an automatic analytical tool is also developed to conduct the static analysis among these apps. Our results indicate that three platforms provide ambiguous guidance of KEY names, which indeed results in mistaken KEY storage in 174 released apps. Besides, *Pushbot* platform is detected to provide the vulnerable messaging API, leading to potential risks of malicious notification tampering. *Umeng* and *Mobpush* platforms with billions of app installations apply the insecure HTTP protocol and adopt weak MD5 for verification, suffering from insider attacks and their vulnerabilities are demonstrated by our case study. For privacy concerns, we expose the shocking fact that privacy policy compliance implementations are either stagnated at the documentation stage (8 of 12 platforms) or never implemented in apps (less than 4% apps). Moreover, 6705 apps (more than 41%), including some influential applications with more than millions of installations, have stealthily collected users' data, such as location, user in-app behaviors, among others. Finally, our inspection results exhibit that mobile notification monitoring behaviors exist in 245 apps and a case study is provided to validate that an app can stealthily monitor all mobile notifications delivered to other apps. Thus, these detected apps can potentially be used for inferring users' other private information. All these results unveil the severe security and privacy issues accompanying the blooming adoption of mobile notification services and they should raise wider and closer attention to help improve and regulate notification services. The contributions and significance of this work are summarized as follows:

- We are the first to dissect the structural elements and implementation process of the emerging mobile notification ecosystem, which pave the way for future research along this direction. We conduct the comprehensive analysis to explore their potential security and privacy issues.
- We disclosed three vulnerabilities and potential attack schemes related to their protocol designs and implementations in the notification enrollment and delivery stages.
- We proposed a three-level inspection method to examine the privacy policy and their compliance, and a top-down approach to reveal data collection behaviors in apps, for comprehensively exposing the potential privacy issues involved in mobile notifications apps.
- A large-scale empirical study over 12 notification platforms and 30,000+ applications in the market is conducted. The security issues on these platforms and the privacy exposure behaviors on involved apps are reported, revealing the potential risks to massive users.

According to our comprehensive and systematic exploration, we have the following novel and critical findings.

- First, we reveal the neglected fact that the mobile notification pushing is implemented by the third-party platforms rather than the app developers themselves, which results in serious security and privacy concerns.
- Second, users are prone to have the wrong perception that their data are used by apps but neglect the privacy exposure that the notification platform can collect the users' data. Unfortunately, the practical privacy policy compliance of these notification platforms is quite subpar, and users are rarely informed of and aware of such data collection practices.
- Third, regarding the most concerned location information exposure, we discovered a new side channel, i.e., geofencing-triggered notification, which can be adopted to track a user's location even if the location permission is not granted to apps.
- Fourth, the Android system's notification permission mechanism has an unexpected flaw: different apps will share the notification bar, enabling a malicious app to acquire the listening permission and masquerade as a normal app to secretly monitor other apps' notification messages. Such a privacy breach can lead to the inference of users' sensitive data.

The remainder of this paper is organized as follows. In Section II, we dissect the mobile notification ecosystem to reveal the flow of its messaging and configuration protocols. In Section III, we conduct the security analysis of notification protocol design and implementation. Section IV illustrates the privacy exposure of mobile notification and the privacy inference issues for notification monitoring. In Section V, we conduct the large-scale empirical studies to investigate the existing notification platforms and released apps. Section VI outlines the related work, Section VII discuss the limitations, and Section VIII concludes this paper.

II. Mobile Push Notification Ecosystem

Intuitively, users may misconceive that the received notifications are directly sent by application servers, however, notifications in fact come from third-party agents who serve for providing the notification delivery services. Such agents are called the third-party *Notification Platforms* in this paper, which provides the notification SDKs to be carried into applications for implementing the functionality of creating channels between a user device and a notification platform, enabling the platform to distribute notification messages to

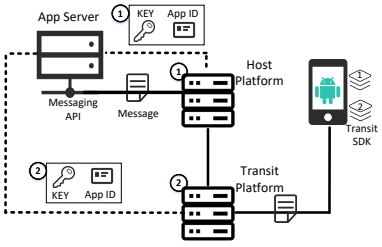

Fig. 1. The dual-platform structure.

target devices. This section dissects the system design of mobile notification systems for gaining a clear understanding of their ecosystem. After investigating the 12 most popular mobile notification platforms, the common structure of this ecosystem is to be introduced, its main design components are to be illustrated, and its delivery flows for notification pushing are to be presented, which will serve as the basis to analyze the security vulnerabilities and privacy issues, to be exposed in Sections III and IV, respectively.

A. System Structure

Through extensive empirical studies on the 12 notification platforms, we have identified the typical structure of notification platforms. We call it the dual-platform structure, which includes both a host notification platform and a transit notification platform, tightly integrated as shown in Fig. 1. The former one is responsible for processing the notification tasks from different app servers and transporting the notification to the latter one. The latter one is responsible for notification delivery to the user devices, which is a system-level notification provider for providing a stable communication channel. The advantage of such a decoupling design is to help lift the notification delivery rate. Note that, the system-level notification platforms can design the transit notification platform by themselves (*e.g.,* FCM from Google).

B. Mobile Notification Mechanism

Through analyzing user documents and application demos from various notification platforms, and personally carrying out the app development, we conclude the procedure of developing an app with mobile notification function into four stages: 1) enrollment and key distribution; 2) SDK configuration; 3) device identification; and 4) notification pushing through messaging API. Details of four stages are elaborated below.

1) Platform Enrollment and KEY Distribution: The enrollment process starts from the platform enrollment. Developing mobile apps with the mobile notification service shall take into account two important issues: 1) the platform needs to know who makes the notification API calls and 2) who is the destined application for notification messages that are sent to the messaging API. Hence, two important parameters are offered by notification platforms, *i.e.,* the messaging API key and the AppId, in regard to the two issues. The former one is used for the authentication purpose, which is uniformly called as KEY in the rest of this paper. When an application server calls the messaging API for sending a notification, this KEY will be used for examining its identity. The AppId serves as the app identification and should be hard-coded in application source codes, to enable notification messages to be delivered to the corresponding target application on a device.

In practice, the enrollment process includes two steps. First, the app developer registers his applications on the transit notification platform to obtain a pair of KEY and AppId. Second, the developer registers the app on the host notification platform and fills in the KEY and AppId derived from the transit platform. After validation, a new pair of KEY and AppId will be generated and assigned to the developer at the account console. Then, the new KEY and AppId pair will be adopted by the app server to send the notification task and the host platform will use the original KEY and AppId to let the transit platform deliver the message.

2) Notification SDK Configuration: Next, the developer imports the notification SDKs to the developed app. Meanwhile, AppId is appended in the AndroidManifest.xml file or in the notification initialization function. For example, in the application with OneSignal SDK, we set the AppId in the initial function named as: "OneSignal.setAppId(ONESIGNAL_APP_ID)". Both host and transit notification SDKs are imported accompanying their AppIds.

3) Device Tracking: AppId can only identify the application but unable to pinpoint the target user device. Hence, two methods are suggested by notification platforms. The first one is to track a device by the DeviceId, which is generated by the SDK to mark the device installing the application. Developers need to call the DeviceId generator at the initialization function of the main activity. When the app is running, the DeviceId will be generated and transmitted to both the notification platform and the application server. The second method comes from the user tag. That is, devices can be marked with different tags (e.g., male/female, user interest, etc.), set up by developers. Developers can call the tag generator at any code position while defining a tag value. When the tag generator's code section is triggered by a user's action, the tag will be generated and transmitted to the notification platform and application server. Notably, these DeviceId and user tags are generated for supporting normal messaging behaviors. So, the behaviors of uploading their parameters are not considered as the sensitive /privacy data exposure.

