Walls Have Ears: Demystifying Notification Listener Usage in Android Apps

JIAPENG DENG*, Huazhong University of Science and Technology, China
TIANMING LIU*, Monash University, Australia
YANJIE ZHAO, Huazhong University of Science and Technology, China
CHAO WANG, Huazhong University of Science and Technology, China
LIN ZHANG, National Computer Network Emergency Response Technical Team (CNCERT/CC), China
HAOYU WANG†, Huazhong University of Science and Technology, China

The Notification Listener Service (NLS) in Android allows third-party apps to monitor and process device notifications, enabling powerful features but also introducing security and privacy risks. Despite the special permission required to access NLS, it has been recurrently exploited by malicious actors. However, there is a lack of systematic investigation into NLS usage patterns and their security implications. In this paper, we propose NLRadar, a hybrid approach combining static analysis and LLM to examine NLS usage in Android apps. We apply NLRadar to a large scale of apps, including both malware and regular apps, to demystify NLS usage and to uncover abuses. Our analysis reveals that NLS is heavily abused, with interesting discoveries such as apps insecurely storing social media messages, exploiting NLS for destructive competition or SMS credential stealing, and leveraging NLS to spread promotional messages or even malicious links. We also find undisclosed changes in NLS usage through app updates and inadequate disclosure in privacy policies. Our findings emphasize the need for more rigorous vetting of NLS usage and better developer education on responsible NLS practices.

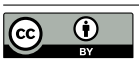
CCS Concepts: • **Security and privacy** → *Software security engineering*.

Additional Key Words and Phrases: Push Notification, Notification Listener, Static Analysis, Android Malware

ACM Reference Format:
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 (July 2025), 23 pages. https://doi.org/10.1145/3728898

*Jiapeng Deng and Tianming Liu are the co-first authors.
†Corresponding author: Haoyu Wang (haoyuwang@hust.edu.cn).

Authors' Contact Information: Jiapeng Deng, 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, Wuhan, China, jiapeng@hust.edu.cn; Tianming Liu, Monash University, Melbourne, Australia, Tianming.Liu@monash.edu; Yanjie Zhao, 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, Wuhan, China, yanjie_zhao@hust.edu.cn; Chao Wang, 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, Wuhan, China, chaowang_@hust.edu.cn; Lin Zhang, National Computer Network Emergency Response Technical Team (CNCERT/CC), Beijing, China, zhanglin@cert.org.cn; Haoyu Wang, 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, Wuhan, China, haoyuwang@hust.edu.cn.



ACM 2994-970X/2025/7-ARTISSTA020
https://doi.org/10.1145/3728898

1 Introduction

Push notifications play a crucial role in the Android system, enabling apps to keep users informed about important events, updates, and messages. Introduced in Android 4.3 (2013), the Notification Listener Service (NLS) empowers third-party apps to monitor and process notifications sent by other apps or the system itself. This service provides developers with the flexibility to create innovative features such as notification management, automation, and customization, ultimately delivering a more intelligent and personalized notification experience to users.

However, the convenience offered by the NLS also comes with significant security and privacy concerns. Once an app is granted access to the NLS, it gains the ability to continuously monitor and access all notification information generated by other apps or the system. This includes sensitive elements such as the title and content of the notifications, without requiring explicit user awareness. For instance, an app with NLS access could obtain SMS verification codes or instant messaging conversations, essentially allowing it to acquire a wealth of private and sensitive information.

To address these concerns, Android employs a special permission called `BIND_NOTIFICATION_LISTENER_SERVICE` to govern access to the NLS. In contrast to regular runtime permissions, which only necessitate user consent through a pop-up dialog, apps seeking to utilize the NLS must obtain explicit user authorization via a dedicated system settings page (Figure 1). This multi-step process aims to strike a balance between convenience and user privacy.

Despite the presence of this special permission, the NLS has been a recurring target for abuse by malicious actors. Numerous security companies have reported instances of malware exploiting the NLS to steal private information, intercept SMS credentials, and propagate malware or spam [2, 3, 5, 8]. While the research community has been aware of the potential side effects of the NLS since 2016 [28], there has been a lack of systematic investigation into how apps utilize this service and the development of tools to understand and mitigate its abuse.

To bridge this gap, we propose NLRadar to systematically investigate NLS usage and identify NLS abuse in Android apps at scale, combining static analysis and large language model (LLM) techniques. Taking an APK file as input, NLRadar examines its NLS usage on various dimensions by performing taint analysis that tracks the flow of notification-related data from NLS APIs to sensitive sinks. It then leverages LLM to identify potential NLS abuses based on static analysis results. We apply NLRadar to a large scale of Android apps, including both malware and regular apps, to demystify NLS usage and to uncover potential abuses.

Our analysis reveals several concerning NLS abuse patterns that poses security and privacy risks. We identify apps with over one million downloads that insecurely store plaintext notification content on the device, potentially exposing sensitive user data. Some apps exploit the NLS dismissal capability for destructive competition by canceling notifications from rival apps. We also discover instances where apps leverage the NLS interaction features to send promotional messages or even malicious links to users' WhatsApp contacts. We also found malware cases using NLS to steal private instant messages and SMS credentials, and canceling bank alert notifications to cover their tracks. We present a detailed characterization of how malware and regular apps use NLS, with interesting findings such as instant messaging apps and system apps are frequently monitored for their notifications, and how apps apply content-based notification filtering to facilitate their benign or malware activities. Furthermore, we found app updates often introduce undisclosed changes in NLS usage that escalate privacy risks, and privacy policies fail to properly inform users about NLS access. These findings highlight the pressing need for more rigorous analysis of NLS usage during app vetting processes and for better developer education on responsible NLS practices.

In summary, our main contributions are as follows:

- We conduct the first systematic study on NLS usage and its security implications in Android.

- We propose NLRadar, an automated approach that uses static taint analysis to characterize NLS usage on various dimensions, and uses LLM to identify potential NLS abuses for Android apps. NLRadar is open-sourced at https://github.com/security-pride/NLRadar.
- We apply NLRadar to a large scale of apps to demystify NLS usage on both malware and regular apps. Our analysis reveals concerning NLS abuse patterns with several interesting findings.

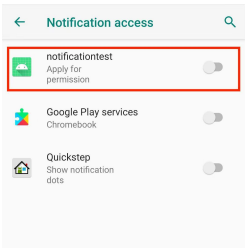
2 Background

2.1 Notification Listener Service

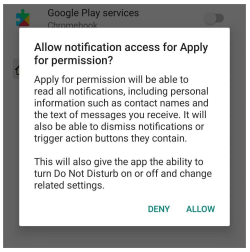
A mobile push notification is a short message from an app that displays outside of the app's UI in the device's notification drawer (as illustrated in Figure 2b) [17]. It typically contains text, multimedia content, or interactive buttons to capture user attention. Engaging with a notification—whether by tapping on its buttons or the notification itself—allows users to swiftly interact with a specific function of the app, even when the app is running in the background. Notifications serve various functions, ranging from facilitating communication (as seen in instant messaging apps) to promotional activities. They also play a role in security mechanisms, such as displaying one-time SMS codes for authentication purposes.

Notification Listener Service [16] (or **NLS** in short) is an Android system service that enables apps to monitor all notifications across the device. It was first introduced in Android 4.3 in 2013 [14], aiming to empower developers to create solutions for managing and responding to notifications. Utilizing this service, apps can obtain a spectrum of information about notifications originating from all apps, including the notification title and text, the originating app, the display style, and more. While NLS does not allow modifying notifications, it grants apps the ability to cancel notifications or automatically interact with notifications by triggering their intended post-click actions, even for notifications from other apps.

