An Empirical Study of Security Risks in the Android Push Notification Ecosystem

Shilong Hu*
Huazhong University of Science and Technology
Wuhan, China
hushilong7@hust.edu.cn

Zikan Dong*
Huazhong University of Science and Technology
Wuhan, China
zikandong@hust.edu.cn

Chao Wang
Huazhong University of Science and Technology
Wuhan, China
chaowang_@hust.edu.cn

Tianming Liu†
Huazhong University of Science and Technology
Wuhan, China
tmliu@hust.edu.cn

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

Push notification services are widely used in mobile apps to deliver time-sensitive content. In the fragmented Android ecosystem, however, no single push service works reliably across all devices and markets, leading developers to integrate multiple push SDKs from device manufacturers and third-party providers. Despite their widespread use, prior studies have focused mainly on Google's Firebase Cloud Messaging, leaving the security risks of other push services largely unexplored. In this paper, we present a systematic security assessment of the Android push notification ecosystem, covering 11 mainstream push SDKs and 52,748 apps across multiple Android app markets. Our analysis reveals three classes of security risks across the push notification workflow: server secret leakage, notification hijacking, and plaintext exposure of notification payloads. Specifically, we identify 517 apps that leak server secrets, accounting for at least 78.79 billion cumulative installs. These leaks enable attackers to forge notifications with attacker-controlled titles and bodies, redirect users to arbitrary URLs after a click, and, in 9 of the 11 evaluated push services, broadcast such notifications to all users. We further show that four widely used push SDKs are vulnerable to notification hijacking, enabling attackers to reroute victim notifications. In addition, among the 110 communication-oriented apps we evaluated, none protects notification payloads before they enter push service infrastructure, leaving sensitive content exposed in plaintext. These findings show that securing real-world push services requires stronger protections from both app developers and push service providers, highlighting the need for more secure push deployment in industry practice. We responsibly disclosed the issues to push service providers and app developers, and received acknowledgments from affected parties.

*Shilong Hu and Zikan Dong contributed equally to this research.
†Tianming Liu is the corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834475

CCS Concepts

• **Security and privacy → Software and application security**; **Mobile platform security**.

Keywords

Android Security; Push Notification Service; Third-party SDK; Mobile Application

ACM Reference Format:
Shilong Hu, Zikan Dong, Chao Wang, Tianming Liu, and Haoyu Wang. 2026. An Empirical Study of Security Risks in the Android Push Notification Ecosystem. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834475

1 Introduction

Push notifications are a core communication mechanism in modern mobile apps, enabling timely delivery of messages, alerts, transaction confirmations, and other user-facing updates. For Android apps, however, notification delivery is complicated by ecosystem fragmentation: no single push service provides reliable coverage across all devices and markets. To improve delivery reliability, developers therefore commonly integrate multiple push SDKs from device manufacturers and third-party providers. However, this multi-SDK strategy requires developers to manage different security assumptions and configurations across multiple push services, increasing system complexity and security management overhead.

The security implications of this ecosystem remain insufficiently understood in three important respects. First, prior work [31, 44] has focused mainly on Google's Firebase Cloud Messaging, leaving the broader landscape of push services from device manufacturers and third-party providers largely unexplored. Second, existing large-scale measurements [36] are often confined to Google Play, which does not fully capture the fragmented Android markets in which alternative push services are more prevalent. Third, prior studies [32, 35, 41] have mostly examined isolated problems such as abusive notification behaviors, identity-binding attacks, or provider-specific privacy leakage, rather than systematically analyzing security risks across the push notification workflow.

To address these gaps, we conduct a systematic security assessment of the Android push notification ecosystem. We extend the scope beyond Google's Firebase Cloud Messaging by analyzing 11 mainstream push SDKs, including push services from device manufacturers such as Huawei Push Kit, and widely deployed third-party providers. To better reflect the fragmented Android ecosystem beyond Google Play, we further include apps from Huawei AppGallery, one of the most prominent alternative Android app markets, and build a dataset of 52,748 real-world apps. This broader coverage gives us a more realistic view of security risks in real-world Android push service deployments. In addition, rather than focusing on isolated notification abuses or provider-specific issues, we systematically examine security risks across the push notification workflow, including server secret leakage, notification hijacking, and plaintext exposure of notification payloads.

Our security assessment yields three main findings. First, across 52,748 apps, we identify 517 that leak server secrets, accounting for at least 78.79 billion cumulative installs. These leaked secrets give attackers high-privilege control over notification delivery, allowing them to forge notifications with attacker-controlled titles, bodies, post-click URLs, and delivery targets. Such control can be abused for phishing, scams, and other social-engineering attacks by redirecting users to arbitrary websites after a click. For several providers, leaked secrets also expose management APIs that allow attackers to access notification history and manipulate push configurations. Second, we show that four widely used push SDKs are vulnerable to notification hijacking, allowing attackers to prevent victims from receiving notifications, redirect victim-targeted messages to attacker-controlled devices, and thereby expose private notification content to unauthorized recipients. Third, among 110 communication-oriented apps we evaluate, none protects notification payloads before they enter push service infrastructure, leaving sensitive content exposed before delivery. This risk is closely related to a common push delivery design, in which notification content is handed directly to the push SDK for display before the app can process it. Overall, these findings reveal that widely deployed push services still expose systemic security risks in real-world Android deployments, underscoring the need for stronger safeguards in industry practice. We responsibly disclosed the identified issues to push service providers and app developers, and received acknowledgments from affected parties.

In summary, our work makes the following contributions:

- We present a systematic security assessment of the Android push notification ecosystem, covering multiple push service providers and app markets rather than focusing on a single provider or distribution channel.
- We conduct a workflow-oriented security analysis spanning device registration and message delivery, through which we identify three security risks: server secret leakage, notification hijacking, and plaintext exposure of notification payloads.
- We provide empirical evidence that these risks have serious real-world impact, including attacker-controlled notification delivery, notification rerouting, and exposure of sensitive content within push service infrastructure.

Organization We begin in Section 2 with the Android push notification workflow and its security foundations. In Section 3, we present the three security risks together with the corresponding threat model. Section 4 describes our analysis approach, and Section 5 presents our experimental results. We then discuss mitigation strategies and limitations in Section 6, review related work in Section 7, and conclude the paper in Section 8.

2 Background

2.1 Android Push Notification Workflow

The Android push notification workflow involves four core entities: (1) the app client on the user's device, (2) the push SDK integrated into the app, (3) the cloud server operated by the push service provider, and (4) the app server maintained by the developer. As shown in Figure 1, the Android push notification workflow consists of two phases: *Device Registration* and *Message Delivery*.

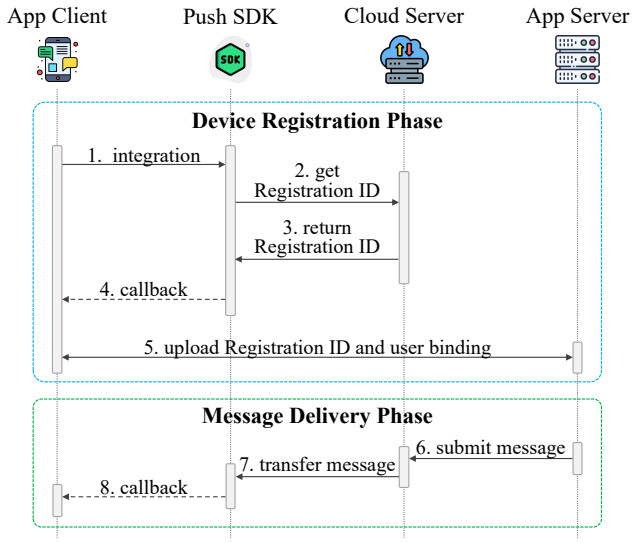

Figure 1: Standard Android push notification workflow, including device registration and message delivery.