4) Notification Pushing through APIs: With the three aforementioned steps, an app with the notifications function is developed. Then, the application server can adopt the notification messaging API provided by the platform to send messages or collect user data. Here, we only present the normal procedure of notification pushing but leave the analysis of stealthy data collection behaviors and privacy issues in Section IV. According to our empirical study towards popular notification platforms, the messaging APIs are all in the form of RESTful API, where an application server only needs to deliver the authentication information, *i.e.,* KEY and AppId, as well as message payloads to the specified URL address. The notification message is in the JSON format, including the target devices information indicated by the DeviceId, user

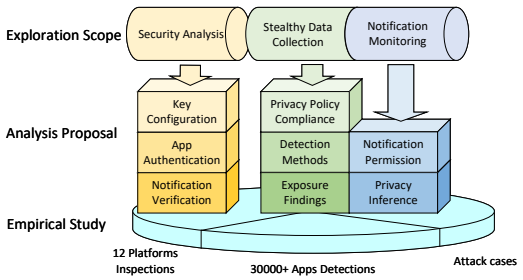

Fig. 2. The overview of our security and privacy area explorations.

tags group, and other control options. Taking OneSignal platform as an example, the notification message payload should be pushed to "https://oneSignal.com/api/v1/notifications", then the platform will make use of the channel established upon the SDK carried in the device for delivering notifications.

In practice, the messages will be first sent via the messaging API to the host platform. After re-organizing these messages to meet a certain format, the host platform calls the messaging API for sending them to the transit platform, who will then deliver messages to the user device, as shown in Fig. 1.

C. Sketch of Our Analysis

After dissecting the mobile notification ecosystem, we will conduct a comprehensive analysis for potential security and privacy issues in Sections III and IV, respectively. Since the transit platforms in the dual-platform structure are the system-level platform, such as FCM from Google, which is considered as a trustful service provider, its security and privacy issues are not in the scope of this paper. We only focus on the host platforms in the dual-platform structure. As shown in Fig. 2, our analysis can be summarized as follows. For the security analysis, we target three key procedures, *i.e.*, KEY configuration, application authentication, and notification verification, while providing the corresponding attack schemes. For privacy exposure, stealthy data collection behaviors are firstly explored from the privacy policy compliance analysis, data collection detection, to our findings of privacy exposure. Then, we discuss the potential privacy inference problem through notification monitoring.

III. Security Analysis

The notification platforms possess strong delivery abilities and can quickly spread messages to a tremendous amount of mobile devices, incurring potential risks by providing channels for malicious entities to publicly propagate messages to the society. Any potential security issue may lead to the severe social impacts and consequences, if an attacker (*e.g.,* terrorist or rumor spreader) hijacks the notification channels. To this end, this section conducts the security analysis to illustrate the vulnerability concealed in such mobile notification services.

A. Security Analysis of KEY Configuration

Our security analysis starts from the KEY configuration, which is used for authentication purpose in the messaging API. Ideally, this KEY should be well-stored at the server end only. But, per our empirical studies, we capture the clue with respect to the mistakes frequently made by app developers in this step. Notably, in Section II-B1, we have differentiated two important parameters as KEY and AppId. However, some platforms just define both parameters as the KEY with different prefixes. For example, in Kumulos, AppId is defined as "API_KEY" while the KEY is defined as "SECRET_KEY", which can easily cause confusion in the practical development, resulting in both of them being hard-coded into apps.

Detection Methods. To detect the misconfiguration of KEY, we can conduct the inspection towards KEY and AppId naming rules in the developer guidance. Also, we will examine the KEY misuse in the released apps. We extract the potential strings that match the format and length of KEY or AppId in the SDK configuration position of app reverse-engineered source codes. To avoid ethical issues and impacts to apps' notification services, we design a special verification method to check whether the extracted strings are KEY or not. Specifically, we observe that the notification platform offers the messaging API to deliver notification to the individual device by providing the DeviceId, and returns different error codes corresponding to device unfound or incorrect KEY. Hence, we call these messaging APIs with the extracted strings as KEY and the null string as the DeviceId. As such, we never deliver notification messages to users but still can receive error codes. By checking them, we can verify whether the extracted string is KEY. Our results will be presented in Section V-C1.

Potential Attack. An attacker can steal the KEY wrongly stored in the application code, and acts as an application server to push his malicious messages to the victim's devices. He can massively crawl the released apps from the application store to search for the potential KEY. These KEYs can be found in the specified field, such as the "AndroidManifest.xml" file and notification initial call functions, while the KEYs strings are of the fixed format. Thus, they can be easily identified by an attacker. Once the KEY is located, the attacker can call the corresponding notification API and broadcast malicious messages to all user devices installing the app.

B. Security Analysis of Application Authentication

As we have discussed in Section II-B3, the notification platform relies on the AppId to identify the application, so an application server can send his KEY and the corresponding AppId to the messaging API for requesting to send notification messages. However, if the notification platform lacks the additional authentication mechanism to validate the application server's identity, any attacker once obtaining the key can send messages to the target app. That is, an attacker can hijack application channels to act as an application server to deliver his notification messages to more user devices.

Detection Methods. To conduct the application authentication examination, we register two applications on the same notification platform and obtain the KEY and AppId for each application. When calling the messaging API with the KEY for the first app and the AppId for the second app, we examine whether the second app can receive the notification message.

If yes, we can claim that this notification platform lacks the secure application authentication.

Attack Scheme. The AppId has to be hard-coded in the released application to construct the network channel. Hence, an attacker can reverse-engineer the application files to search for the AppId. Besides, the attacker can also get the employed notification platform by identifying the keywords of notification SDK carried in the app. After that, the attacker can register an account on this platform and create his application on the corresponding platform to gain a legitimate identity for the KEY. When calling the notification messaging APIs, the attacker can append his KEY and the AppId obtained in the victim application and then deliver his message to the users of victim application by hijacking its channel.

C. Security Analysis of Message Generation and Verification

We next analyze the message generation and verification, aiming to raise the serious concern for mobile notification protocol design. Based on our observation, some notification platforms have realized the security risk of HTTP-based API and then designed MD5 signature-based methods in the message generation process, to promote the authentication and message examination. However, MD5 has been demonstrated to be a weak method in [22], [54], [30], [14], regarding which the chosen-prefix collision attacks were proposed in [51], [50], [53], [52], enabling an attacker to change the prefix of input but still generate the same MD5 output.

Realizing the security issues that lurk in the message generation and verification process with the MD5 signature, we propose an insider attack. Notably, in the insider threat model, an internal attacker cannot directly send the malicious notifications with API KEY since the application servers can discover and block them. That is, he cannot access the KEY but is only responsible for the design of and distributing the notification contents. However, he can still bypass the verification mechanism to distribute the malicious message. We expose such an insider threat because the mobile notification possesses wide social impacts, if succeeded, an attacker can quickly broadcast malicious messages to the great public.