Considering the sensitive nature of NLS, Android guards its access with a special permission called `BIND_NOTIFICATION_LISTENER_SERVICE` [11] (**NLS permission** for brevity). Unlike regular runtime permissions, such as storage permission, which only require a single user confirmation via pop-up dialog, NLS access demands a two-phase authorization process through a dedicated system settings page. As shown in Figure 1a, users must manually activate NLS access for the requesting app. Then, the system presents an additional confirmation dialog (Figure 1b) that elaborates on the permission's implications, requiring explicit user consent before granting the permission.

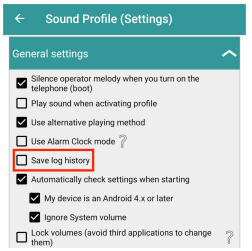


(a) Dedicated System Setting Page

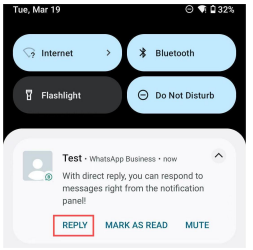


(b) Pop-up Warning

Fig. 1. Process for Users Granting NLS Permission



(a) Example A's Setting Page



(b) WhatsApp's Notification

Fig. 2. Screenshots of Motivating Examples

2.2 The Abuse of NLS

While NLS is widely adopted by apps for legitimate purposes, such as enhancing user experience through task automation and notification bar customization, it has a significant history of being exploited to facilitate malicious activities. The NLS, along with its associated permission, has been recurrently abused by malware, as reported by multiple antivirus companies [2, 3, 5, 8], to the point that this permission has gained an infamous reputation [6]. These malicious apps generally leverage NLS to steal private information and SMS credentials from notifications or to propagate malware and spam. Despite the research community's awareness of this side effect of NLS since 2016 [28], there remains a notable absence of comprehensive studies from either academia or industry examining the patterns and implications of NLS usage. A few works [34–36] have focused on NLS, but only as a potential side channel for attacks. More recently, Lou et al. [47] briefly discussed NLS abuse risks and associated privacy concerns.

As one of the most frequently used components for information transfer on mobile devices, notification contains **substantial sensitive user data**. The abuse of NLS could easily lead to a series of damaging consequences, ranging from **location information exposure**, **unauthorized surveillance of instant messaging** to **authentication compromise** (e.g., stealing SMS credentials to bypass two-factor authentication for banking apps [2, 5]). However, no prior work has systematically investigated NLS usage or proposed any form of approach to understand or mitigate NLS abuses.

3 NLS Abuse Taxonomy and Motivating Examples

To understand the landscape of NLS usage and abuse within Android apps, we conducted an extensive preliminary investigation, which encompasses a comprehensive review of existing literature (such as [28, 34–36, 47]) and security reports (such as [2, 3, 5, 6, 8, 10]), and a manual inspection of over 300 apps utilizing NLS, including both malware from the Malradar dataset [57] and regular apps from legitimate app stores. Through this extensive investigation, we identified **four types of NLS abuse** that pose security and privacy risks, along with **three key dimensions** of NLS usage crucial for investigating these abuses, as outlined in Table 1. In this section, we further motivate our work with two real-world examples from our preliminary investigation that demonstrate the security hazards of NLS abuse overlooked by previous research.

Table 1. NLS Abuse Taxonomy

Types of NLS Abuse	Key Dimensions
1. Inadvertent Exposure	A. Notification Leakage
2. Deliberate Harvesting	B. Notification Filtering
3. Unwarranted Notification Cancellation	C. Notification Operation Triggering
4. Notification Interaction with Unsolicited Crafted Content	

Example A is a volume control app with over a million downloads on Google Play, which serves its primary function of adjusting volume settings based on different scenarios and conditions. It can also adjust notification volume separately from system volume, a functionality otherwise unsupported by the Android system, which explains its request for the NLS permission. However, upon enabling a seemingly innocent configuration item "Save log history" (Figure 2a) in the app settings, this app unexpectedly **stores** the contents of all notifications on the device in plaintext on the external storage, including the originating app, the notification title, and text. While this action may not be inherently malicious, as we did not observe the app actively stealing or transmitting the notification contents off the device, such misuse of NLS introduces **a new attack vector**: these plaintext notification contents become potentially accessible to other apps with storage permission, leading to a myriad of security and privacy issues as discussed in §2.2. Although modern Android

versions (10 and above) implement Scoped Storage [7] that restricts cross-app storage access, apps declaring lower targetSDKVersions (common due to Android's extensive fragmentation [37]) are not under Scoped Storage's protection, making this attack vector remain relevant in practice.

This example illustrates the **first dimension** for inspecting NLS abuse: **Notification Leakage**. Apps with NLS access can retrieve diverse notification information, ranging from notification timing and frequency to originating app names and actual content. Improper handling of this sensitive data can lead to significant security and privacy issues. The example app demonstrates how even benign apps with NLS permission can inadvertently compromise security and privacy through poor data handling practices. More broadly, notification leakage can manifest in two distinct types of NLS abuse:

- ***Inadvertent Exposure***: Benign apps may unintentionally expose notification content due to improper data handling. This includes storing plaintext data on external storage or transmitting through insecure protocols like HTTP.
- ***Deliberate Harvesting***: Apps with malicious intent can exploit NLS access to systematically capture and exfiltrate sensitive information from notifications, including two-factor authentication codes, private messages, financial transaction alerts, or other confidential data.

Example B is a malware sample instead (package name: com.fab.wflixonline, from the Malradar dataset). This app monitors incoming notifications and specifically **filters** those originating from WhatsApp, a popular instant messaging app, as its targets. Upon detecting a WhatsApp message notification, the app attempts to **reply** to the message with malicious URLs, aiming to propagate scams and malware to the victim's WhatsApp contacts. This is achieved by exploiting the quick reply button present in WhatsApp notifications, as shown in Figure 2b. After replying with the malicious content, the app covertly **cancels** the notification to avoid user awareness.

This example illustrates **two additional dimensions** for inspecting NLS abuse:

Notification Filtering: The malware specifically targets notifications from WhatsApp by filtering based on the originating app's package name. While **filtering alone may not directly constitute a security or privacy risk**, it warrants attention as **a potential precursor** to more sophisticated attacks. Malicious apps can employ filters to target sensitive notifications from various sources, such as SMS apps, banking apps, or instant messaging platforms. Additionally, apps can implement content-based filtering, such as scanning for patterns that match common verification code formats.

Notification Operation Triggering: The example app, upon identifying WhatsApp notifications, performed two sensitive notification operations: replying with customized content and notification cancellation. As mentioned in §2.1, while NLS does not support modifying the original notification, it does allow for triggering notification operations. We identify two types of potential abuses resulting from these operations[1]:

- ***Unwarranted Notification Cancellation***: Apps may misuse the ability to dismiss notifications in ways that go beyond their intended functionality or user expectations, such as concealing their own malicious activities or removing competitors' notifications unfairly.
- ***Notification Interaction with Unsolicited Crafted Content***: Notifications may contain actionable elements that allow users to interact with apps directly without opening the corresponding app. Apps may exploit these interactive elements by injecting carefully crafted content. This can range from malicious material (such as phishing links or malware) to unsolicited commercial content (like advertisements).

[1]It is worth noting that in this work, we deliberately exclude the mere triggering of a notification's intended post-click action from our subsequent analysis of potential abuse, as we have not observed or conceived of any attack scenarios arising solely from such actions. Moreover, this feature is commonly utilized by legitimate task automation apps.