Phase 1: Device Registration. Before push messages can be delivered, the app must first register the device with the push service. This phase starts with SDK integration (Step 1 in Figure 1), where the developer embeds the push SDK into the app together with the required service credentials and configuration values (e.g., the `App Key`). When the app launches, the SDK initializes using these embedded settings and sends a registration request to the cloud server (Step 2). The cloud server then returns a Registration ID (RID) to the push SDK (Step 3), which serves as the device's unique identifier within the push service and is later used to target the device during message delivery. After receiving the RID, the SDK invokes a callback to notify the app client that registration has completed successfully (Step 4). The app client subsequently uploads the RID to the developer's app server (Step 5), allowing the backend to bind the app instance to its corresponding push address for future message delivery.

Phase 2: Message Delivery. Once device registration is complete, the developer's backend can send push messages through the

provider's push API. This phase starts with message submission (Step 6 in Figure 1). To send a message, the app server constructs a message payload and authenticates to the cloud server using the high-privilege `Server Secret`, a server-side credential required for privileged API access. Typically, the app server targets specific devices by specifying their previously stored RIDs. Some push services also support broadcast delivery, allowing developers to send messages without specifying individual device identifiers. After validating the request, the cloud server forwards the message to the target device through a persistent background connection, typically a long-lived TCP connection, and delivers it to the push SDK (Step 7). The final step is message handling on the device (Step 8). After receiving the payload, the push SDK processes it according to the message type: it either automatically displays a system notification for standard notifications or invokes a callback to pass the raw data to the app client for custom handling of pass-through messages.

2.2 Message Types and Security Foundations

The final step of the workflow already highlights the role of message types in push delivery. To support the later analysis, we briefly introduce these message types together with the authentication and identification mechanisms that determine how messages are submitted and routed to devices.

Message Types. Push services generally support two main message types: standard notifications and pass-through messages. Standard notifications are processed directly by the push SDK and displayed as system notifications. Their structure is therefore usually fixed, typically consisting of a title and a body, and their content is not handled by the app before display. By contrast, pass-through messages deliver raw data, often in JSON format, to the app without triggering an immediate visual alert. This design gives developers more flexibility, but it also means parsing, interpretation, and any protection of the message content must be handled by the app itself.

Authentication and Identification. While the way a push message is handled on the device depends on its message type, push delivery also requires mechanisms for authenticating requests and identifying target devices. For service authentication, developers typically obtain a public `App Key` and a private `Server Secret`. The `App Key` is embedded in the app during push SDK integration and used to initialize the SDK on the client side. The `Server Secret`, by contrast, is kept on the backend and used to authenticate server-side requests, such as push API calls and related management operations. For device identification, the push service assigns a RID to the device during registration, which is later used to address the device during message delivery. The exact mechanism for RID generation and binding varies across push service providers.

3 Insecure Practices

Within the push notification workflow shown in Figure 1, we examine three security risks that arise at different steps of device registration and message delivery. Specifically, we focus on Step 1, where the app integrates the push SDK and related credentials; Steps 2–3, where the push service assigns the RID; and Step 6, where the app server submits message payloads to the push service provider. Failures at these steps can lead to three distinct classes of risk: insecure integration may expose server secrets, insecure RID assignment may enable notification hijacking, and insufficient payload protection may expose sensitive content to push service infrastructure. These risks are distinct and can arise independently. Any one of them can lead to serious security consequences on its own. At the same time, they can all be understood within the same push notification workflow. The following subsections describe, for each risk, the underlying security assumption, how it fails in practice, and the resulting security impact, before briefly summarizing the corresponding threat model.

3.1 Server Secret Misconfiguration

Server secret misconfiguration refers to the exposure of highly privileged backend secrets in client-side app packages. In the push notification workflow shown in Figure 1, this risk arises at Step 1 of device registration, where the app integrates the push SDK and related credentials. A secure design requires that high-privilege backend credentials, especially the `Server Secret`, remain in developer-controlled backend environments and never be embedded into client-side code or resources. In practice, this assumption can be violated during SDK integration when developers package the server secret into the APK and treat it as a normal initialization parameter. This risk often stems from ambiguous SDK setup guidance and weak separation between client-side configuration values and server-side secrets. Once embedded in the app package, the secret can be extracted through reverse engineering.

Server secret misconfiguration can give attackers backend-level control over the push service. In the most direct case, attackers can send notifications with attacker-controlled titles and bodies; for some push services, such notifications can also be broadcast to all users of the app. Attackers then can craft alluring notifications to induce clicks and further control the redirection logic, which can be abused to route users to phishing pages, malicious landing pages, or unauthorized app promotions. For several push services, attackers may further access management functions beyond message delivery. These functions can include querying push history and modifying tags or segments, which can affect user management and app business logic. If the same secret is reused across multiple apps or shared through common service configuration, a leak in one app may also affect other apps maintained by the same developer.

3.2 Deterministic RID Hijacking

Deterministic RID hijacking occurs at Steps 2–3 of device registration, when replaying the same device-side identifiers causes the push service to assign the same RID. Ideally, at this step, the security assumption is that the RID should not be reproducible from device-side identifiers that can be harvested or spoofed. However, this assumption breaks down when the backend deterministically derives the RID from client-supplied identifiers, meaning that submitting the same identifiers repeatedly yields the same RID. In contrast, more robust designs incorporate server-side randomness or session-specific state, so replaying the same device-side identifiers alone does not reproduce the same RID.

If an attacker can replay device-side identifiers from a victim device, deterministic RID assignment may cause the push service to return the victim's RID and bind it to the attacker's connection. The push service then treats the attacker's device as the active

delivery route for that RID. As a result, the legitimate user may stop receiving notifications, and messages intended for the victim may be redirected to the attacker's device until the victim re-registers with the push service. Beyond service disruption, this also creates a direct privacy risk, as private notification content may be exposed to the attacker during the hijacked period.

3.3 Plaintext Payload Exposure

Plaintext payload exposure arises at Step 6 of message delivery, when the app server submits message payloads to the push service provider. At this step, a secure design requires that sensitive content should be protected before it enters provider infrastructure, rather than relying solely on transport-layer protection. This assumption breaks down when apps send messages in plaintext and depend only on TLS for protection. Although TLS protects the network channel from external eavesdroppers, it only secures the payload in transit; once the payload reaches the provider's gateway, it remains visible to the provider for routing and processing. In the standard notification path, the payload is rendered directly by the push SDK, leaving the app no opportunity to process or decrypt it before display. To preserve the simplicity of this model, developers often keep notification content in plaintext unless they explicitly redesign the notification flow or add their own end-to-end protection.

If the payload contains sensitive information, such as one-time passwords, private chat messages, or financial alerts, that information becomes visible inside provider infrastructure even when transport security is in place. Depending on provider-side operations, such content may also be retained in logging or analytics systems. As a result, sensitive notification content becomes exposed once it passes through the push service.

3.4 Threat Model

The three risks discussed above do not share a single adversary model. (1) For **server secret misconfiguration**, we assume an external attacker whose goal is to obtain the server secret and abuse the push service. This attacker can obtain the distributed APK, extract embedded secrets through reverse engineering, and invoke the provider's public push APIs. (2) **Deterministic RID hijacking** involves two practical attacker settings. In the first, the attacker is a network observer who can monitor traffic in which replayable identifiers are exposed, including the Android system identifier `android_id` and third-party-generated identifiers. In practice, this can happen when such identifiers are transmitted over observable HTTP traffic [40, 43], particularly when the victim and attacker are connected to the same public Wi-Fi network or when the traffic routes through an attacker-controlled Wi-Fi access point [22, 42]. In the second, the victim installs a seemingly benign but actually malicious app that, with standard non-root permissions, may harvest replayable identifiers from accessible external storage or legacy files and pass them to the attacker. The attacker can then replay these identifiers on an attacker-controlled device to spoof registration and hijack the victim's notification route. (3) **Plaintext payload exposure** does not require an active attacker who breaks TLS. Instead, the risk exists because sensitive content is sent in plaintext and can therefore be read by the push service provider during routing and processing, and may also enter internal logging or analytics systems depending on provider-side operations.