Before presenting our attack, we will first illustrate the process of MD5-based message authentication in some mobile notification systems. In which, the application server is required to calculate an MD5 signature for the notification payloads in the HTTP request combined with his KEY such as: $MD5\{Payloads||KEY\}$, where $||$ represents the string concatenation. When calling the messaging API, this signature is appended in the URL as follows: *http://msg.xxxx.com/api/send?signature*. After receiving the network packet, the notification platform will calculate the signature in the same format. Notably, based on the AppId in the received payloads, the platform can recognize the application and choose the corresponding KEY. If the two signatures collide, this notification message passes the authentication.

Detection Methods. We can inspect the message content verification mechanism, especially the MD5-based examination, adopted by these messaging APIs.

Attack Scheme. Here, an insider attacker aims to maliciously distribute notification without being detected by the notification platform. She can first generate two pieces of notification messages, one is a normal message (*e.g.,* a general weather forecast) and the other is a malicious content (*e.g.,* a designed rumor). The approach proposed in [52] can be employed to create the MD5 collision by generating two different suffixes corresponding to the two notification messages. To eliminate the JSON parse mistake, we can place the generated data blocks into the optional field. Furthermore, for MD5, if *String1* and *String2* collide, then appending the same string before or after *String1* and *String2* would also collide. Hence, the two different notification messages can result in the same MD5 signature when appending the KEY. Note that, the notification payloads are delivered through JSON format and some characters, such as " {} []", are keywords, so the generated suffixes that contain these characters can cause the parsing problem. In the practical experiment, we will calculate multiple collision cases to avoid this situation. The insider attacker will send normal notification messages to the application server for MD5-based signature authentication. After that, he substitutes the payload with the prepared malicious rumor before delivering the signed packet to messaging API. This malicious message can pass the authentication at the notification platform for delivering. A case study to demonstrate the feasibility of such an attacker is exploited in Section V-C4.

IV. Privacy Exposure Analysis

We next turn our attention to the privacy leakage issues resulting from mobile notification services. Per our empirical study, we discover an unexpected circumstance: users' private data are stealthily collected and displayed on the account console of the notification platform. Such a discovery motivates us to examine user data exposure issues from notification SDKs mounted in apps. Besides, we also find that an app can access the messages shown on the notification bar sent from other apps. Such shared access to the notification bar on the Android system may incur an unexpected privacy inference risk. As such, this section focuses on two aspects, *i.e.,* the data collection behaviors of notification SDKs and the privacy inference from mobile notification monitoring.

A. Stealthy Data Collection

The first privacy issue comes from the user data collection, whereas users' private or sensitive data are uploaded to notification platforms rather than to the application server without users' awareness. We detail our threat model and then expose the privacy issues regarding data collection behaviors.

1) Threat Model for Mobile Notification: We first clarify our user data exposure threat model, which is defined as: *User and device's data accessed by or created in the host applications are uploaded through notification SDK to the notification platform, without users' agreement or awareness.*

The notification SDK mounted in an app is invisible to users when the host app is installed on the device and applied for required permissions. The permission control mechanism

will treat it as a part of the app to inherit its permission. Once a user installs such an app and grants permissions to it, he misunderstands that his data will be utilized by the app server rather than being uploaded to the third-party notification platform. As such, we treat such sensitive data uploading to the notification platforms as privacy exposure and the data transmission to the app server as normal behavior.

2) Privacy Policy Analysis: We have realized the condition that the privacy policies of notification SDKs may state that notification servers will collect the data, so we first conduct the privacy policy compliance analysis to explore whether the app users are correctly informed and know that their data is collected by notification servers. Three questions guide our analysis: 1) whether mobile notification platforms provide the privacy policy to the public claiming their data collection behaviors? 2) whether they offer mechanisms for checking the user agreement? and 3) whether the user is informed?

Different countries/regions have proposed strict regulations to guide data collection behaviors [3], [5], [6]. Hence, mobile notification providers have noticed the potential privacy dispute derived from the data collection and provided privacy policies for compliance. These privacy policies should be displayed to users and seek for granting permission when an app is installed or operated for the first time. However, such important processes are usually missing. To uncover the privacy policy issues, we design three levels of detection guidelines for various practical compliance implementations.

1 *Privacy Policy-only.* Some notification platforms provide a privacy policy containing announcements of data collection behaviors and asking for developers to display them to users for granting agreement. However, no enforcement is put to developers and no mechanism is provided to verify users' acknowledgments. As a result, such a privacy policy is more like a disclaimer, failing to bring users' particular attention to the privacy exposure risks.

2 *User Confirmation Required.* Beyond providing the privacy policy, some platforms require feedback from the app when a user confirms to understand the data collection behavior. For example, Mobpush platform asks the developer to add a confirmation function "submitPolicyGrantResult(boolean isGranted, com.mob.OperationCallback callback)" before initializing notification services. Only if the "isGranted" parameter is true, the SDK can provide the notification-related service and enable the data collection. However, this can be forged by a developer by setting the respective parameter to be true regardless of users' behaviors.

3 *Privacy Policy Display.* Some notification platforms also require the carrying app to display the privacy policy to users for authorization. For example, the Umeng Platform demands the developer to add a function "UMConfigure.preInit()" before initializing the notification service, to let the privacy policy display on apps and require users' agreement. However, the use of these functions is not enforced, resulting in the service still being provided even if such a function is not called in the code.

Pessimistically, if a user accepts the privacy policy and terms of use in default without carefully checking them, his private data can be stealthily collected. With the three aforementioned guidelines, we present our detection solutions. **Phase 1: Documents Analysis.** We first conduct the documentation analysis on the privacy policies provided by notification platforms and classify them based on the aforementioned three aspects, *i.e.,* data collection claims, user confirmation callback APIs, and policy display APIs. Considering only 12 most popular notification platforms, this documentation analysis is conducted manually by searching the API keyword matching. **Phase 2: Source Code Static Analysis.** Regarding the platforms that provide APIs for privacy compliance, we design a 3-step detection approach to conduct the static analysis on released apps. We first check the existence of privacy policy APIs and then compare the calling time of these APIs and notification initial functions. Finally, for the user confirmation callback APIs, we track the uploaded parameter by checking if it is hard-coded with a true value.

- *Step 1 – Policy APIs Searching.* We search the corresponding policy-related APIs, such as "UMConfigure.preInit()", in source codes to filter out the applications that do not inject the demand APIs.
  Step 2 – Calling Time Checking. According to the development guidance, these APIs should be called before initializing notification services, so we next locate the initial function of notification services. After that, we generate the calling graph of apps and conduct topological sorting for comparing the calling order of policy APIs and notification services APIs. If the latter one is in front of the former one, the app will be reported.
- *Step 3 – Confirmation Parameter Tracking.* For the user confirmation callback APIs, we use taint analysis to track the uploaded parameters. For example, when checking the API of "submitPolicyGrantResult(boolean isGranted)", we track the definition and assignment flow of "isGranted", which should be assigned based on the user input. If the confirmation parameter is hard-coded as the true value, the app will be reported.

Notably, our efforts undertaken on the privacy policy analysis and the compliance implementation detection go beyond the previous studies on third-party SDKs exploration, which only target the common and sensitive system-level APIs. Our inspection of privacy policy-related APIs, including the calling time and important parameter tracking, can promote the third-party SDK analysis to a more practical level by taking into account real-world applications.

3) Discovering Data Collection Behaviors: While Section IV-A2 focuses on the privacy policy compliance, this section will further explore if the stealthy data collection behaviors indeed exist in notification SDKs. Our analysis includes three steps, depicted as follows. Notably, we contribute two new sensitive data sources, *i.e.,* the developer-defined user in-app action and geofencing event, in the taint analysis.