4 Approach

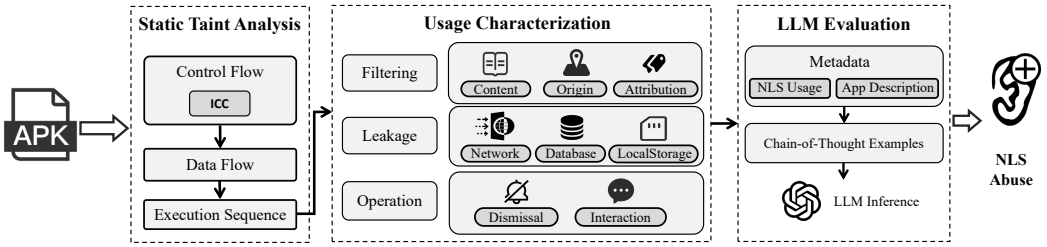


Fig. 3. Overview of NLRadar

In order to systematically investigate NLS usage and identify potential abuses in Android apps at scale, we propose NLRadar, an automated approach that utilizes both static taint analysis and large language model (LLM) techniques. Figure 3 provides an overview of NLRadar. Taking an Android app as input, NLRadar first comprehensively characterizes its NLS usage through static taint analysis. It then leverages these results to identify potential NLS abuses using LLM.

4.1 Sources and Sinks

To facilitate our taint analysis approach, we first customize taint sources and sinks related to NLS usage, as shown in Table 2. These sources and sinks are derived through a systematic investigation of standard Android APIs and sampling apps with NLS usage to inspect their source codes. We also referenced prior work that utilized Android taint analysis [33, 52, 63] for sinks.

Table 2. The Taint Sources and Sinks Used in NLRadar

Type		Class	API
Sources		NotificationListenerService	onNotificationPosted(StatusBarNotification) onNotificationRemoved(StatusBarNotification), getActiveNotifications()
Sinks	Filtering	StatusBarNotification	getPackageName(), getTag(), getNotification(), getChannelId() getPostTime(), isClearable(), isOngoing(), getUser(),getKey()
		String	equals(Object), equalsIgnoreCase(String), startsWith(*), endsWith(*) contains(CharSequence), indexOf(String), matches(String),isBlank(), isEmpty()
	Leakage	okhttp3.* java.net.* org.apache.http.*	add(String,String), post(okhttp3.RequestBody) init(URL), getOutputStream() init(String), execute(HttpUriRequest)
		java.io.* SQLiteDatabase	write(*), println(*), append(CharSequence), writeBytes(String), writeChars(String) insert(*), execSQL(java.lang.String)
	Operation	NotificationListenerService	cancelNotification(String), cancelAllNotifications()
		PendingIntent	send(*) & RemoteInput.build()

- **Taint Sources**. For sources, we focus on NLS APIs that can access notifications. Through a careful review of Android developer documents regarding NLS [16, 17, 21], as well as manually developing demo apps to interact with NLS, we enumerate three such APIs: "onNotificationPosted", "onNotificationRemoved", and "getActiveNotifications", the first two allow access to the newly posted or removed notifications on the device, while the last return a list of notifications visible to users.[2]

[2]Note that we designate all APIs that access StatusBarNotification as sources, rather than just those that specifically extract notification content (e.g., getNotification()). This approach enables us to comprehensively characterize all NLS-related behaviors, even though StatusBarNotification contains both sensitive notification content and potentially less sensitive metadata (such as ID and Post Time). Our subsequent analysis in §4.2 and §4.3.1 differentiates between these elements to accurately assess potential security and privacy risks.

- **Taint Sinks**. Our analysis focuses on sinks that are related to potential notification processing procedures of our concern, i.e., the three key dimensions for inspecting NLS abuse outlined in §3. These include various APIs such as those related to network and storage operations, accessing specific notification information, and performing notification operations. We further detail how we use these sinks to characterize NLS usage in §4.3.

4.2 Taint Analysis

We build our taint analyzer on top of the widely-adopted static analysis frameworks Soot [56] and FlowDroid [31]. Our analysis begins with generating an inter-procedural control flow graph (ICFG) of the app using FlowDroid. Noticing that apps with NLS usage often leverage inter-component communication (ICC) [42], we further incorporate ICCBot [62] a state-of-the-art tool dedicated to ICC resolution, into our analyzer. This integration produces an extended ICFG that connects components with ICC relationships. We also handle asynchronous function calls within this extended ICFG by identifying thread-starting APIs (e.g., thread.start(), AsyncTask.execute()), tracing their 'init' methods, and linking them to corresponding 'run' execution methods.

We then perform our taint analysis based on this extended ICFG, starting from the three source APIs introduced in §4.1 to propagate taint both forward and backward, following FlowDroid's standard taint analysis process [31]. For alias analysis, we employ FlowDroid's strategy using Soot's PointsToAnalysis class for backward pointer analysis on tainted fields. Following FlowDroid's approach, our taint analysis continuously tracks data flows even after reaching one sink. The analysis concludes only when the global taint state stabilizes (i.e., no new variables become tainted and all possible propagation paths have been explored) or when reaching the predefined timeout of 10 minutes (ref.§5.1). Beyond FlowDroid's standard taint process, we leverage ICCBot to precisely model data flows in Intent [27] for cross-component interactions by tracking the Intent's key in putExtra(key, value) and getExtra(key) operations on the sending and receiving sides respectively and propagating taint only when the key matches. We apply similar handling to SharedPreferences [20], Android's lightweight key-value storage mechanism.

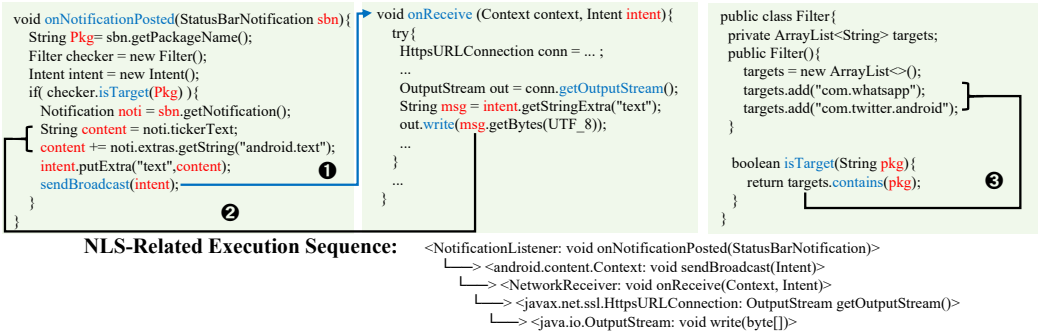


Fig. 4. A Simplified Example of our Taint Analysis Process

To illustrate our taint analysis process, Figure 4 presents a simplified code example from a malicious calculator app that exploits NLS to steal notification content from social media apps. The red variables demonstrate taint propagation from the source API onNotificationPosted's sbn parameter to the sink API write's msg parameter. In this example, the app transmits notification data through the sendBroadcast API, an ICC method. Using ICCBot, we link this broadcast to its onReceive receiver (Step ❶) and precisely track the intent's data flow by matching the "text" key at both ends. The analysis records the complete function call sequence from source to sink, as shown

in Figure 4, which captures how notification data flows through the app and serves as crucial input for our LLM evaluation module (§4.4).

Backward Analysis. To achieve a granular understanding of how apps process notifications, we further perform backward analysis on the NLS-related execution sequences. This analysis is particularly crucial in two scenarios: (1) When tainted variables flow into sink functions (e.g., data being transmitted), we trace backward to identify what specific notification data is involved. For instance, in Figure 4's Step ❷, we trace back from the sink API and identify that the 'msg' variable contains the notification's tickerText[3] and text. This distinction is crucial as these textual information contains the actual notification content, representing the most sensitive notification data, while other data such as notification timestamps are far less sensitive. (2) When filtering operations use comparison functions (e.g., *contains()*, *equals()*), we determine the concrete values being compared. As shown in Step ❸, we trace backward from the string list 'targets' that is compared with the tainted 'pkg'. This analysis reveals the app's specific notification filtering behavior, which in this case targets social media apps like WhatsApp.