4 Approach

To study security risks in real-world push deployment, we design a workflow-oriented analysis approach for the Android push notification ecosystem. Our approach follows the push notification workflow introduced in Section 2 and is organized around the corresponding security assumption failures defined in Section 3. Figure 2 illustrates the overall architecture of our proposed approach. Specifically, we examine SDK integration to detect server secret misconfiguration (Section 4.1), registration to analyze deterministic RID hijacking (Section 4.2), and message submission to verify plaintext payload exposure (Section 4.3).

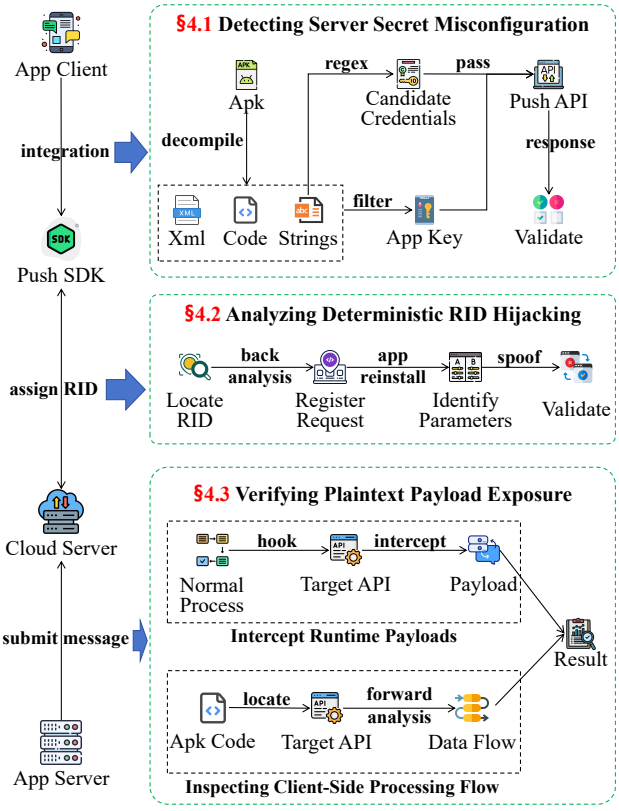

Figure 2: Overview of our workflow-oriented approach, consisting of three analysis components aligned with different steps of the push notification workflow.

4.1 Detecting Server Secret Misconfiguration

We begin with Step 1 of the push notification workflow, where developers integrate push SDKs and configure initialization credentials. Our analysis focuses on whether backend-only secrets are embedded in APKs and validates the extracted credentials through a three-stage pipeline consisting of APK preprocessing, credential filtering and pairing, and controlled API-based validation.

APK Preprocessing. In the first stage, we use static analysis tools such as Androguard [3] and ClassyShark [5] to decompile the target APKs. We then extract string constants from three common locations where push SDK credentials may be configured: the app's Dalvik bytecode (`classes.dex`), metadata entries in `AndroidManifest.xml`, and resource files such as `strings.xml`.

Filtering and Pairing Credentials. Using the extracted raw strings, we apply heuristic rules, such as length constraints and vendor-specific prefixes, together with regular expressions to construct a candidate pool of potential `App Keys` and `Server Secrets`. To verify whether a candidate `Server Secret` is valid, we first pair it with its corresponding `App Key`, which serves as the required initialization credential for the push SDK. Because developers must include the `App Key` in the app to establish stable client-side connectivity, it is generally retrievable from the APK. We identify two common configuration patterns for `App Keys` and apply a corresponding extraction strategy to each. The first pattern is metadata-driven configuration, where the key is stored as an XML attribute (e.g., `JPUSH_APPKEY`). In this case, we directly parse the manifest to retrieve the value. The second pattern is code-driven configuration, where the key is embedded in code and programmatically passed as an argument to push SDK initialization APIs (e.g., `PushManager.startWork(context, LOGIN_TYPE, APP_KEY)` in Baidu Cloud Push). For this pattern, we first leverage the push provider's API error-handling behavior as a lightweight signal. If the push API returns distinguishable errors for an invalid `App Key` and an invalid `Server Secret`, we use these responses to validate candidate pairings without performing deeper static tracing. Otherwise, when the API exposes only ambiguous authentication failures, we fall back to static argument tracing with LeakScope [51], a static analysis framework for tracing value flows in Android apps, to locate the `App Key` passed into push SDK initialization calls.

Validating Candidate Secrets. After pairing each candidate `Server Secret` with the correct `App Key`, the final stage performs controlled validation. Since `Server Secrets` are intended to remain strictly on the backend, any valid secret extracted from an APK indicates a confirmed misconfiguration. To validate candidates without violating ethical boundaries or disrupting real users, we use a safe probing strategy based on the API's return values.

For each verification request to the push API, we intentionally pair the candidate credentials with a `NULL` or syntactically invalid target user identifier. We then infer the secret's validity from the specific error codes returned by the server. If the secret is invalid, the request is rejected at the authentication stage and returns an authorization error (e.g., "Invalid Server Secret"). If the secret is valid, the request passes authentication but fails at the routing stage because the target identifier is intentionally malformed, returning a logical error (e.g., "User Not Found"). This distinction allows us to validate exposed secrets based solely on API behavior, without sending arbitrary content to end users. To further reduce operational impact, we rate-limit all probing requests to a conservative frequency (e.g., one request every five seconds).

4.2 Analyzing Deterministic RID Hijacking

Next, we analyze Steps 2–3 of the push notification workflow, where the client sends a registration request and the push service assigns a RID. At this step, our goal is to determine whether RID assignment depends on replayable device-side identifiers and whether such a design can enable notification rerouting in practice. To do so, we build a minimalist test app for each target push service and apply a three-stage procedure: locating the registration request, identifying the parameters that influence RID assignment, and validating rerouting through targeted spoofing.

Locating the Registration Request. The first stage is to locate the network request function that obtains the RID from the push service. However, directly searching for this request in a decompiled APK is often difficult, because the relevant registration code logic inside push SDKs is usually complex and may also be obfuscated. Instead, we start from an observation: most push SDKs store the assigned RID locally, for example in `SharedPreferences` or internal storage files, so that the app can efficiently reuse it across subsequent sessions. This provides a more tractable analytical entry point. Once the local RID storage location is identified, we manually inspect the corresponding write logic and trace the related code path backward to recover the registration request function that obtains the RID from the push service. A direct search for the RID in local storage is often ineffective, because many SDKs encrypt the RID before storing it. To address this problem, we use public SDK getter APIs (e.g., `getRegistrationId()`) as reliable entry points. By inspecting how these getter methods retrieve and decode the RID, we can locate the corresponding storage location and the associated decryption logic. We then manually identify the write operation for that storage location and follow the relevant control and data flow backward to recover the complete registration request function and its precise payload structure, even when the RID is stored in encrypted form or the SDK code is obfuscated.

Identifying RID-Determining Parameters. After locating the registration request function and recovering its payload structure, we analyze which registration parameters influence RID assignment. To do so, we first install and initialize the test app on a clean device while capturing the registration traffic, and then repeat the same process after a clean reinstallation on the same device. We compare the registration payloads from these two runs and divide the parameters into two categories. Static parameters (e.g., `android_id` or hardware serials) remain unchanged across reinstallations, whereas dynamic parameters (e.g., random nonces or resettable identifiers) vary. We then evaluate how these parameter variations affect the server's RID assignment behavior. If the server returns the same RID after reinstallation, this suggests that RID assignment is deterministically derived from static parameters. If the RID changes, the backend assignment logic is not solely determined by static identifiers. This may indicate either dependence on dynamic parameters or a more robust design in which RID assignment is not determined solely by the client payload. In the former case, security still depends on whether those dynamic parameters can be practically obtained by an attacker. Therefore, in both cases, we further assess whether the RID-determining parameters are practically obtainable, namely, whether they are exposed in observable network traffic (e.g., plaintext HTTP) or stored in a location accessible to a third-party app with standard non-root permissions.