Phase 1: Documentation Analysis. We manually conduct the analysis in this step. We focus on the notification SDK APIs uploading user data, such as location, network connection parameters, in-app actions, among others. These APIs will be further leveraged to guide the taint analysis design.

Phase 2: Application Network Traffic Study. Next, we develop the testing apps for network traffic analysis. We follow the developer documents to integrate the notification SDKs into our testing apps and initialize all functions. When installing them on our smartphones, we capture the network packets during the app installation and running. If necessary, we also decrypt the TLS-protected context with a security certificate. Then, we examine packet payloads to check whether our privacy data are carried. The privacy data we found in the captured network packet will serve as the evidence for guiding the static code analysis in the next phase. According to our empirical study, the notification platform displays all data that it collects from the user's device on the user console. We log into the account and switch to the console, for checking the device installation state and examining if user's data are indeed uploaded to the notification platform.

Phase 3: Taint Analysis among Released Apps. Finally, we conduct the taint analysis to examine if mobile notification SDKs will upload sensitive data from the user device to the notification platforms. Our key idea is to identify data flows that originate from sensitive sources (*e.g.,* location calls) and end up in the suspect sinks (*e.g.,* notification SDK uploading APIs). Note that, only the data flow pointing to the notification SDK uploading APIs will be considered as the sensitive data flow. Once the sensitive data flow appears, private data uploading through the notification SDK is identified. We develop a detection framework based on the FlowDroid [10] to characterize apps' behaviors, but make the customized design to suit the mobile notification-specific source and sink. In particular, two new sources (*i.e.,* the user in-app event definitions and the Android geofencing events) are proposed, which have never been considered in existing third-party libraries analyses.

Step 1 – Sensitive Source Configuration. Based on the Android APIs presented in SUSI [44], we select the Android system call that covers the majority of sensitive data including locations, sensors data, *etc*. Besides, for the user in-app actions, their sources should be the definition of the functions. We locate this type of source according to the event definition function in notification SDK, such as Airship SDK event definition function: "ActionRunRequest.createRequest().setValue(actionValue).run();" Considering that the location information can be uploaded through the geofencing event, we add this event call as the source, *e.g.,* "geofencingEvent = GeofencingEvent.fromIntent(intent)". The details of user in-app actions and genfencing functionalities will be presented in Section IV-A4.

Step 2 – Uploading Sink Configuration. To improve the efficiency and avoid false positive, we only target two types of sensitive sinks, *i.e.,* notification SDK data collection APIs and Android network transmission functions. In addition, we also consider the scenario that once the mobile notification service is initialized in the host app, its SDK will automatically trigger data uploading functions. For example, the app moving from the foreground to the backend can trigger app usage time tracking. So, we also mark all network transmission calls in the notification SDK, including HTTP and TCP socket connections, as the sinks. The existence of sensitive data flow pointing to a network transmission call that stays in the notification SDK will indicate a data uploading behavior.

4) Data Collection Findings: We next unveil some representative data uploading behaviors. Note that the data collection behaviors exhibited here are discovered and verified in the empirical study with our proposed detection schemes. We bring these findings upfront to help readers better comprehend the risk of privacy exposure. According to our exploration among various mobile notification providers and the released apps, the collected data can be categorized as follows.

Basic Device Information. We observe collecting such information is a common phenomenon in mobile notification SDKs. In most platforms, device model, os version, network connection state, and other device information are all automatically uploaded when an app is initialized, leading to potential privacy exposure. Such information is used by the platform to conduct the application installation statistic analysis, device tracking, and notification channel connection maintenance, but they can be leveraged to infer users' privacy information.

User In-app Actions. Some sensitive data collection behavior comes from the user in-app action tracking, including the app usage time, user clicks actions, purchasing, among others. Action tracking can be achieved by three methods. First, the general action tracking, such as app use time or interface jumping, is automatically collected when the corresponding event is triggered. Second, developers can call event tracking APIs at specific action time points. For example, developers can call the function "Leanplum.trackGooglePlayPurchase()" in their in-app payment code section, to upload user's purchasing behaviors in Google Play. Third, some SDKs provide APIs for developers to define special actions. For example, "Kumulos.Current.TrackEvent(EventName)" allows the app to track actions with the developer-defined event name.

Location Trace. Location is the most sensitive information that is related to user's routine. Unsurprisingly, mobile notification SDKs are eager to collect them. Users grant the app location permissions to allow it to fulfill the normal functionalities, however, the notification SDK will collect the location data to the notification platform, thereby causing privacy exposure. Through our analysis, we summarize three types of location collection, as follows.

1. **Network State Location.** Mobile notification SDKs can track the location through IP or MAC address at the coarse-level even users disable the location permission.
2. **GPS-based Location.** Location information can be uploaded by directly sending the geographic coordinates. Once the host app applies the location permissions, the notification SDK can access the geographic information and upload them through the APIs. Most notification

SDKs possess the location uploading APIs, such as in Kumulos: "Kumulos.SendLocationUpdate()".

3. **Geofencing.** The notification SDK makes use of the geofencing supported by Android to accomplish location-aware message delivery. To create Android geofencing, developers need to configure the latitude and longitude as the circle center, and set the radius to define a circular area. When the device enters or exits the area, the geofencing service can automatically generate Android events. This kind of location-aware event can be utilized by the notification platform to perform location-based notification messaging. That is, the developer first configures the Android geofencing area in apps and sets the same area at the notification console with the corresponding message that needs to be delivered to the device. Then, the developer calls the geofencing event collection APIs supported by notification SDKs. Once a user enters or exits this area, the Android geofencing event will be uploaded to the notification platform and triggers the message delivery. Although the geographic coordinates are not uploaded, the location information that a user arrives at an area can still be collected by mobile notification platforms.

B. Notification Monitoring

We next explore the notification monitoring behaviors, which may be leveraged to infer users' other private information. Since notification messages may carry sensitive contents or special events reminders, by monitoring and capturing them through malicious apps, an attacker can infer users' sensitive information, leading to serious privacy exposure. Note that, such monitoring is not the behavior of notification SDKs or the apps that push this notification message. It is executed by the malicious app that is developed by an attacker, which pretends to perform normal behaviors while stealthy sniffing the notifications from other apps. This threat raises a new side-channel attack method for inferring user privacy.

1) Android Notification Permission Flaw: The Android permission mechanism protects notification contents with the "BIND_NOTIFICATION_LISTENER_SERVICE". Apps typically require applying for this permission and waiting for the user granting. If a user grants this permission to the app, the notification listener function in the app can listen to the respective notification events and obtain the notification messages shown on the notification bar. However, the design flaw of this mechanism is that the notification bar is not isolated app by app, thus any app with this permission can access all messages in this bar. In other words, notification listeners in different apps can access all messages on the notification bar even if the messages are from other apps. Such shared access to the notification bar can cause the risk of side-channel attacks which allow a malicious app to monitor the notification messages for user privacy inference. In practice, the malicious app can pretend to be a normal one and perform its normal behavior, such as listening to the notification for automatically receiving verification codes. This can induce the user to grant the corresponding permission for monitoring notifications. But, it can stealthily monitor the sensitive message from other apps for privacy inference.