Our backward analysis traces the variable of interest through the ICFG, identifying both concrete values (e.g., string constants like "com.whatsapp") and notification fields (e.g., tickerText) that contribute to the value of our interested variable. The analysis handles both intra-procedural data flows (examining local variables and class fields) and inter-procedural data flows (tracking method parameters and return values).

4.3 NLS Usage Characterization

We then detail how we use the extracted execution sequences to characterize NLS usage across the three key dimensions outlined in §3: Notification Filtering, Leakage, and Operation Triggering.

4.3.1 Notification Filtering. Apps can pick out notifications of their interests from all notifications on the device by leveraging both the notification content and diverse notification attributes provided by the system. Though notification filtering itself may not directly compromise security, it can be a crucial component in more complex malicious strategies, as evidenced by the examples in §3.

- Content Filtering: Apps can access the `Notification` object through the "getNotification()" method, enabling them to extract the notification content, including its title and text. They can then apply string filtering operations, such as keyword matching or regular expressions, using the APIs listed in Table 2 (Row String) to identify notifications of interest.
  We characterize these string filtering operations by examining the usage of these APIs within the extracted execution sequences. Whenever a match occurs between two string variables being compared at both ends of the API, we perform the aforementioned backward analysis with two main objectives: (1) to determine if the variables originate from notification contents by checking for the presence of "getNotification()" and its subsequent access to the title and text fields, and (2) to determine the specific keywords or patterns used in the filtering process, providing insights into the app's notification selection criteria.
- Originating App Filtering: Apps often utilize the package name of the originating app that sent the notification to identify notifications of interest. They can obtain this information by invoking the "getPackageName()" method on the `StatusBarNotification` object.
- Other Attributes: While Content Filtering and Originating App Filtering are the most favored, apps also use other attributes for filtering. For example, apps can invoke "isClearable()" to identify if the notification is dismissible before canceling it, or"isOngoing()" to distinguish notifications

[3]A non-visible text set by developers that provides a brief summary of the notification's content for accessibility services.

related to ongoing events such as active phone calls or music playing. They can also filter notifications based on the post time, tag, id, channel, etc.

4.3.2 Notification Leakage. Notification content may be leaked to various destinations:

- File Storage: After accessing notification data through NLS, apps can store it in files on the device's local storage using APIs from the java.io package in Table 2. As introduced in §3, notification data stored on external storage becomes readily accessible to other apps with the commonly available storage permissions when the storing app has a targetSDKVersion below Android 10, thereby raising security and privacy concerns. We further discuss external storage's exploitability in §5.3.1.
- Structured Storage: Notification content may also be persisted within structured storage mechanisms such as SharedPreferences or databases. While these storage options are generally specific to each app and not directly accessible to others, recent research [45] has shown that they could potentially be compromised if the app contains vulnerabilities that enable cross-layer exploitation attacks. Furthermore, apps could potentially record notification content in one version and later update their code to exfiltrate this information—**the fundamental principle is that apps should not record notification content unless it is necessary for their intended functionalities**.
- Network Transmission: Apps may transmit notification content to remote servers using network APIs such as "HttpURLConnection" or "OkHttp". **With malicious intent, this could constitute the most severe type of leakage.** Even with benign intentions, transmitting plaintext notification content with insecure protocols like HTTP makes it vulnerable to interception.

4.3.3 Notification Operation Triggering. While NLS does not support modifying notifications, it allows for the apps to cancel or interact with notifications.

- Notification Dismissal: Apps can programmatically dismiss notifications without user interaction. They achieve this by obtaining the notification's unique identifier using the "getKey()" method and passing it to the 'cancelNotification(key)' method.
- Automated Notification Interaction: Through NLS, apps can interact with notifications by triggering their intended post-click actions, achieving the same effect as users clicking on the notification itself or its associated buttons. To trigger the post-click action on the notification itself, the app can call the "send()" method on the notification's `contentIntent` field. For triggering post-click actions on notification buttons, the app can use "NotificationCompat.getAction()" to retrieve the action associated with each button, and then call "send()" on the action's `actionIntent`. In cases where a notification button requires customized input, apps need to construct a "RemoteInput" object with the customized input before invoking "send()". All of the above scenarios are detected by checking the extracted execution sequences related to NLS usage.

4.4 LLM-based NLS Abuse Identification

Our static taint analysis module can extract NLS-related execution sequences from the app and characterize NLS usage across the three key dimensions. However, **this characterization alone is insufficient** to fully assess whether an NLS usage is abusive or benign. For instance, consider the behavior of transmitting notification content to a remote server. This same action could be entirely legitimate for a benign messaging synchronization tool, while raising suspicions for other types of apps, such as a shopping app monitoring notifications for users' financial status, or a malicious app stealing SMS credentials. Assessing the security risks of NLS usage demands **understanding the app's intention** in utilizing NLS, which necessitates **semantic comprehension of app descriptions and code functionalities** - a problem that traditional approaches struggle to address.

To address this challenge, we turned to large language models (LLMs). Recent advancements in LLM, particularly in their capabilities for semantic understanding and logical reasoning [49, 55],

have led to their widespread application in various software engineering and security tasks [38, 61]. We devised a workflow that combines our static taint analysis results with LLM-based assessment to identify potential NLS abuses, aiming to bridge the gap between code behavior and app intentions and provide a more nuanced and context-aware assessment. This approach, while not perfect, offers a possible and promising solution in NLS abuse detection.

4.4.1 Prompt Engineering. Given the logical reasoning demands of our task, we integrate chain-of-thought reasoning [59] with few-shot learning [58] to enhance the LLM's performance. Specifically, for each NLS abuse identification query, we provide the model with carefully curated examples structured as <*input, reasoning, output*> triplets, where each example demonstrates a detailed step-by-step reasoning process. This methodology ensures that the model has immediate access to demonstrated reasoning patterns, facilitating more robust and justifiable conclusions. Table 3 illustrates our prompt engineering strategy, which consists of the following components:

Table 3. An example of our prompt engineering

Prompt Type	Instantiation
NLS Background	NLS in Android detects and retrieves device notifications...
App Description	A lightweight, user-friendly calculator for quick everyday math...
Execution Sequences	ref.Figure 4
Characterization Result	**Overview**: The NLS usage is associated with Notification Leakage, specifically through Network Transmission, where it may transmit the notification's tickerText and text. **Filtering Elements** include package names like "com.whatsapp" and "com.twitter.android". **Source Code**: ref.Figure 4
Assessment Questions	**Inadvertent Exposure:** Does the app unintentionally expose notification content by transmitting or storing it insecurely, thereby creating potential attack vectors? **Deliberate Harvesting:** Does the app misuse NLS to intentionally collect sensitive information from notifications? Evaluate: • Does the app's use of NLS align with its stated or reasonably expected functionalities? • Are there filtering processes that specifically target sensitive content (e.g., SMS credentials) or notifications from apps likely to contain sensitive information (e.g., messaging or banking apps)? • Note: The presence or absence of sensitive filtering alone is not definitive evidence of deliberate harvesting. Apps may collect all notification content without filtering for malicious purposes, while sensitive filtering could also be used for legitimate functions (e.g., message auto-reply apps). Prioritize assessing if the app's overall intention in using NLS appears benign or malicious. **Unwarranted Notification Cancellation:** Does the app exhibit unjustified dismissal behaviors? **Notification Interaction with Unsolicited Crafted Content:** Does the app unsafely trigger notification actions (such as replying to social media messages directly via notifications) with unsolicited crafted content?
CoT Reasoning Process*	Let's think step-by-step: **Step 1. Expected Behavior Analysis**: This is a calculator app. Generally speaking, a calculator app shouldn't access notifications. **Step 2. Actual Behavior Determination**: Based on the characterization result and examination of the execution sequences' source code, this app specifically targets and collects sensitive text and tickerText content from WhatsApp and Twitter notifications, then transmits them over network. **Step 3. Behavior Comparison**: This app is not supposed to access notifications, yet it ... This behavior constitutes the abuse of **Deliberate Harvesting**.
Example Output*	1.NO. The app securely transmits notification content over HTTPS protocol. 2.YES. The app collects and transmits social media notifications beyond its intended functionalities. 3. No... 4. No...
Output Requirement	Please answer each question with a YES or NO, followed by a brief explanation within 100 words.