Validating Rerouting via Spoofing. In the final stage, we validate whether spoofed identifiers can successfully reroute notifications in practice. Using the dynamic instrumentation framework

Frida [4], we hook the registration request function identified in the first stage. Based on the dependency inferred in the second stage, we perform a targeted parameter spoofing experiment: on an attacker's test device, we intercept the registration request and replace the RID-determining parameters, whether static or dynamic, with values obtained from the victim's device. If the push service returns the victim's active RID to the attacker, we then test whether downstream notifications are actually rerouted. We send a test notification through the push service provider's API. If the attacker's device receives the notification while the victim's device does not receive the same notification, we confirm that rerouting succeeds in practice. This result indicates that the push SDK design is fundamentally flawed and practically exploitable.

4.3 Verifying Plaintext Payload Exposure

Finally, we examine Step 6 of the workflow, where the app server submits message payloads through the push service provider's API. At this step, we verify whether sensitive content remains in plaintext before entering the provider infrastructure. To achieve this, we adopt a two-stage procedure that combines runtime payload interception with static inspection of client-side message processing.

Intercepting Runtime Payloads. Our primary method is to detect plaintext exposure through dynamic API hooking. We set up a controlled environment with two physical devices: a *Sender* (Device A), which manually triggers target scenarios such as chat messages or transaction alerts, and a *Receiver* (Device B), which is instrumented with Frida. Manual triggering is necessary because many target scenarios, especially chat delivery, depend on prior account setup, user interaction, and relationships between devices (e.g., adding Device B as a contact), making reliable automation difficult. On the receiver side, we hook the callback APIs of the integrated push SDKs. For standard notifications, we intercept the callbacks responsible for rendering system notifications (e.g., `onNotificationArrived`). For pass-through messages, we hook the callbacks that receive raw payload data (e.g., `onReceiveMessageData`). If the intercepted payload contains recognizable plaintext sensitive content, we conclude that the corresponding notification path lacks end-to-end protection.

Inspecting Client-Side Processing Flow. When dynamic triggering is infeasible because of complex business logic, strict triggering conditions, or account restrictions, we instead analyze the client-side message processing flow statically. The goal of this stage is to infer whether the payload was protected before it was submitted to the push service. If the app server encrypts message content before submission, the client should perform a corresponding decryption after receiving the message. Conversely, if the received payload is extracted and used directly without any decryption, this suggests that the message was sent in plaintext. Based on this reasoning, we decompile the target app (e.g., using JEB [7] or JADX [6]) and locate the implementation of the push SDK listener interfaces. Starting from the notification entry point (e.g., the developer-overridden `onMessageReceived` method), we perform forward data flow analysis on the received `message` parameter. We trace the propagation of this parameter to determine whether it goes through any decryption routine (e.g., AES- or RSA-based processing) before it is processed by the app logic or rendered to the UI. If the payload is directly extracted and processed without a preceding decryption, we likewise conclude that the corresponding notification path lacks end-to-end protection.

5 Experimental Results

This section evaluates the three identified risks from three perspectives: their prevalence in the real world, their practical exploitability, and their security impact in deployed push services. We begin by introducing the experimental setup, and then present the results on server secret misconfiguration, deterministic RID hijacking, and plaintext payload exposure.

5.1 Experimental Setup

App Dataset. To empirically assess the prevalence of the identified security risks in real-world apps, we construct a large-scale dataset of 52,748 Android apps. Our data collection, completed in May 2025, covers two major Android app markets: Google Play [1] and Huawei AppGallery [2], one of the largest markets in China. By combining these two sources, we compare apps from different Android distribution environments, where developers often rely on different push services. To ensure that our study reflects real-world impact, we collect 28,457 high-profile apps from the AndroZoo repository [12], selecting those with more than one million downloads on Google Play. To complement this dataset, we develop a custom crawler to collect 24,291 apps from Huawei AppGallery.

Target Push Services. We focus our analysis on 11 widely used push notification services, including both globally deployed services and major providers in the Chinese market: Firebase Cloud Messaging (FCM) [19], OneSignal [39], Airship [10], JPush [25], GeTui [21], Umeng [46], Alibaba Cloud Push [11], Tencent Cloud Push [45], Baidu Cloud Push [13], Huawei Push Kit [23], and MobPush [38].

We select these services based on two considerations: market prevalence and platform accessibility. To ensure that our study reflects real-world usage, we focus on services with broad market adoption. Based on keyword searches and industry insight reports, such as the SDK analysis provided by 42matters [9], services like FCM, OneSignal, and Airship consistently rank among the most widely integrated push SDKs on Google Play. Similarly, JPush, GeTui, Umeng, and major cloud-backed push services such as Baidu Cloud Push, Alibaba Cloud Push, and Tencent Cloud Push remain influential in China's third-party push service ecosystem.

For device-manufacturer push services, we include only Huawei Push Kit in our study, because access to similar services from other manufacturers is much more difficult. Although major manufacturers such as Xiaomi, Vivo, and Oppo also provide their own push services, access typically requires stricter verification, including enterprise developer accounts and business-license certification. These requirements create a practical barrier for security research. In contrast, Huawei Push Kit allow for broader accessibility, enabling us to register accounts and conduct the required validation experiments without corporate credentials.

5.2 Server Secret Misconfiguration

Applying the analysis pipeline described in Section 4.1, we identify 517 apps leaking server secrets. These affected apps account for at least 78.79 billion cumulative installs, indicating that server

secret misconfiguration remains both prevalent and high-impact in real-world push integration. The distribution of these cases differs substantially between the two analyzed app markets. Specifically, 501 affected apps come from Huawei AppGallery, whereas only 16 are found in the Google Play dataset. This disparity partly reflect differences in credential guidance and integration models across push services. Among the 11 services, only MobPush, Airship, and OneSignal explicitly warn developers not to embed server-side credentials in APKs, whereas the other services do not provide comparably explicit guidance. Nevertheless, MobPush still accounts for the largest number of credential leaks, indicating that documentation alone is insufficient. Integration design may also affect the likelihood of accidental credential exposure. For most of the affected services, backend credentials are provided as standalone secret strings that can be directly embedded in client-side source code or resource files. FCM, by contrast, authenticates backend requests using a service-account JSON file. Accidental exposure would therefore typically require developers to mistakenly bundle the backend credential file into the APK. This file-based model introduces an additional barrier to accidental credential embedding, although it does not eliminate the risk entirely.

Table 1: Distribution of leaked server secrets by marketplace and push service.

Push Service	Huawei AppGallery	Google Play	Total Leaks
MobPush	329	2	331
Umeng	88	1	89
Huawei Push Kit	34	3	37
JPush	22	0	22
GeTui	19	0	19
Baidu Cloud Push	5	4	9
Alibaba Cloud Push	4	0	4
Tencent Cloud Push	1	0	1
FCM	0	5	5
OneSignal	0	1	1
Airship	0	0	0
Total	501	16	517

Note: The summation of individual push SDK leaks in the Huawei AppGallery column (502) is slightly higher than the total count of unique affected apps (501). This discrepancy is because one single app in our dataset leaks server secrets for multiple push services simultaneously.

Table 1 summarizes the distribution of leaked server secrets across the 11 target push services. The results show that these leakage cases are highly concentrated in a small set of push services that are widely used in China. MobPush accounts for the largest number of leaks, with 331 affected apps, followed by Umeng with 89. Other major services, including Huawei Push Kit, JPush, and GeTui, also exhibit varying degrees of server secret exposure.