2) Potential Privacy Inference through Notification: We further discuss the possibility of privacy inference, if notification messages are monitored by apps.

1. **Financial Status Monitoring.** Mobile financial apps, such as PayPal, Alipay, Amex, *etc.*, often deliver some notifications containing a user's financial information, such as "received money transfer xxx$". By monitoring such financial mobile notifications, a malicious app can acquire a user's rough financial status description.
2. **Location Tracking.** As discussed in Section IV-A4, apps can perform location-based notification delivery with Android geofencing. Such location-aware notification messages can be used by other apps for location tracking.
3. **User Portrait.** Plenty of apps are supported by recommendation systems to learn users' interests and then provide personalized notifications to direct users to click on their apps. Intuitively, the set of user mobile notification messages can be used to learn the user Portrait.

Hence, the mobile notification messages contain plentiful information to be utilized by attackers for inferring user privacy with inference methods proposed in [38], [48], [28], [59], [33]. We will provide the statistic data for released apps requiring the notification listening permissions in Section V-D4.

V. Empirical Study

We conduct extensive empirical studies from different perspectives, *i.e.,* documents analysis, apps collection and analysis, case studies, *etc.,* to examine the mobile notification systems, aiming to expose the potential security and privacy issues that have been presented in Sections III and IV. Our goal is twofold. First, we reveal vulnerabilities lurking in mobile notification services. Second, we expose the privacy issues from notification SDKs and released apps in the market. Some case studies are also conducted.

A. Notification Platforms and Apps Collection

We collect mobile notification platforms primarily via Internet crawling with keywords, such as "Mobile Notification", "Mobile Pushing", "Cloud Messaging", and many other related descriptions. In addition, we parse blogs and statistic news with the topics of mobile notifications, such as [42], to refine and enlarge platform collections. In total, 12 most popular mobile notification platforms are identified, covering the majority of mobile notification markets in Europe, North-America, and Asian areas. We collect all versions of notification SDKs, user documents, and application demos of these platforms for analysis. For the released apps, we collect the top-100 apps in each category in app stores, such as Google Play, CoolApk, and APKpure, *etc*, obtaining a total of 31049 apps. Our crawling starts from December 2019 to February 2020 and from August 2021 to September 2021.

We conduct a large-scale analysis of our collected 31049 apps and find that over half of the Internet-required apps, *i.e.,*

TABLE I
THE LIST OF COLLECTED 15 NOTIFICATION PLATFORMS

Notification Platforms	Homepage Website
Airship	https://www.airship.com/
Getui	https://www.getui.com/
Jpush	https://www.jiguang.cn/
Kumulos	https://www.kumulos.com/
Leanplum	https://www.leanplum.com/
Mobpush	https://www.mob.com/
OneSignal	https://onesignal.com/
Pushbot	https://pushbots.com/
Pusher	https://pusher.com/
Pushwoosh	https://www.pushwoosh.com/
Taplytics	https://taplytics.com/
Umeng	https://www.umeng.com/

TABLE II
INSTALLATION STATISTICS OF COLLECTED APPS CORRESPONDING TO 12 MOBILE NOTIFICATION PLATFORMS

Notification Platforms	App Amounts	Installation Amounts
Airship	1705	291,000,000+
Getui	2279	510,000,000+
Jpush	1564	520,000,000+
Kumulos	1417	427,000,000+
Leanplum	471	680,000,000+
Mobpush	1985	1,170,000,000+
OneSignal	1687	870,000,000+
Pushbot	231	1,100,000+
Pusher	126	2,500,000+
Pushwoosh	307	150,000,000+
Taplytics	482	70,000,000+
Umeng	4015	920,000,000+

16269 of 31049, have adopted third-party notification SDKs to fulfill their notification services, having billions of installation amounts. The names of notification platforms, app amounts, and the installation amounts are summarized in Table II.

B. Static Analysis in Collected Apps

We develop an automatic analytical tool, NotiLeak, to automatically analyze the apps for exposing security and privacy issues. The workflow of NotiLeak is detailed below:

Phase 1: Notification SDK Identification: NotiLeak decompiles the collected Android application APK files into the analyzable intermediate code and generates the usable code resources. It integrates the classical app analysis tool Apktool [7] to extract resource files from the APK file. NotiLeak adopts a three-step identification framework to fulfill the time efficiency requirement of our large-scale analysis. Details are shown as follows:

Step 1: Permission-based Filtering. This step aims to filter the required permissions for receiving the notification messages. Two types of permissions are considered, i.e., "android.permission.INTERNET" and "OP_POST_NOTIFICATION", one serves for the network connection and the other helps enable the notification function, respectively. The permission "android.permission.INTERNET" can be directly checked from the AndroidManifest.xml file while the "OP_POST_NOTIFICATION" is dynamically configured at the application runtime, demanding our searching of resources code. Only the applications that enable both permissions will be analyzed.

Step 2: SDK Identifier Matching. This step aims at the fast notification SDK identifier matching. Since the number of notification platform SDKs studied in this paper is determined, it is feasible to obtain certain keywords and static features in the notification library SDK. These SDK identifiers are the combination of provider's names and the important initialization calls. As such, we create a list of identifiers based on platform SDK collections to support fast matching. After this step, NotiLeak could find out most of the desirable applications, while the applications that cannot match any identifier will be sent to step 3.

Step 3: Structure-based Identification. In practice, app developers conventionally use obfuscation tools (e.g., ProGuard [27], DexProtector[2], and DexGuard [1]) to prevent reverse engineering. There are currently two common obfuscation strategies, namely deadcode removal and identifier renaming. The deadcode removal is to remove unused functions in the SDK which are prone to expose the SDK packages characteristics. The identifier renaming strategy may obfuscate package/class/method/variable names to meaningless characters. However, the hierarchical structure of the classes, inherent Android system APIs, class inheritance relationship, and vital function call graph remains invariant under two kinds of obfuscation strategies. NotiLeak adopts such architectural characteristics to detect the SDK. According to collected SDKs, NotiLeak builds their architectural signatures and conducts a fine-grained searching among the applications to detect the apps that contain obfuscated SDKs. In practice, we have conducted a small scale of testing on 36 apps developed by ourselves (i.e., 3 apps for each platform with different obfuscation methods) to demonstrate that NotiLeak can recognize all notification SDKs.

Phase 2: KEY Misuses Analysis: The NotiLeak conducts the KEY misuses analysis based on the methods we have provided in Section III-A. Recall that, it will extract the suspect strings in the app source codes and automatically send the crafted message to the notification messaging APIs. By examining the returned error code, it can identify the misuse of KEY.

Phase 3: Privacy Policy Analysis: The NotiLeak performs privacy policy analysis based on the source code analysis methods we have provided in Section IV-A2. All analysis results corresponding to each app will be recorded for further statistic analysis.

Phase 4: Data Collection Analysis: Following the taint analysis methods we have provided in Section IV-A3, NotiLeak will extract the sensitive data flow in apps. These data flows will be classified into different types, such as user in-app actions and location traces, recorded for statistic analysis.

Phase 5: Notification Monitoring Analysis: In the end, Notileak conducts notification monitoring inspections by filtering out the apps that apply for the notification listening permission. All the detected apps will be marked as the suspect app for further analysis.

C. Security Inspection Results

With the resources including user documents, application demos, and our testing apps, from 12 notification platforms, we examine their mobile notification services following our proposed security analysis in Section III. Six notification platforms are discovered to have vulnerable notification services, impacting around billions of users with security risks. We have informed these platforms about our findings for the purpose of ethical disclosure. Detailed results are illustrated as below.