*: Only included in few-shot learning examples

- **NLS Background**. We begin by providing the LLM with essential NLS knowledge and the three key dimensions, establishing a foundation for understanding our characterization results.
- **App Description**. We include app description sourced from the app store to enable LLM to evaluate if the observed NLS usage aligns with the app's declared functionality and intended purpose.

We then proceed to provide our static analysis results to the LLM, enabling it to form a comprehensive understanding of the app's functionalities and intentions for a thorough assessment. To illustrate, we reference the example (Figure 4) introduced in our taint analysis (§4.2), which is a malicious calculator app that exploits NLS to steal social media notifications, and include its prompt instantiation for LLM evaluation in Table 3.

- **Execution Sequences**. We begin with the NLS-related execution sequences within the apps, each of which originates from the source APIs that access notifications and terminates at sinks that could potentially lead to **the four types of NLS abuse** outlined in Section 3:
  - Network Transmission and Local Storage: these sinks could potentially constitute **Deliberate Harvesting** or **Inadvertent Exposure**.
  - Notification Dismissal: potentially constituting **Unwarranted Cancellation**.
  - Automated Notification Interaction with Customized Input: the input potentially including **Unsolicited Content**.

  If our static analysis identifies no such sequences within an app, the app is deemed as having low security risks in terms of NLS usage, and will not be forwarded to the LLM for further evaluation.
- **NLS Usage Characterization Result**. We then provide the LLM with detailed results of our characterization analysis to enhance its understanding of the app's NLS usage, structured as follows:
  - **Characterization Overview**: a concise summary of the app's NLS usage patterns across the key dimensions (see Table 3 for example).
  - **Filtering Elements**: a comprehensive list of notification filtering-related comparison strings, functions, and objects encountered during our taint analysis, to help the model understand the specific criteria and mechanisms the app uses in its notification filtering process.
  - **Source Code** of the functions within the identified execution sequences. This additional contextual information enhances the LLM's understanding of the app's intended use of NLS.
- **Assessment Questions**: To evaluate the security of NLS usage for the app, we have carefully formulated four questions, each addressing one of the four types of NLS abuse in our taxonomy. Each question is accompanied by specific evaluation criteria to guide the assessment. Due to space constraints, we present the complete question format with detailed criteria only for Deliberate Harvesting, while providing concise versions of the other three questions. The full set of questions with detailed criteria is available on our website [18].
- **Examples with chain-of-thought reasoning**: We have carefully developed five representative examples based on real-world apps with NLS usage, collectively covering all four types of NLS abuses. Each example comprises all prompt components described above, followed by an example output and a detailed step-by-step reasoning process (as shown in Table 3) demonstrating the LLM how to deduce the example output based on the given input. To ensure the clarity and comprehensiveness of this thought process, we engaged two experienced Android security researchers in a collaborative discussion during the construction of these examples. Due to space limitations, these examples are provided online [18].
- **Output Requirement**: To ensure consistent and analyzable responses, we specify a structured output format that standardizes both the decision format and explanation length.

Using this carefully constructed prompt template, we evaluate each app by providing our static analysis results alongside these examples. An affirmative response from the LLM to any assessment question indicates potential NLS abuse of a specific type is identified within the app.

5 Experimental Results

This section presents our experimental results. We begin by introducing the datasets we used throughout the experiments, comprising both a malware dataset and a regular app dataset (§5.1).

We then present our evaluation of NLRadar (§5.2), assessing the effectiveness of our static analysis module for NLS usage characterization and our LLM-based NLS abuse identification. Next, we apply NLRadar to both our malware and regular app datasets, reporting potential NLS abuses identified (§5.3), and providing detailed NLS usage characterization results (§5.4) to illuminate how NLS is used in both malicious and regular apps. Finally, we present our preliminary findings on the impact of app updates on NLS usage and the timeliness and adequacy of NLS usage disclosures in their privacy policies (§5.5).

5.1 Dataset

To characterize NLS usage in both commercial market-store apps and malware, our study utilizes two distinct datasets: **Dataset M (Malware)** and **Dataset R (Regular Apps)**. Dataset M is derived from the MalRadar dataset introduced in a recent study [57], which contains 4,534 unique Android malware samples collected from leading security companies' reports. Dataset R is sourced from the widely-adopted AndroZoo [30], a daily-updating collection of Android apps with over 20 million .APK files, a majority of which are from Google Play. As of October 2023, We randomly collected around 10% of the AndroZoo samples (2.09 million)[4] to construct Dataset R.

Table 4. Overview of Dataset Construction

Source	#Sample	#NLS Sample	Percentage
MalRadar	4,534	(Dataset M) 299	6.6%
AndroZoo Apks	2,098,861	(Dataset R Apks) 22,839	1.1%
AndroZoo Apps	1,238,139	(Dataset R Apps) 8,101	0.65%

For apps from both sources, we identified those that requested the `BIND_NOTIFICATION_LISTENER_SERVICE` permission within their Manifest files. This filtering process resulted in the formation of Dataset M (299 APKs, 6.6%) and Dataset R (22,839 APKs, 1.1%). It is worth noting that Dataset R includes different versions of .APK files for the same app. By counts of unique app package name[5], Dataset R comprises 8,101 apps (0.65%). By comparing the percentage of apps requesting the NLS permission from both sources, we observe that **malware exhibits a significantly higher tendency to request NLS permission**, suggesting that malware may be abusing NLS for malicious purposes.

Experimental Setup. Our experiments are conducted on a server running Ubuntu 22.04 with dual 64-core AMD EPYC 7713 processors and 512 GB RAM. For NLRadar's static analysis module, we set an analysis timeout of 10 minutes for apps in both datasets. Apps exceeding this limit are considered to have failed the analysis. Overall, we achieved an overall success rate of 91.7%, with an average runtime of 6 minutes per app. For LLM-based NLS abuse identification, we employed GPT-4o [13] for its superior capabilities.

5.2 Effectiveness of NLRadar

5.2.1 Static Analysis Evaluation A. As our work is the first to systematically investigate NLS usage, there are no existing datasets available for evaluating our approach. Therefore, we construct a new dataset by randomly selecting **50 apps**: 25 malware samples from Dataset M and 25 regular apps from Dataset R. We manually labeled the NLS usage of these apps in terms of the three dimensions and then compared the results against the output of NLRadar. The labeling process was carried out independently by two authors. They then convened to compare and discuss their labeling results. In cases where disputes arose, a third author was consulted to settle the disagreements.

[4]We only collect AndroZoo apps sourced from legitimate markets. The market distribution can be found on our website [15].
[5]Unless specified explicitly, when we refer to the term "app" in this Section, we mean app with a unique package name.

Although all 50 apps declared the NLS permission, only 38 (23 malware and 15 regular apps) actually accessed NLS within their codes. Among these, 34 performed filtering, 20 exhibited leakage (4 network, 4 external storage, 14 structured storage), and 22 triggered notification operations (21 cancellation, 2 interaction). The dataset and the detailed labeling results are available on our website.

In terms of identifying if NLS is used within apps, NLRadar achieves perfect results. Regarding NLS usage characterization, NLRadar yields 2 false negatives (FN) and no false positives (FP), reaching a recall of 94.7% (36/38). Both FNs are from regular apps in Dataset R, The first FN is due to the use of Java reflection, a mechanism that allows for dynamic invocation of methods by specifying them with strings, which our tool fails to handle. The second FN is a notification management app that uses Binder [22] to invoke all its notification operations. Binder is a special Android IPC mechanism working at the kernel level, which can also cause disconnections in static analysis.