Although globally deployed push services appear less affected overall, they are not immune to misconfiguration. In the Google Play dataset, we identify five cases in apps using FCM and one case in OneSignal. We also observe cross-market usage of push services: for example, Baidu Cloud Push and Huawei Push Kit, which are more commonly used in China, also appear in the Google Play dataset and contribute to the detected leaks.

Practical Consequences of Leaked Server Secrets. Beyond prevalence, we examine the practical consequences of leaked server secrets. Once attackers obtain a leaked server secret, they can authenticate to the push API and gain high-privilege control over notification delivery. This allows them to specify both the notification content, including the title, body, and the URL opened when the notification is clicked, and the delivery targets, such as app users. For 9 of the 11 evaluated push services, such notifications can also be broadcast to all users of the app without requiring individual RIDs. By contrast, FCM and Huawei Push Kit do not support unrestricted broadcasting and instead require valid delivery targets, such as specific RIDs. Even so, leaked secrets in these services still enable targeted notification abuse once such targets are known.

In practice, attacker-controlled notifications can be used to spread false or misleading information at scale, impersonate trusted app messages, or send repeated spam and nuisance notifications. Such abuse can undermine user trust, damage the app's reputation, and degrade the overall user experience. In addition, control over the post-click URL creates a direct social-engineering risk. Attackers can craft persuasive notifications, such as fake prize claims, promotional offers, or urgent account alerts, to lure users into clicking. Once clicked, the notification can redirect users to arbitrary websites, thereby enabling phishing, malicious landing pages, drive-by malware downloads, or other deceptive attacks.

For several providers, the impact extends beyond notification delivery. In these cases, possession of the leaked server secret alone is sufficient to access provider-side management APIs, without requiring individual RIDs. Our verification shows that leaked secrets for Alibaba Cloud Push, Tencent Cloud Push, Baidu Cloud Push, and OneSignal allow attackers to query historical push records, enabling large-scale access to past notification logs. Likewise, JPush, Alibaba Cloud Push, Baidu Cloud Push, and OneSignal expose APIs for querying and manipulating existing tags or segments, which are commonly used to group users for targeted delivery. Attackers can first enumerate existing tag or segment names through the query APIs requiring only the leaked server secret, and then rely on the returned values to perform subsequent modification or deletion operations. Since push campaigns often rely on these attributes for audience targeting and user grouping, such tampering may disrupt user management, segmentation logic, and business operations.

The most severe case appears in Alibaba Cloud Push, where leaked push credentials can extend beyond notification operations. Unlike service-specific push credentials, the secrets used for Alibaba Cloud Push (`AccessKeyId` and `AccessKeySecret`) may also function as account-level credentials for other services in the same cloud platform. In the affected cases, we verify that these leaked credentials grant unauthorized access to additional cloud resources, specifically the **Object Storage Service (OSS)**. This finding shows that, in some deployments, a push integration misconfiguration can expose cloud resources beyond the notification channel, including data stored in the developer's OSS buckets. More broadly, it suggests that push credentials may unintentionally unlock other interconnected services in the same cloud platform.

Finally, the impact of a leaked server secret can extend beyond a single app. To understand this broader risk, we examine how far one leak can propagate across related apps maintained by the same developer. Our analysis reveals two cross-app misconfiguration patterns. The first is *Shared App Keys*, where multiple apps are configured to use the same app key and therefore share the same server

secret. Under this pattern, we identify nine additional affected apps that do not leak the secret directly but remain exposed because they share the same push configuration with another app from the same developer that leaks it, including four cases in Umeng, three in GeTui, one in Baidu Cloud Push, and one in Huawei Push Kit. The second is *Credential Reuse*, where different apps use different app keys but are configured with the same server secret. In this case, a secret leaked from one app may also authenticate requests for other apps maintained by the same developer. We observe this pattern in two apps using Baidu Cloud Push. Although this specific case does not reveal additional affected apps in our current dataset, it confirms the broader risk that a leak in one project can laterally endanger other apps maintained by the same developer.

Given the severity and practical impact of this issue, we also conducted responsible disclosure to affected app developers. Among the contacted developers, 11 acknowledged the issue and began remediation, of these, seven indicated that a fix had been deployed. These included several top-ranked consumer apps, such as leading search and video platforms, whose combined cumulative installs are close to 66.4 billion.

5.3 Deterministic RID Hijacking

Following the analysis workflow described in Section 4.2, we evaluate whether RID assignment in mainstream push SDKs can be reproduced from accessible device-side identifiers and whether such behavior enables practical notification hijacking.

Table 2: Practical feasibility of deterministic RID hijacking across push services

Push Service	Practically Hijackable	Determining Value / Why Not Practical
Baidu Cloud Push	✓	`android_id`
JPush	✓	`android_id`
Umeng	✓	`utdid`
Alibaba Cloud Push	✓	`utdid`
GeTui	✗	Depends on `UUID` and Timestamp
Airship	✗	Depends on FCM RID
Tencent Cloud Push	✗	RID Not Reproduced
Huawei Push Kit	✗	RID Not Reproduced
MobPush	✗	RID Not Reproduced
FCM	✗	RID Not Reproduced
OneSignal	✗	RID Not Reproduced

Note: ✓ indicates that deterministic RID hijacking is practically feasible for the push service in our evaluation. ✗ indicates that the attack is not practical because the determining values cannot be reused, or replay does not result in successful hijacking.

Our analysis shows that 4 of the 11 evaluated push services are susceptible to deterministic RID hijacking. As summarized in Table 2, these services assign the same RID when the same determining device-side identifiers are replayed, even after app reinstallation. Specifically, Baidu Cloud Push and JPush rely on `android_id`. When we inject a victim's `android_id` into a controlled device, the push service assigns the same RID associated with the victim. Similarly, Umeng and Alibaba Cloud Push depend on `utdid`, a widely-used third-party-generated identifier.

In contrast, the remaining seven services do not lead to practical hijacking in our experiments. GeTui depends on a `UUID` and timestamp included in the registration request. Although replaying these values can reproduce the same RID, they are generated for each request and not stored locally, which prevents practical reuse by an attacker. Airship depends on the RID assigned by FCM. In our analysis, this FCM RID is stored in the app's private `SharedPreferences`, which makes it less likely to be exposed and thus limits the practicality of replay-based hijacking. The other five services, namely Tencent Cloud Push, Huawei Push Kit, MobPush, FCM, and OneSignal, do not reproduce the same RID in our experiments, even when the observed registration payload is replayed.

The practical feasibility of deterministic RID hijacking depends on whether the determining identifiers can be obtained in realistic settings. For `android_id`, the main risk comes from network exposure. Because it is not treated as a highly sensitive identifier by many apps, it may be transmitted in plaintext in ordinary network requests, allowing a network observer to obtain it. For `utdid`, the risk comes from local exposure. Previous research has shown that `utdid` may be stored in external storage by third-party identifier SDKs, where it can introduce cross-app exposure risks [17]. Thus, a seemingly benign but malicious app can obtain it with standard non-root permissions. Taken together, these exposure paths make the attack practical, because the identifiers used to determine RID assignment can be harvested from the victim's environment.

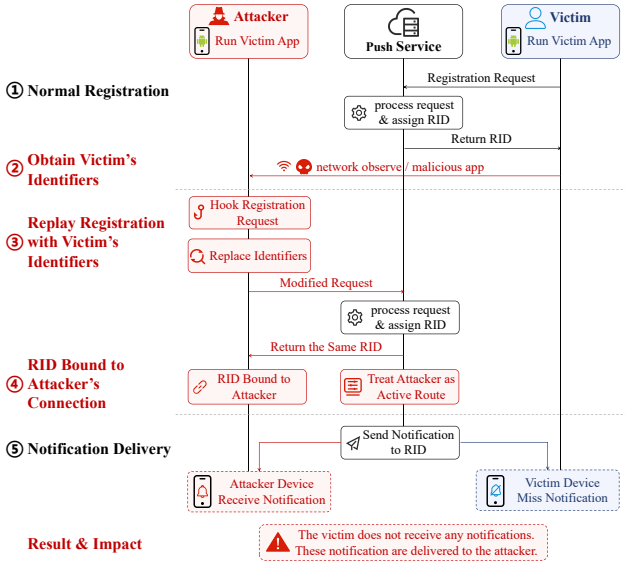

Figure 3: Attack flow of deterministic RID hijacking.