1) KEY and AppId Misuse: We examine the KEY and APPId names as well as their storage guideline among these 12 platforms. Three platforms, *i.e.,* Airship, Taplytics, and Kumulos, are found to name both parameters as KEY with different prefixes, prone to lead confusion to developers. Then, we conduct the KEY inspection among released apps in order to detect the mistaken KEY storage. Our results indicate that app developers store the KEY for messaging API in 174 apps with more than 500,000+ installations. We verify our findings by using our approach proposed in Section III-A, where the messaging API returns the error code "device not found" rather than "KEY incorrect". Note that, the attacker with these keys can send the mal-notifications to real users. Among 174 apps, 88 apps adopt the Kumulos, 63 apps adopt the Airship, and 23 apps adopt the Taplytics. This observation demonstrates that the similar names between KEY and APPId, can cause serious security issues. We have contacted the three platforms and suggested them to distinguish the two names.

2) Application Authentication Inspection: We inspect the application authentication mechanism of all platforms. For ethical considerations, we perform our proposed attack (i.e., Section III-B) on two applications developed by ourselves corresponding to each platform. Our results exhibit that all 12 notification platforms are free from this security issue.

3) Notification Messaging API Analysis: To discover the insecure notification messaging APIs, our analysis follows two criteria: 1) whether the RESTful API is protected by HTTPS? and 2) whether a strong authentication is adopted?

We discover that three platforms adopt insecure messaging APIs with HTTP. That is, Pushbot and Mobpush only provide the HTTP URL while Umeng provides both HTTP and HTTPS URLs for better compatibility. In particular, the Pushbot platform, which is employed by 231 apps with 1,100,000+ installations, only adopts the basic HTTP authentication method, resulting in the unprotected plaintext transmission of KEY. There is also no notification message verification mechanism to check the integrity of payloads. Consequently, an attacker can perform the man-in-the-middle attack to steal the KEY or tamper the notification messages.

Umeng and Mobpush provide the HTTP URL messaging API and then employ MD5-based message verification to protect the notification from tampering. But, their methods cannot prevent the insider threat as proposed in Section III-C, whereas an attacker can modify the notification payloads without changing the MD5 signature. Notably, the Mobpush is employed by 1985 apps, which can cause severe security risks to its involved billions of installations.

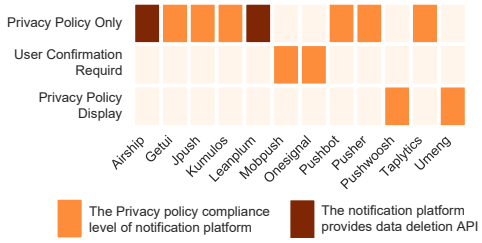

Fig. 3. Privacy policy compliance levels among 12 notification platforms.

4) Insider Attack Case Study: We conduct a case study to perform the insider attack towards Umeng messaging API, validating that the MD5 collision-based insider attack is practical in real-world notification systems. For ethical considerations, we only conduct the attack on the app that was developed by ourselves. That is, we develop a testing app that carries the Umeng notification SDK. Then, we generate a notification message with the weather forecast as the target and design an adversary notification message with the advertising content. The open resource MD5 collision attack on Github [49] is employed to calculate the chosen-prefix MD5 collision blocks, for the weather forecast and advertisement contents. The calculation was processed on a computer with CPU: Intel i7-8700k, GPU: NVIDIA GeForce GTX1080 Ti, and RAM: 64G, where the GPU acceleration was employed. It takes 4 days to find a collision with the two prefixes while the generated collision blocks do not include the JSON keywords. We put collision blocks in the optional field, *i.e.,* "extra:", and the collided MD5 value is "9b4af15c5b932858f26cb22f3420a86c". We start to push the notification message with weather forecast payloads to the messaging API. Then, the insider attacker replaces the payloads with colliding advertising payloads and delivers them to the messaging API. The testing user device receives the pushed notification with the modified content. The two collision files in this attack are exhibited in [4] with the file names of "prefix.txt.coll" and "prefix2.txt.coll".

D. Privacy Exposure Analysis

Next, we focus on privacy exposure for apps carrying notification SDKs and aim to answer the following questions: 1) whether users are informed about the privacy policy of notification services? 2) whether these notification SDKs upload users' data to their platforms? and 3) how much user data is leaked to the notification platforms? Our explorations of sensitive data exposure shall raise important concerns regarding privacy protection and mobile notification supervision.

1) Privacy Policies Study: We first analyze the privacy policy compliance levels and their implementations, aiming to expose whether the notification platforms comply with data collection regulations and whether users are informed in practical development.

According to our document analysis, the privacy policy compliance levels among these 12 mobile platforms are shown in Fig. 3. We can observe that 8 of 12 platforms only provide the privacy policy on their websites, claiming that their SDKs will collect user data. Among these 8 platforms, Airship and Leanplum provide the data deletion APIs to app developers for

TABLE III
STATISTICS OF APIS RELATED TO PRIVACY POLICIES IMPLEMENTATION

Platforms	App Amounts	Percentage
Mobpush	114	5.74%
OneSignal	121	7.17%
Pushwoosh	54	17.59%
Umeng	26	0.65%
Total	318	3.98%

erasing the user data collected by their SDKs. On the other hand, only 1/3 of platforms provide the privacy policy APIs, where 2 platforms, *i.e.,* OneSignal and Mobpush, provide APIs for acquiring user agreement parameters, and 2 platforms, *i.e.,* Pushwoosh and Umeng, design UIs and APIs for displaying their privacy policy to users.

We inspect the released apps that are employing the four platforms (OneSignal, Mobpush, Pushwoosh, and Umeng) and verify if their respective APIs are indeed implemented in the development. The results are listed in Table III. Surprisingly, we discover that less than 4% of apps are actually calling the privacy policy-related APIs before initializing mobile notification services. Some developers hard-code user confirmation parameters, with 10 apps (7 apps with OneSignal and 3 apps with MobPush) having this issue. This observation is quite astonishing as only 1/3 platforms provide the APIs, so their practical development is even worse. As such, we conclude that it is a common case: users are never aware of the stealthy data collection from notification platforms.

2) Stealthy Data Collection: To exploit the stealthy data uploading behaviors, we employ the testing apps as we have mentioned in Section IV-A3 to analyze such behaviors when the apps are running on the testing smartphone (Samsung Galaxy S9+). We use Wireshark to capture the network packets from this device and check the account console of notification platforms to examine user data. The statistical results are shown in Table IV, with details depicted as follows.

Six platforms collect user in-app behaviors. We find that 6 notification SDKs (i.e., Airship, Jpush, Kumulos, Leanplum, OneSignal, and Pushwoosh) upload the user in-app behaviors, such as the app running time, user click events, and others, as shown in Table IV. Besides, Leanplum SDK can collect the users' Google Play purchasing behaviors. Such a data collection behavior is quite similar to that of the analytical SDK. We further discover that the other two notification platforms, *i.e.,* Getui and Umeng, separate the analytical SDK and notification SDKs, so that the user in-app behavior will not be collected in the notification SDK.