Our approach demonstrates good performance on this evaluation dataset. However, it is important to note that this evaluation dataset may not cover all types of NLS usage, especially since most apps that triggered notification operations primarily focused on notification dismissal, with only two apps interacting with the notification and no apps sending their own customized input.

5.2.2 Static Analysis Evaluation B. To further assess the accuracy of NLRadar at a finer granularity, we apply our tool to both datasets. For each of the three dimensions, we randomly select 20 apps (10 malware apps and 10 regular apps) that our tool identified as having relevant operations (**60 apps** in total). We then manually check if the detection is indeed accurate.

For notification filtering, as mentioned in §4.3, we characterize filtering in three aspects: Content Filtering (17 apps), Originating App Filtering (11 apps), and Attribute-based Filtering (11 apps). Among the 20 apps, NLRadar achieves 100% accuracy in identifying these filtering aspects. However, we observed some limitations when extracting comparative objects due to dynamic behavior and complex code constructs. Firstly, our tool only analyzes hard-coded variables (e.g., constants and string arrays) and does not handle values dynamically generated at runtime or information obtained using Android APIs like getPackageName() to retrieve the app's package name. Secondly, our tool does not simulate complex string manipulations (e.g., "split()"or "replace()") to obtain the final comparative objects. We can only track the original source strings of the comparative objects as a reference, but not the results of these granular string operations.

Our tool achieved 95% accuracy in identifying notification data leakage destinations among the 20 selected apps (5 network, 11 external storage, 12 structured storage). The only exception occurred in a particular app that extends the Google Room framework [19] and customizes a database to store notification information. Our tool failed to identify this behavior because our pre-defined sinks only covered common databases. However, although our tool can accurately identify data leakage destinations, it faces challenges in precisely determining the specific notification content being stored or transmitted. This limitation arises from the difficulty in ascertaining the exact values reaching the sink through static analysis alone, as string operations or logical conditionals may modify the data during variable tracing. Nevertheless, our tool provides valuable insights by tracing the origins of the stored or transmitted data back to specific parts of the notification it potentially is related to, aiding in the assessment of potential data leakage risks.

NLRadar demonstrated 100% accuracy in identifying notification operation triggering among the 20 selected apps. However, notification dismissal still dominates the selected apps, with only 2 apps triggering notification interaction. Therefore, we further selected **30 apps** we identified with automated notification interaction: 10 for triggering the post-click action on the notification itself, 10 for triggering actions on notification buttons, and another 10 for triggering buttons with customized input. In these 30 additional apps, NLRadar still yields 100% accuracy. We make available all our datasets to boost further research in this direction.

5.2.3 LLM-based Approach Evaluation. We next evaluated our LLM-based approach for identifying the four types of NLS abuse outlined in §3. For this evaluation, we reuse the 140 apps (50+60+30) from our static analysis evaluation, as we have already manually analyzed them, making it easier for us to label abuses. As described in §4.4.1, only apps containing execution sequences terminating at sinks that could lead to NLS abuses undergo LLM-based identification. Of the 140 apps, 77 meet this criteria. Two authors then independently labeled these 77 apps for the four types of NLS abuse following the same procedure in the static analysis evaluation, and identified 44 apps with at least one type of abuse (37/40 for malware apps, 7/37 for regular apps). We also manually analyzed the remaining 63 apps (140-77) for fishy NLS usage and found no such instances.

We then compared these manual labels with the LLM output for evaluation on an app-by-app basis. An app is considered a true positive (TP) only if the LLM correctly identifies the existence or absence of all four types of NLS abuse.

Using GPT-4o, our LLM approach with chain-of-thought reasoning and few-shot learning yielded 4 FNs (1 malware, 3 regular apps) and 17 FPs (8 malware, 9 regular apps), achieving a recall of 89.2% (33/37) and precision of 66.0% (33/50). In comparison, the LLM without chain-of-thought reasoning and few-shot learning achieved a lower recall of 70.3% and precision of only 52.0%[6].

These results demonstrate that, while our LLM-based approach shows **promising ability in identifying apps with NLS abuse** (89.2% recall), its **precision is suboptimal** (66.0%). Further investigations revealed limitations in the LLM's ability to comprehend ambiguous or incomplete app descriptions. For instance, a WhatsApp helper app claiming to "clone WhatsApp messages" actually synchronizes WhatsApp notifications across devices via network transmission. LLMs struggle to connect concepts like "cloning" with "notification synchronization".

Given that determining NLS abuse is inherently challenging, such performance is somewhat expected. Fortunately, its relatively high recall makes it suitable for identifying potential NLS abuses that warrent further investigation. This approach represents our best effort to automate the detection of apps with potential ill intentions in using NLS. It also serves as a preliminary attempt to leverage LLM for security analysis based on static code analysis results. Despite its limitations, it provides initial screening of apps with potential NLS abuse and reduces the manual effort required for subsequent investigations.

5.3 Potential NLS Abuses Identified

This subsection presents the potential NLS abuses we identified after applying NLRadar to both our malware and regular app datasets. We first analyzed both dataset to extract apps containing execution sequences that could lead to NLS abuse (§4.4.1) for subsequent LLM inspection. Our analysis revealed a stark contrast between malware and regular apps, with 67.9% (203/299) of malware containing such sequences compared to only 24.0% (1,947/8,101) of regular apps. Subsequently, we employed our LLM-based approach with GPT-4o to evaluate these apps (203 malware and 1,947 regular apps) for potential NLS abuses. Table 5 summarizes the distribution of the four abuse types (outlined in §3) as identified by our LLM analysis. Our LLM-based approach significantly reduced the number of apps requiring further investigation, filtering potential abusers from 203 to 166 for malware and, more remarkably, from 1,947 to 467 (76.0% reduction) for regular apps. This dramatic decrease in manual inspection workload suggests the approach's potential for app markets' adoption to implement NLS scrutiny. To verify the effectiveness of this filtering, we manually examined all 37 malware apps (203-166) and sampled 40 regular apps not flagged by the LLM, and discovered 4 false negatives among malware and only 1 false negative among regular apps. These results,

[6]We also evaluated with the open-source DeepSeek-R1 [24] and achieved comparable performance with a recall of 82.1% and precision of 62.7%, demonstrating the accessibility of our approach.

particularly the substantial filtering rate (76.0%) and the minimal false negatives in regular apps, demonstrate the effectiveness and necessity of the LLM-based component in our approach.

To gain deeper insights into these potential NLS abuses, we randomly selected 20 apps for each abuse type for manual examination and report our findings below. Our manual analysis combined both static and dynamic approaches: we used jadx [26] to decompile the apps and examine their NLS-related execution sequences, verifying the behaviors identified by NLRadar. We then conducted dynamic analysis on a Google Pixel 3 by running each app with NLS permission granted. During a 10-minute testing period, we attempted to trigger the NLS-related behaviors (where possible, as some cannot be triggered due to dead servers, especially for malware) identified in the static code while monitoring network traffic with Fiddler [25]. Finally, we evaluated the legitimacy of each app's NLS-related behavior considering both our analysis results and the app's intended purpose. Due to space limits, code snippets of NLS abuse identified during manual analysis are available online [23].

Table 5. Distribution of Potential NLS Abuses

#	IE	DH	UC	IC	Total
Malware	36 (17.7%)	153 (75.4%)	16 (7.9%)	2 (1.0%)	166 (81.7%)
Regular	226 (11.6%)	184 (9.5%)	131 (6.7%)	36 (1.8%)	467 (24.0%)

IE: Inadvertent Exposure, DH: Deliberate Harvesting, UC: Unwarranted Notification Cancellation, IC: Notification Interaction with Unsolicited Crafted Content, see §3. One app may be associated with multiple abuses.