Figure 3 illustrates the attack flow of deterministic RID hijacking. Before the attack takes place, the victim runs the target app normally, and the push service processes the registration request and assigns a RID to the victim device, as shown in Step 1. The attack begins once the attacker obtains the victim's identifiers in Step 2. In Step 3, the attacker runs the victim app on an attacker-controlled device and uses a dynamic analysis tool such as Frida to hook the registration request sent by the integrated push SDK. The attacker then replaces the RID-determining parameters in that request with the identifiers obtained from the victim device and submits the modified registration request. In the affected push services, the server returns the RID originally assigned to the victim and binds

that RID to the attacker's connection (Step 4). The push service consequently treats the attacker's device as the active delivery route for that RID. When a notification is later sent to the same RID, it is forwarded according to this new binding state (Step 5). As a result, the legitimate user no longer receives subsequent notifications, and notifications intended for the victim are rerouted to the attacker's device until the victim re-registers with the push service. This causes both service disruption for the victim and direct privacy exposure, since private notification content can be disclosed to the attacker during the hijacked period.

Since this issue lies in the registration design of the push service itself, we submitted responsible disclosure reports to the affected push service providers. Following our disclosure, JPush acknowledged the issue and deployed a fix. Our retesting confirmed that replaying the same identifiers no longer reproduced the same RID.

5.4 Plaintext Payload Exposure

Applying the two-stage verification procedure outlined in Section 4.3, we evaluate whether apps protect message payloads before those payloads enter push service infrastructure. We select 110 communication-oriented apps, a category in which the confidentiality of delivered content is especially important because notifications often carry private chat messages or message previews. Specifically, for each of the 11 investigated push services, we select the top 10 communication-oriented apps, allowing our evaluation to cover both widely used apps and different push service ecosystems.

Our results show that payload protection is absent throughout the tested sample. Among the 110 apps, none (0%) protects message payloads before they are submitted to the push service provider. In all tested cases, sensitive chat content is transmitted in plaintext through the push service, indicating that plaintext payload exposure is pervasive among communication-oriented apps.

During dynamic analysis, our Frida hooks intercept the raw message payloads arriving at the device before they are processed by the app. In every tested app, when the sender transmits a chat message, the same text appears in the callback arguments received by the push SDK, such as `onNotificationArrived` or `onReceiveMessageData`. This provides direct runtime evidence that the payload remains readable when it passes through the push service. To rule out the possibility that protection is enabled only in scenarios not triggered during dynamic testing, we also perform static fallback analysis on these apps. The code inspection confirms our dynamic observations: the tested apps directly extract the payload from the received message object and pass it to UI rendering logic or local storage, and we observe no decryption routine, such as AES or RSA, on the data flow path between the SDK receiver and the app logic.

This pattern is closely related to the two message types supported by push services. For standard notifications, the submitted payload is handled directly by the push SDK for notification display, leaving the app no opportunity to process or decrypt it before display. By contrast, pass-through messages are first delivered to the app without an immediate visual alert, allowing developers to process the message, including decryption, before explicitly triggering notification display (e.g., by calling `notify()`). Because push service providers offer standard notifications as a convenient delivery path, developers often favor them over pass-through messages. Reducing plaintext exposure therefore requires changes on both sides: providers should make pass-through delivery easier to adopt through safer defaults, stronger SDK support, and clearer guidance, while developers should use pass-through messages with app-side decryption when delivering sensitive content.

As a result, sensitive contents remain visible inside provider infrastructure even when transport security is in place. Depending on provider-side operations, such payloads may also enter internal logging or analytics systems. For communication-oriented apps, this means that private chat content may be exposed beyond the intended sender and recipient.

6 Discussion

In this section, we present practical mitigation strategies, summarize lessons from fragmented push deployment, and acknowledge the limitations of our study.

6.1 Practical Mitigation Strategies

Reducing security risks in real-world push deployments requires coordinated action from both push service providers and app developers. In practice, this requires safer integration defaults, stronger credential and registration controls, as well as clearer operational guidance for secure deployment.

Strengthening Backend Credential Handling. For server secret misconfiguration, the primary mitigation is to ensure that the server secret remains strictly on the developer-controlled backend and should never be packaged into the APK. Push SDK providers can reinforce this boundary by providing build-time checks that flag server-side credentials in client-side configurations. App-market operators can further detect such misconfigurations before publication by integrating our credential-detection pipeline into the app-submission process. When a server secret has already been exposed, the first remediation step is to remove the embedded secret from the client-side code or configuration files, so that newly released app versions no longer distribute the backend credential to client device. If the push service supports secret reset, such as JPush and Baidu Cloud Push, developers should reset the exposed secret and migrate the backend to the new credential pair as soon as possible. If the provider does not offer a reset mechanism, developers may need to replace the affected service account entirely to revoke the compromised credential. Providers should also follow the principle of least privilege by ensuring that credentials used for notification delivery cannot access management APIs. As a short-term containment measure, providers can restrict API access to trusted backend IP ranges, which reduces the usability of leaked secrets without requiring immediate client updates.

Moving to Replay-Resistant Registration. For deterministic RID hijacking, the key design change is to ensure that replaying the same device-side identifiers does not lead to the same RID being assigned again. Push service providers should avoid registration designs that derive routing identities directly from identifiers such as `android_id` or `utdid`, even when those identifiers are convenient for cross-install tracking or analytics. Instead, more robust designs should incorporate server-generated randomness or session-specific state, so that replaying the same device-side identifiers alone does not reproduce the same RID. From a deployment

perspective, this change reduces the attack surface without requiring app developers to redesign their business logic. App developers should also avoid exposing device-side identifiers unnecessarily, for example by transmitting them in plaintext or storing them in accessible local storage. These measures provide a more robust baseline for registration security in real-world push deployments.

Reducing Plaintext Exposure in Notification Delivery. For plaintext payload exposure, app developers should assume that sensitive message content needs to be protected before it is submitted to a push service provider. In practice, this often requires using pass-through messages, so that developers can encrypt the message content on the server before submission and decrypt it in the app before deciding whether and how to display a notification. By contrast, standard notifications are more convenient and therefore widely used, but their content is handled directly for display, leaving the app no opportunity to process it. Push service providers can help reduce this risk by offering safer defaults, improving support for pass-through delivery, and providing clearer guidance on when plaintext message content is inappropriate. More broadly, secure push delivery requires a better balance between convenience and security, rather than treating plaintext message delivery as the default operational choice.

6.2 Lessons from Fragmented Push Deployment

Our findings suggest that the identified risks stem less from fundamentally unsolved push-security challenges than from deployment trade-offs that prioritize convenience over secure design. In fragmented Android deployment, developers often integrate multiple push SDKs to improve delivery reliability across devices and app distributions. Under these conditions, these risks can be traced to engineering choices that simplify deployment in the short term. Embedding credentials in the client simplifies deployment, deterministic registration reduces backend complexity while preserving stable identity across reinstalls, and plaintext message payloads keep provider-handled delivery convenient. From a product and deployment perspective, these choices are understandable. However, once they become the default in production systems, the resulting security and privacy risks are shifted downstream to apps and users.

Our results also highlight a recurring mismatch between developer assumptions and the actual role of push service providers. In practice, providers do more than forward messages: they authenticate privileged requests, terminate channel protection, process routing state, and often expose management APIs beyond simple notification delivery. When documentation, onboarding guidance, or integration examples do not make these security boundaries explicit, developers may over-trust the provider channel and fail to adequately protect sensitive content or privileged credentials.