Nine platforms collect location information. Nine notification SDKs (i.e., Airship, Getui, Jpush, Kumulos, Leanplum, Mobpush, OneSignal, Taplytics, and Umeng) are found to upload location information, including IP, geographic coordinates, and geofencing events. Once the location sharing functions are called in the application such as "OneSignal.setLocationShared()", the device location information will be shared to the respective notification platform. Although such functions facilitate the application development, the location information will be shared without users' consent, representing a fatal sensitive information leakage. Considering that massive amounts of applications could access notification services from the same platform, such data exposure issues could explain why users always receive over-precise location-based notifications in some applications even they never grant the corresponding permissions.

3) Released Apps Inspections: We further analyze the collected 16269 applications with notification services for detecting privacy exposure issues. We conduct the taint analysis as proposed in Section IV-A3 to automatically scan the apps and generate the sensitive data flows. Our analytical results reveal that 6705 apps (over 41%) upload the sensitive user data (*i.e.,* in-app behavior and location) to notification platforms. Their detailed distributions are shown in Table IV. Such results indicate that the user data exposure issues, especially location data uploading (detected in 5567 apps), are the common phenomenon. In addition, user in-app information collection is also found in 1138 apps, including page jumping, app using time, purchase actions, and other actions, which possess billions of installations.

To highlight the severity of user data exposure, we list some apps with large installation amounts. For instance, one app (employing Taplytics platform) called "Monster Job Search" has 5,000,000+ installations, which is a job searching application that requires location information to support its social community. Our taint analysis detects a data flow from the system location APIs to the Taplytics location uploading function, *i.e.,*"optInTracking", indicating that the notification SDK shares the user location with Taplytics. Moreover, a local shopping app (employing Getui) having 500,000+ installations is also discovered to upload app installation lists, and a weather forecast app (employing Umeng) having 10,000+ installations is found to share the smartphone sensor data.

4) Notification Monitoring Study: Since the inference actions after acquiring notification messages occur on the server side, it is difficult to track them. So, our analysis will focus only on exhibiting the statistical results of released apps that apply for notification listening permission and raise attention to such a new side-channel attack. According to our analysis of collected apps, 245 apps with 1,420,000+ installations enable the listening permissions to capture the notification messages. They can monitor and collect the notifications to infer user privacy information. We further conduct a case study to exhibit notification monitoring behaviors. We develop a testing app that carries Kumulos notification SDK and let it pretend to be a normal weather app while applying for "BIND_NOTIFICATION_LISTENER_SERVICE" permission. However, this app is coded to record the message displayed in the notification bar. We install it on our smartphone, which also installs Alipay, DoorDash, and Twitter, to simulate the real scenario. After receiving a money transfer into the account in Alipay, our testing app succeeds in capturing the Alipay notification "xxx transfers 10 yuan to your account". Also, the nearby restaurant recommendation notifications from DoorDash, which can help to infer the location, and the notification of interested friends from Twitter, which can be further trained for user portrait inference, are all

TABLE IV
THE DATA UPLOADING BEHAVIOR IN 12 NOTIFICATION PLATFORMS AND APP AMOUNTS

Notification Platforms	User in-app Actions	App Amounts	Location Information	App Amounts
Airship	App usage time, User click event, Developer-defined action	217	IP address, GPS	561
Getui	None	0	IP address, GPS, Geofencing	1022
Jpush	App usage time, User click event, Developer-defined action	229	IP address, GPS, Geofencing	627
Kumulos	Developer-defined action	151	IP address, GPS, Geofencing	287
Leanplum	Developer-defined action,Google Play purchase	68	IP address	65
Mobpush	None	0	IP address, GPS	627
OneSignal	App usage time, User click event, Developer-defined action	402	IP address, GPS, Geofencing	874
Pushbot	None	0	None	0
Pusher	None	0	None	0
Pushwoosh	Developer-defined action	29	None	0
Taplytics	None	42	IP address, GPS,	69
Umeng	None	0	IP address, GPS, Geofencing	1435

captured. We hope that our results can raise public attention to privacy inference issues from mobile notification messages.

VI. RELATED WORK

Our work is related to the security vulnerability and privacy exposure regarding the messaging APIs, the notification SDKs, the mobile notification service.We briefly discuss the existing works and differentiate our work with them.

To discover vulnerabilities of network protocols and web APIs, different automatic tools have been proposed. For example, [15], [32], [16], [20] have developed Polyglot, AutoFormat, Dispatcher, and Discoverer, respectively to automatically dissect the web protocols. Regarding web API-based apps, Waptec [13] and NoTamper [12] were proposed to identify parameter tampering vulnerabilities while [37] proposed a solution to detect the mobile app-to-web API communication inconsistency. Recently, [60] proposed LeakScope to identify the data leakage vulnerabilities on cloud APIs and [26] analyzed the authentication and authorization flaws of user account access in web APIs. On the other hand, the security and privacy analysis of third-party SDKs have also been widely studied. In [45], [46], [19], [23], authors explored the personal identification information leakage through third-party SDKs while [24], [11], [29], [47] studied the privacy exposure caused by the Android permission inheritance mechanism.Targeting widely-adopted SDKs, in [36], [35], authors examined what information is collected by the analytics library in the Android apps. In [21], [38], [40], [43], [57], authors explored the data collection issues of in-app advertising and payment SDK. Different from them, we target the mobile notification service via analyzing both the API and the notification SDK.

Regarding mobile messaging services, some efforts have been done in unveiling the vulnerabilities and privacy issues. In [34], authors developed a tool for identifying aggressive notifications received at the device. Besides, [8], [39] detected the malicious apps with the misuse of Google Firebase Cloud Messaging, while [56] discussed phishing and spamming attacks by abusing notification services on smartphones. In [17], a detection tool Seminal was designed to extract semantic information from source code. In [31], authors analyzed Google older GCM messaging system, revealing the vulnerabilities of stealing or wiping sensitive messages and of installing or uninstalling apps on a user's device.

VII. LIMITATIONS AND FUTURE WORK

This section discusses our limitations and the future work.

First, this paper provides the first comprehensive and systematic study toward mobile notification services in the Android ecosystem. However, the corresponding study of the iOS ecosystem remains open. Per our preliminary investigation, the Apple corporation provides its own notification platform and APIs for the applications to push mobile notifications, which is quite different from the Android system where the notification services employ third-party platforms. Hence, the corresponding security and privacy analysis toward mobile notification services in the iOS system calls for new analytical approaches, which are deferred to our future work.

Second, we have developed a tool NotiLeak for automated analysis, including KEY misuse, privacy policy, data collection, and notification monitoring analysis, which can help us significantly speed up the inspections on massive amounts of apps in the Android market. Considering many strong techniques have been proposed, our NotiLeak can be further improved to integrate them, to make our NotiLeak more effective and efficient. For instance, some techniques proposed [9], [58], [25] for tracking the data flow can be integrated into NotiLeak with the customized design to improve our privacy policy compliance analysis. On the other hand, our current documentation analysis relies on manual checking, which is cumbersome and ineffective. In the future, we plan to employ the existing techniques [18] or develop new tools, to be integrated into our NotiLieak, for enabling the automated documentation analysis.

Third, our notification monitoring analysis has exhibited that an attacker can acquire permission to monitor other apps' notification messages displayed in the notification bar, which can potentially incur the privacy inference risk as discussed in Section IV-B2. But, how to infer a user's private information based on these messages remains unexplored, which is not the goal of our current work. Hence, in our future work, we plan to develop inference attack solutions based on these notification messages, which can further exhibit the severe consequences of privacy risks incurred from mobile notification services.