5.3.1 Inadvertent Exposure. A total of 36 malware and 226 regular apps were found to potentially exhibit Inadvertent Exposure behavior. Among these 226 regular apps, 51 apps have been downloaded over 100K times. Within the regular apps we manually inspected, we found an app that syncs phone notifications to PC using insecure HTTP protocol; and another app with over **100K downloads**, designed to recover deleted social media messages, **stores plaintext notification content in external storage**. However, this Google Play app targets Android versions higher than 10, as do most samples in our dataset sourced from AndroZoo[7], and therefore is protected by Scoped Storage, rendering the stored content inaccessible by other apps as introduced in §3. Google Play enforces strict requirements for apps' target API level [9], currently demanding that apps target Android 13 or higher. Nevertheless, recent research on external storage security [37] revealed that other major third-party app markets, like Samsung [4] and Huawei [1], do not maintain such strict requirements, only mandating Android 8 or 9. Additionally, Android devices running lower system versions such as smart TVs, VR headsets, and IoT devices remain vulnerable. These severe fragmentation issues [60] in the Android ecosystem make exploitation via external storage still a relevant security concern, including for the vector of notification data exposure highlighted in our study.

5.3.2 Deliberate Harvesting. We uncovered 153 malware and 184 regular apps potentially exhibiting Deliberate Harvesting, while over 34 of the regular apps have more than 100K downloads. **Within malware**, our manual analysis uncovers **cases of harvesting social media messages, banking app alerts, and SMS credentials through NLS.** For the regular apps, they primarily fall into two categories: social media helper apps and message synchronization tools. While these apps may have legitimate reasons to handle notification messages, the highly sensitive nature of these messages necessitates careful evaluation of whether their notification processing behaviors cross boundaries. For example, **a tool app for downloading videos on Instagram** was found to be **recording and storing all Instagram notification content in its database.** This behavior constitutes an excessive collection of users' private information, far beyond the app's stated purpose.

[7]Among the identified 36 malware and 226 regular apps, all malware targets SDK versions lower than 10, while only 81 regular apps target SDK versions below 10.

5.3.3 Unwarranted Notification Cancellation. Our findings indicate that 16 malware and 131 regular apps exhibit Unwarranted Notification Cancellation behaviors. **For malware**, we found cases where they **cancel banking alerts, WhatsApp notifications, or SMS notifications to cover their malicious activities**. Interestingly, within regular apps, we identified five unofficial Reddit browsing apps that exploit the notification dismissal capability for **destructive competition**, two of which have over one million downloads. These apps are designed to provide users with alternative ways to explore the Reddit platform, which offers functionalities similar to the official Reddit app but with some unique features. They promote a "Reddit notification sync" feature which merely cancels notifications from the official Reddit app without performing any actual synchronization. This practice potentially reduces user attention and engagement with the official app, thereby diverting users to their own apps. We also identified four camera and gaming apps that specifically block SMS notifications, with code patterns resembling those found in malware for hiding subscription messages. However, we were unable to reproduce the actual subscription functionality during testing due to the associated servers being inactive.

What raises greater concern is the potential security risks associated with this competitive approach. Malicious actors could exploit this technique by creating apps that impersonate official ones, **intercepting and canceling their notifications**, and **sending seemingly legitimate notifications that closely resemble those from genuine apps**. Although we have not encountered specific instances of this attack vector in our datasets, it is important to recognize that such attacks are feasible and relatively easy to implement.

5.3.4 Notification Interaction with Unsolicited Crafted Content. We found 35 regular apps and 2 malware may exhibit the abusive behavior of Notification Interaction with Unsolicited Crafted Content. The **two malware apps** exhibit the same behavior as our Motivating Example B, **abusing NLS to spread malware links through WhatsApp messages**, while most of the regular apps are social media assistant apps, which indeed have a legitimate need for automatically responding to social media notifications. However, upon our manual investigation, **we did find one case of promoting malware through NLS in the regular app dataset**. The app, com.magis.app, originated from Google Play, according to AndroZoo. It has been removed from the store by now, but we did find a snapshot of its Google Play metadata on a third-party app repository website [12]. The app had a very enticing description, portraying itself as a gift app. Additionally, we found that some auto-reply apps include promotional links when responding to instant messages. Specifically, they typically append a replying message with “sent using:” followed by a link to their apps on Google Play.

5.4 NLS Usage Characterization Results

This subsection presents the detailed characterization results of NLRadar in terms of the three NLS usage dimensions after applying it to both datasets to reveal how malicious apps and regular apps utilize NLS. Despite all of them declared the NLS permission, 95.0% apps in Dataset M and 63.1% apps in Dataset R actually utilized NLS.

5.4.1 Notification Filtering. Originating app filtering is a common practice among apps that monitor notifications. Our results reveal that **notifications from instant messaging apps and system apps are the most frequently monitored**. Top instant messaging apps monitored includes WhatsApp, Facebook, WeChat, Telegram, etc. Both our malware and regular app datasets exhibit similar percentages of monitoring instant messaging apps (13.4% versus 14.0%). For system app notifications, 11.7% (30) of apps in Dataset M specifically filter notifications from the Android SMS app (com.android.sms). While in Dataset R, 7.7% (626) of apps filter for notifications from a variety of system apps, the top three monitored apps are SMS (200 apps), Phone Calls (85), and System UI (84).

As for content-based filtering, in Dataset M, **a majority of malicious apps tend to indiscriminately obtain notification content** without applying specific filtering criteria. However, one notable example is a malicious app that utilizes a regular expression pattern (?<!\d)(\d{4,6})(?!\d) to extract of 4 to 6 digit sequences within the notification text, a pattern closely resembling the format of **verification codes**. The app then sends the extracted digit sequences to the attacker's server. In contrast, content-based filtering is more prevalent and diverse among regular apps in Dataset R. For example, 50 apps monitor for the Chinese phrase "red envelope" - a feature offered by popular Chinese instant messaging apps that enables users to send monetary gifts within digital envelopes to their contacts or group chats. By detecting this specific keyword, these apps can automatically execute scripts to claim the red envelopes. Other prevalent keywords monitored by apps include "new message" (131 apps) and "SMS" (72 apps). In addition to keyword-based filtering, 160 apps employ regular expressions to extract specific information from notifications for filtering purposes. Specifically, 31 apps use regular expression to extract specific formats of time string from notification text such as "HH:MM (08:30)", while 46 apps use '\d+' to extract numbers.

As for filtering based on other attributes, 23.4% of malware and 10.9% of regular apps use "isClearable()" to determine whether the notifications can be dismissed. while "isOngoing()" is leveraged (17.3% malware, 5% regular apps) to check for ongoing events such as phone calls and music playing.

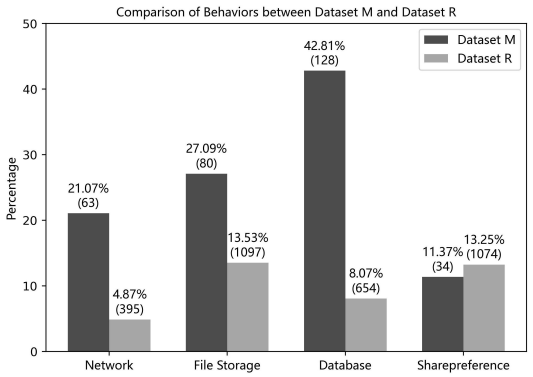


Fig. 5. Distribution of Leakage Destinations

5.4.2 Notification Leakage. As shown in Figure 5, we analyzed four potential channels for notification data exposure: file storage, databases, SharedPreferences, and network transmission. In Dataset M, a concerning 61.8% (185/299) of malicious apps exhibited potential notification leakage behavior. Notably, 21.07% (63) transmitted notification data over the network. Among the notification components potentially exposed, we observed that 52.5% (157) handled the notification content (title and text), 51.8% (155) processed the notification tickerText, and 60.8% (182) accessed the package name of the originating app. In contrast, regular apps showed a significantly lower incidence of potential notification data exposure, with only 21.8% (1,770) of apps exhibiting such behavior. The distribution across exposure channels in Dataset R was more evenly spread, with network transmission being the least common at 4.87% of apps. Regarding specific notification components, 18.8% of apps in Dataset R handled the package name, and 13.7% processed the notification content. A smaller portion also handled details like post time (4.3%), tickerText (4.6%), and ID (6.4%).