6.3 Limitations

Our study has several practical limitations. First, our analysis of server secret misconfiguration relies on pattern matching, decompilation, and data flow tracing. Apps that heavily obfuscate credentials or retrieve them dynamically from backend services may evade detection, so our measurements should be interpreted as lower-bound estimates. Second, our analysis of plaintext payload exposure focuses on a targeted sample of 110 communication-oriented apps. Although this sample is sufficient to show that plaintext exposure remains common in an important high-risk category, it does not cover the full long tail of Android apps or all notification use cases. Third, our provider coverage is shaped by real-world access constraints. For example, among push services from device manufacturers, we are able to evaluate Huawei Push Kit but not several other major manufacturer services that require stronger enterprise verification or business credentials. Finally, for ethical reasons, we do not validate the full downstream impact of these risks on real users or production cloud resources. Even with these constraints, our study captures security-relevant integration patterns across a large and diverse corpus of real-world Android apps.

7 Related Work

Security and Privacy of Push Notification Ecosystems. Research on push notifications spans service evaluation, abusive use, insecure integration, and privacy leakage. Early work examined push service mechanisms and deployment trade-offs, including comparative studies of cloud messaging services [27, 47], user experience in push-notification design [20], and challenges in balancing notification delivery with background execution and battery constraints [14]. Other work studied aggressive or abusive notification behaviors, such as intrusive push advertising in Android apps [32].

A second line of work shows that push notification systems can introduce concrete security risks. Researchers have demonstrated how push services can be repurposed as command-and-control channels for Android malware [24, 26], and have explored attacks that bind virtual and real identities through mobile push notifications [35]. Security studies have also examined integration flaws in push ecosystems, including exploitable PendingIntent exposures in in Google Cloud Messaging and Amazon Device Messaging [31]. These studies reveal that push channels can become practical attack surfaces, but they typically focus on a specific provider, attack type, or stage of the notification workflow.

More recent work has highlighted privacy risks in deployed push services. Reardon et al. reported sensitive identifier and location-data collection in JPush [41]. Lou et al. conducted a broad study of vulnerabilities and privacy exposure in mobile notification systems and designed NotiLeak to detect such issues [36]. Deng et al. investigated the abuse of Android Notification Listener Service and showed that notification contents may expose sensitive information to third-party apps [16]. Samarin et al. showed that even secure-messaging apps using FCM can leak sensitive information to push providers [44].

Compared with prior work, our study expands both the scope of providers and the scope of app markets. In particular, server secret exposure has been studied before [36], but our analysis extends beyond Google Play to a more fragmented Android push ecosystem that includes Huawei AppGallery and widely used push services in China. More broadly, rather than focusing on a single provider or a single class of issues, we examine backend credential handling, registration logic, and provider-visible message delivery within one workflow-oriented security analysis.

Security and Privacy Risks of Third-Party SDKs in Mobile Apps. Third-party SDKs have long been recognized as an important source of security and privacy risk in mobile apps. Prior work has

examined SDK misuse and insecure apps across a range of functional domains. In the biometric domain, Zhang et al. studied misuse patterns in fingerprint authentication APIs [50] and analyzed the security properties of cross-side face verification systems [49]. In the payment domain, previous studies surveyed mobile payment protocols and infrastructures [15], analyzed payment service provider SDKs [37], and exposed flawed implementations of third-party in-app payment logic in Android apps [48]. Other work has focused on privacy and data-collection risks introduced by third-party components, including analytics-library leakage [34], missing runtime privacy notices [29], inconsistencies between withdrawal choices and actual third-party data collection [18], deceptive ad content [33], user-tag spoofing [30], and cross-user privacy leakage [28].

As these studies show that third-party SDK integration is a recurring source of deployment risk in mobile software. Our work builds on this line of research, but focuses on push service SDKs, where backend credentials, registration logic, and provider-visible message delivery are all part of the same deployment setting.

8 Conclusion

In this paper, we present a systematic security assessment of the Android push notification ecosystem, covering 11 mainstream push SDKs and 52,748 apps across multiple Android app markets. Through a workflow-oriented security analysis spanning device registration and message delivery, we identify three classes of risks in real-world push deployments: server secret leakage, notification hijacking, and plaintext exposure of notification payloads. These findings suggest that several security assumptions embedded in push deployments do not hold in practice. Improving push security therefore requires corresponding changes in both provider design and app integration, including keeping server secrets on the backend, adopting replay-resistant registration, and protecting sensitive notification content before it enters push service infrastructure. We responsibly disclosed the identified issues to push service providers and app developers, and received acknowledgments from affected parties.

Data Availability Statement

The publicly released artifact for this study includes analysis code, configuration files, scripts, and test apps. These materials are publicly available at a long-term archived repository with DOI: `10.5281/zenodo.19915918` and on GitHub repository [8]. The full APK dataset is not publicly redistributed because it contains third-party apps subject to copyright and access restrictions. The experimental results are also not publicly released because they contain sensitive security-related information, including details about affected apps and validation artifacts.

Acknowledgment

The full name of the authors' affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology. This work was supported in part by the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057) and the National Natural Science Foundation of China (grant No.62572209).