VIII. CONCLUSION

In this paper, we have conducted the first comprehensive analysis toward the mobile notification ecosystem. We analyzed the employment of third-party notification platforms and detected three vulnerabilities derived from the misconfiguration of KEY, weak application authentication, and weak message verification mechanism. Regarding the privacy leakage issues, we analyzed the privacy policy compliance implementation and proposed a top-down scheme to explore the sensitive

data collection from both notification platform and application perspectives. Our empirical studies on 12 popular notification platforms discovered several insecure mobile notification designs and uncovered that the privacy policy compliance is under the subpar implementation. In addition, over 50% of 30,000+ applications are exposed to stealthily collect user data, impacting billions of users. We hope our efforts will not only inspire the in-depth research toward exploring the security and privacy issues in mobile notification ecosystem, but also raise public attention to regulate the design and implementation of mobile notifications.

ACKNOWLEDGMENT

This work was supported in part by NSF under Grants 1763620, 1948374, 2019511, and 2146447. Any opinion and findings expressed in the paper are those of the authors and do not necessarily reflect the view of funding agency.

REFERENCES

[1] Dexguard android obfuscator. https://www.guardsquare.com/dexguard.
[2] Dexprotector android obfuscator. https://dexprotector.com.
[3] General data protection regulations. https://gdpr-info.eu/.
[4] Md5 collision materials. https://www.dropbox.com/sh/gviy9s1xenbl3fk/AABZLVCAqIZH81e_KlRbgmFta?dl=0.
[5] The personal data protection bill. https://prsindia.org/billtrack/the-personal-data-protection-bill-2019.
[6] Provisions on the scope of necessary personal information for common types of mobile internet applications. http://www.cac.gov.cn/2021-03/22/c_1617990997054277.htm.
[7] Apktool. http://ibotpeaches.github.io/Apktool/, 2016.
[8] Mansour Ahmadi, Battista Biggio, Steven Arzt, Davide Ariu, and Giorgio Giacinto. Detecting misuse of google cloud messaging in android badware. In *Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 103–112, 2016.
[9] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. Policylint: Investigating internal privacy policy contradictions on google play. In *USENIX Security Symposium*, pages 585–602, 2019.
[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.
[11] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 356–367, 2016.
[12] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and VN Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, pages 607–618, 2010.
[13] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and VN Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, pages 575–586, 2011.
[14] John Black, Martin Cochran, and Trevor Highland. A study of the md5 attacks: Insights and improvements. In *Proceedings of International Workshop on Fast Software Encryption*, pages 262–277. Springer, 2006.
[15] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, pages 621–634, 2009.
[16] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, pages 317–329, 2007.
[17] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1260–1272, 2015.
[18] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *USENIX security symposium*, pages 747–764, 2019.
[19] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
[20] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of USENIX Security Symposium*, pages 1–14, 2007.
[21] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. Free for all! assessing user data exposure to advertising libraries on android. In *Proceedings of Annual Network and Distributed System Security symposium (NDSS)*, 2016.
[22] Bert Den Boer and Antoon Bosselaers. Collisions for the compression function of md5. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 293–304. Springer, 1993.
[23] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2187–2200, 2017.
[24] Michalis Diamantaris, Elias P Papadopoulos, Evangelos P Markatos, Sotiris Ioannidis, and Jason Polakis. Reaper: Real-time app analysis for augmenting the android permission system. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 37–48, 2019.
[25] Zikan Dong, Liu Wang, Hao Xie, Guoai Xu, and Haoyu Wang. Privacy analysis of period tracking mobile apps in the post-roe v. wade era. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.
[26] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1953–1970, 2020.
[27] Google. Proguard. http://developer.android.com/tools/help/proguard.html, 2014.
[28] Payas Gupta, Swapna Gottipati, Jing Jiang, and Debin Gao. Your love is public now: Questioning the use of personal information in authentication. In *Proceedings of the ACM SIGSAC symposium on computer and communications security (ASIA CCS)*, pages 49–60, 2013.
[29] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1037–1049, 2017.
[30] Arjen K Lenstra, Xiaoyun Wang, and BMM de Weger. Colliding x. 509 certificates. https://eprint.iacr.org/2005/067, 2005.
[31] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 978–989, 2014.
[32] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–15. Citeseer, 2008.
[33] Jack Lindamood, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. Inferring private information using social network data. In *Proceedings of the international conference on World Wide Web (WWW)*, pages 1145–1146, 2009.
[34] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. Dapanda: Detecting aggressive push notifications in android apps. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 66–78. IEEE, 2019.
[35] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. Privacy risk analysis and mitigation of analytics libraries in the an-

droid ecosystem. *IEEE Transactions on Mobile Computing (TMC)*, 19(5):1184–1199, 2019.

[36] Xing Liu, Sencun Zhu, Wei Wang, and Jiqiang Liu. Alde: Privacy risk analysis of analytics libraries in the android ecosystem. In *International Conference on Security and Privacy in Communication Systems (CCS)*, pages 655–672, 2016.

[37] Abner Mendoza and Guofei Gu. Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *Proceedings of Symposium on Security and Privacy (S&P)*, pages 756–769. IEEE, 2018.

[38] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[39] Mohamed Abdalla Mokar, Sallam Osman Fageeri, and Saif Eldin Fattoh. Using firebase cloud messaging to control mobile applications. In *Proceedings of International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, pages 1–5. IEEE, 2019.

[40] Suman Nath. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 59–73, 2015.

[41] Business of Apps. Push notifications statistics (2019). https://www.businessofapps.com/marketplace/push-notifications/research/push-notifications-statistics.

[42] Business of Apps. Top push notifications services (2020). https://www.businessofapps.com/marketplace/push-notifications/, 2020.

[43] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 71–72, 2012.

[44] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2014.

[45] Jingjing Ren, Martina Lindorfer, Daniel J Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. A longitudinal study of pii leaks across android app versions. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[46] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 361–374, 2016.

[47] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[48] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *Proceedings of Annual Network and Distributed System Security symposium (NDSS)*. Citeseer, 2016.

[49] Marc Stevens. Md5 and sha-1 cryptanalytic toolbox. https://github.com/cr-marcstevens/hashclash.

[50] Marc Stevens. On collisions for md5. https://www.win.tue.nl/hashclash/, 2007.

[51] Marc Stevens, Arjen Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and colliding x. 509 certificates for different identities. In *Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–22. Springer, 2007.

[52] Marc Stevens, Arjen K Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and applications. *International Journal of Applied Cryptography*, 2:322–359, 2012.

[53] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne De Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In *Proceedings of Annual International Cryptology Conference*, pages 55–69. Springer, 2009.

[54] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35, 2005.

[55] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of European Symposium on Research in Computer Security*, pages 200–215, 2009.

[56] Zhi Xu and Sencun Zhu. Abusing notification services on smartphones for phishing and spamming. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[57] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[58] Le Yu, Xiapu Luo, Jiachi Chen, Hao Zhou, Tao Zhang, Henry Chang, and Hareton KN Leung. Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies. *IEEE Transactions on Software Engineering*, 47(2):221–242, 2018.

[59] Elena Zheleva and Lise Getoor. To join or not to join: the illusion of privacy in social networks with mixed public and private user profiles. In *Proceedings of the international conference on World Wide Web (WWW)*, pages 531–540, 2009.

[60] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 1296–1310. IEEE, 2019.