References

[1] 2025. Android Apps on Google Play. https://play.google.com/store/games?hl=en.
[2] 2025. HUAWEI AppGallery - HUAWEI Global. https://consumer.huawei.com/en/mobileservices/appgallery/.
[3] 2026. GitHub - androguard/androguard: Reverse engineering and pentesting for Android applications. https://github.com/androguard/androguard.
[4] 2026. GitHub - firda/fida: Frida. https://github.com/frida.
[5] 2026. GitHub - google/android-classyshark: Android and Java bytecode viewer. https://github.com/google/android-classyshark.
[6] 2026. GitHub - skylot/jadx: Dex to Java decompiler. https://github.com/skylot/jadx.
[7] 2026. JEB Decompiler by PNF Software. https://www.pnfsoftware.com/.
[8] 2026. Notification Security Research Artifacts. https://github.com/huker7/notification_security.
[9] 42matters. 2026. Top 20 Push Notifications SDKs Used in Android Apps on Google Play. https://42matters.com/sdk-analysis/top-push-notifications-sdks. Accessed: 2026-04-28.
[10] Airship. 2026. Install and Set Up the Android SDK. https://www.airship.com/docs/developer/sdk-integration/android/installation/getting-started/.
[11] Alibaba Cloud. 2026. Android SDK Integration. https://help.aliyun.com/zh/document_detail/434660.html?spm=a2c4g.11186623.0.0.70c443daAnqTLL#topic-1824033.
[12] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories*. 468–471. doi:10.1145/2901739.2903508
[13] Baidu Cloud. 2026. Android SDK Development Document. https://push.baidu.com/doc/android/api.
[14] Darshan Mohan Bidkar, Ashok Gadi Parthi, Durgaraman Maruthavanan, Balakrishna Pothineni, and Srivenkateswara Reddy Sankiti. 2024. Developing user-facing experiences in Android applications: A focus on push notifications and background operations. *DM Bidkar, AG Parthi, D. Maruthavanan, B. Pothineni, and SR Sankiti," Developing user-facing experiences in Android applications: A focus on push notifications and background operations," IJRAR* (2024). doi:10.2139/ssrn.5042159
[15] Sriramulu Bojjagani, V. N. Sastry, Chien-Ming Chen, Saru Kumari, and Muhammad Khurram Khan. 2023. Systematic survey of mobile payments, protocols, and security infrastructure. *Journal of Ambient Intelligence and Humanized Computing* 14 (2023), 609–654. doi:10.1007/s12652-021-03316-4
[16] Jiapeng Deng, Tianming Liu, Yanjie Zhao, Chao Wang, Lin Zhang, and Haoyu Wang. 2025. Walls Have Ears: Demystifying Notification Listener Usage in Android Apps. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA020 (June 2025), 23 pages. doi:10.1145/3728898
[17] Zikan Dong, Tianming Liu, Jiapeng Deng, Haoyu Wang, Li Li, Minghui Yang, Meng Wang, Guosheng Xu, and Guoai Xu. 2024. Exploring covert third-party identifiers through external storage in the android new era. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 254, 18 pages.
[18] Xiaolin Du, Zhemin Yang, Jiapeng Lin, Yinzhi Cao, and Min Yang. 2024. Withdrawing is believing? detecting inconsistencies between withdrawal choices and third-party data collections in mobile apps. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 735–751. doi:10.1109/SP54263.2024.00014
[19] Firebase Cloud Messaging. 2026. Get started with Firebase Cloud Messaging in Android apps. https://firebase.google.com/docs/cloud-messaging/android/get-started.
[20] Diana Gavilan and Gema Martinez-Navarro. 2022. Exploring user's experience of push notifications: a grounded theory approach. *Qualitative Market Research: An International Journal* 25, 2 (2022), 233–255. doi:10.1108/QMR-05-2021-0061
[21] GeTui. 2026. Android Integration Guide. https://docs.getui.com/getui/mobile/android/androidstudio/.
[22] Fu-Hau Hsu, Chuan-Sheng Wang, Yu-Liang Hsu, Yung-Pin Cheng, and Yu-Hsiang Hsneh. 2017. A client-side detection mechanism for evil twins. *Computers & Electrical Engineering* 59 (2017), 76–85. doi:10.1016/j.compeleceng.2015.10.010
[23] Huawei Push Kit. 2026. Android SDK Integration. https://developer.huawei.com/consumer/cn/doc/HMSCore-Guides/android-app-0000001071076052.
[24] Sangwon Hyun, Junsung Choi, Geumhwan Cho, and Hyoungshick Kim. 2018. Design and Analysis of Push Notification-Based Malware on Android. *Security and Communication Networks* 2018 (2018), 8510256. doi:10.1155/2018/8510256
[25] JPush. 2026. Android Quick Access. https://docs.jiguang.cn/jpush/quickstart/Android_quick.
[26] Hayoung Lee, Taeho Kang, Sangho Lee, Jong Kim, and Yoonho Kim. 2013. Punobot: Mobile botnet using push notification service in android. In *International workshop on information security applications*. Springer, 124–137. doi:10.1007/978-3-319-05149-9_8
[27] Na Li, Yanhui Du, and Guangxuan Chen. 2013. Survey of cloud messaging push notification service. In *2013 International Conference on Information Science and Cloud Computing Companion*. IEEE, 273–279. doi:10.1109/ISCC-C.2013.132

[28] S. Li, Z. Yang, N. Hua, P. Liu, X. Zhang, G. Yang, et al. 2022. Collect Responsibly But Deliver Arbitrarily? A Study on Cross-User Privacy Leakage in Mobile Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1887–1900. doi:10.1145/3548606.3559371

[29] Shuai Li, Zhemin Yang, Yuhong Nan, Shutian Yu, Qirui Zhu, and Min Yang. 2024. Are We Getting Well-informed? An In-depth Study of Runtime Privacy Notice Practice in Mobile Apps. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 1581–1595. doi:10.1145/3658644.3670377

[30] S. Li, Z. Yang, G. Yang, H. Zhang, N. Hua, Y. Huang, et al. 2023. Notice the Imposter! A Study on User Tag Spoofing Attack in Mobile Apps. In *Proceedings of the 32nd USENIX Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 307, 17 pages.

[31] T. Li, X. Zhou, L. Xing, Yeonjoon Lee, Muhammad Naveed, X. Wang, et al. 2014. 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 (CCS)*. 978–989. doi:10.1145/2660267.2660302

[32] T. Liu, H. Wang, L. Li, G. Bai, Y. Guo, and G. Xu. 2019. DaPanda: Detecting Aggressive Push Notifications in Android Apps. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 66–78. doi:10.1109/ASE.2019.00017

[33] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé Bissyandé, and Jacques Klein. 2020. Maddroid: Characterizing and detecting devious ad contents for android apps. In *Proceedings of The Web Conference 2020*. 1715–1726. doi:10.1145/3366423.3380242

[34] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang. 2019. Privacy Risk Analysis and Mitigation of Analytics Libraries in the Android Ecosystem. *IEEE Transactions on Mobile Computing* 19, 5 (2019), 1184–1199. doi:10.1109/TMC.2019.2903186

[35] Pierpaolo Loreti, Lorenzo Bracciale, and Alberto Caponi. 2018. Push attack: binding virtual and real identities using mobile push notifications. *Future Internet* 10, 2 (2018), 13. doi:10.3390/fi10020013

[36] J. Lou, X. Zhang, Y. Zhang, X. Li, X. Yuan, and N. Zhang. 2023. Devils in Your Apps: Vulnerabilities and User Privacy Exposure in Mobile Notification Systems. In *Proceedings of the 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 28–41. doi:10.1109/DSN58367.2023.00017

[37] Samin Yaseer Mahmud, K. Virgil English, Seaver Thorn, William Enck, Adam Oest, and Muhammad Saad. 2022. Analysis of Payment Service Provider SDKs in Android. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 576–590. doi:10.1145/3564625.3564641

[38] MobPush. 2026. Android SDK Integration Guide. https://www.mob.com/wiki/detailed?wiki=498&id=136.

[39] OneSignal. 2026. Android SDK setup. https://documentation.onesignal.com/docs/en/android-sdk-setup.

[40] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, Phillipa Gill, et al. 2018. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In *The 25th annual network and distributed system security symposium (NDSS 2018)*. doi:10.14722/ndss.2018.23353

[41] Joel Reardon, Nathan Good, Robert Richter, Narseo Vallina-Rodriguez, Serge Egel-man, and Quentin Palfrey. 2020. Jpush away your privacy: A case study of Jiguangs Android SDK. *International Computer Science Institute* (2020).

[42] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. doi:10.14722/ndss.2018.23143

[43] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 361–374. doi:10.1145/2906388.2906392

[44] Nikita Samarin, Alex Sullins, Trinity Chapp, B. J. Akshay Datta, Conor Gormley, Nick Mcdonald, et al. 2024. The Medium is the Message: How Secure Messaging Apps Leak Sensitive Data to Push Notification Services. arXiv preprint arXiv:2407.10589. doi:10.48550/arXiv.2407.10589

[45] Tencent Cloud. 2026. Android SDK Integration. https://cloud.tencent.com/document/product/548/36652.

[46] Umeng. 2026. Android Automatic Integration Guide. https://developer.umeng.com/docs/67966/detail/206987.

[47] Ian Warren, Andrew Meads, Satish Srirama, Thiranjith Weerasinghe, and Carlos Paniagua. 2014. Push notification mechanisms for pervasive smartphone applications. *IEEE Pervasive Computing* 13, 2 (2014), 61–71. doi:10.1109/MPRV.2014.34

[48] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, et al. 2017. 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)*. San Diego, CA, USA. doi:10.14722/ndss.2017.23091

[49] X. Zhang, H. Ye, Z. Huang, X. Ye, Y. Cao, Y. Zhuang, et al. 2023. Understanding the (In)Security of Cross-side Face Verification Systems in Mobile Apps: A System Perspective. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–950. doi:10.1109/SP46215.2023.10179474

[50] X. Zhang, X. Zhang, Z. Liu, B. Zhao, Z. Yang, and M. Yang. 2025. An Empirical Study on Fingerprint API Misuse with Lifecycle Analysis in Real-world Android Apps. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA. doi:10.14722/ndss.2025.230699

[51] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud From Mobile Apps. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. San Francisco, CA. doi:10.1109/SP.2019.00009

Received 2026-05-01; accepted 2026-07-01