Among the leakage of all notification components to various destinations, **sending notification content (title and text) to the Internet warrants the most attention**, as it could directly compromise sensitive and private information. Our analysis reveals that **21.07% (63) of malicious apps and 2.88% (233) of regular apps** engage in this concerning behavior. Among the 63 malicious

apps, we found that 21 (33.3%) specifically target notifications from social media apps, 3 focus on SMS messages, while the remaining 39 indiscriminately steal all notification content without any filtering. In the case of regular apps, 30.0% target social media app notifications, 21.5% target notifications about ongoing calls or new messages. To better understand the motivation behind regular apps transmitting notifications, we examined their app categories and found the the majority falling under the Tools (29.2%) and Health & Fitness (22.7%) categories, followed by Communication (9.5%) and Productivity (6.5%). Tools apps are for security monitoring and social media response management. On the other hand, Health & Fitness apps mainly sync notification data to smartwatches or fitness trackers.

5.4.3 Additional Operations Triggering. Regarding notification dismissal, **52.2% (156/299) of malicious apps** in Dataset M exhibit this behavior, possibly as a way to prevent users from noticing their malicious activities. In contrast, only 13.6% (1,103) of regular apps in Dataset R demonstrate the same behavior. These apps primarily belong to the Tools (48.7%), Personalization (15.8%), and Productivity (9.9%) categories, which often include mobile management or cleaning tools offering features such as notification organization, bulk removal, or customization.

As for automated notification interaction, we only found two malware cases, both of which target WhatsApp's quick-reply button to spread malware, as demonstrated by Motivating Example B in §3. In contrast, a total of 890 apps in Dataset R interact with notifications. Among these, 664 apps exhibited the behavior of triggering the post-click action on the notification itself, while 290 apps triggered actions on notification buttons. Additionally, 119 apps trigger buttons with customized input, demonstrating auto-reply functionality.

5.5 App Updates and Disclosure Practices

In constructing the regular app dataset, we collected multiple versions of apps with the same package name. This allowed us to systematically investigate how app updates potentially alter NLS usage patterns. Our dataset comprises 3,317 apps with multiple versions, each having at least one version that requested NLS permission. We uncovered notable variations in NLS usage behavior among them. Firstly, 242 apps did not request NLS permissions in certain versions, while in later versions, they did. Secondly, 216 apps did not exhibit notification data leakage in some versions, but in later versions, they did. Lastly, in 72 apps, we witnessed a more severe escalation of privacy risks. Specifically, these apps merely stored notification data locally in certain versions, while in later versions, they further uploaded this data to the network, implying that users' privacy faced a greater risk of exposure. We attempted to investigate whether these apps make timely notifications to users in their privacy policies regarding these changes. However, our analysis was limited by the lack of historical privacy policies for each app version, as privacy policies are typically updated to reflect only the most recent version. Nevertheless, we randomly selected 20 apps with such leak escalation and checked the newest version of their privacy policies. Surprisingly, only 4 of them disclosed their notification access within their privacy policies, suggesting that user notification is not timely.

Upon analyzing these privacy policies, we discovered even more serious issues. The majority of the apps did not mention the use of notification data or the request for the NLS permission in their privacy policies, instead opting for vague statements like "user's private data." Even when a small number of privacy policies did include notification-related explanations, the descriptions were also limited, such as "captures messages from WhatsApp." However, these statements failed to inform users that granting the NLS permission allows the app to monitor all notifications sent by the system. Moreover, they did not detail the purpose, method, content, and scope of how they deal with users' notification content, which potentially constitute privacy infringements.

6 Discussion

6.1 Implication and Mitigation

Android provides NLS to enhance user experience. However, this study reveals NLS is heavily abused by malware and often misused by regular apps, introducing a variety of security and privacy issues.

For apps legitimately using NLS, the fundamental principle should be to avoid recording notification content unless it is absolutely necessary for their intended functionalities. Developers must carefully assess the need for accessing and storing notification data and implement strict security measures when handling such sensitive information. To mitigate the risks associated with NLS misuse, Google should provide clear guidelines and best practices for developers on how to use NLS securely. This includes offering detailed instructions on implementing proper encryption techniques and secure storage mechanisms to protect notification data. By educating developers and promoting secure coding practices, Google can help minimize the unintentional misuse of NLS.

In the case of deliberate or even malicious NLS abuse, users should be educated about the potential risks associated with granting the NLS permission and encouraged to exercise caution when allowing apps to access their notifications. However, the primary responsibility lies with the app stores to prevent malicious apps from reaching users in the first place. Stringent vetting processes should be implemented specifically for apps requesting the NLS permission. As a starting point, only a subset of app categories that genuinely require NLS functionality should be allowed to request this permission, while other categories like games or photo editors should be scrutinized more closely. Furthermore, app stores should employ advanced malware detection techniques and conduct thorough code analysis to identify any suspicious or malicious behavior related to NLS usage. To the best of our knowledge, no specific regulations or guidelines have been put in place to govern the use of NLS as of today. It is crucial for Google and other stakeholders in the Android ecosystem to recognize the potential risks of NLS and take proactive measures to address them.

6.2 Limitations and Future Work

Our tool suffers from common shortcomings of Android static analysis, such as overapproximation, which can lead to false positives in certain cases. Additionally, like other prior work that has utilized Android static analysis [31, 45, 63], we fail to handle Java Reflection in our approach. Fortunately, there are already dedicated works [43, 54] on taming reflection, and perhaps their approaches could be incorporated within well-adopted tools like Soot and FlowDroid to improve the coverage of static analysis. NLRadar also faces granularity issues mentioned in §5.2.2. For instance, it can only determine whether the origin of the content being leaked is related to the notification content or not, instead of precisely obtaining the actual content. These limitations present opportunities for future work and improvements in the tool's capabilities.

Our LLM-based evaluator demonstrated promising recall but fell short in precision (§5.2.3). One potential approach to improve precision could be incorporating non-abusive examples in our prompts. Considering LLM context length constraints, we currently have only included examples with abuse in our few-shot learning approach (§4.4). This design choice may have inadvertently hindered the LLM's accuracy in recognizing legitimate NLS usage. To address the false positive concerns, we conducted a preliminary experiment by adding two non-abusive examples to our prompt. This modification resulted in a 4.5% increase in precision at the cost of a 5.4% decrease in recall. These results suggest a promising direction for improving the precision-recall balance in our approach.

While NLRadar provides valuable insights for initial assessment of NLS abuse patterns, we acknowledge several limitations that prevent it from serving as a robust standalone protection mechanism. First, although we identified four types of NLS abuse through comprehensive literature review, security reports analysis, and manual app inspection, there may exist other abuse patterns

yet to be discovered. Second, determined attackers could potentially evade NLRadar's detection through various techniques: using code obfuscation and reflection to impede static analysis, employing dynamic code loading to bypass static analysis entirely, or crafting misleading app descriptions to deceive LLM-based assessment (which could potentially be mitigated by including adversarially crafted deceptive examples in the few-shot learning process). As discussed in §6.1, effectively combating NLS abuse requires a ecosystem-wide effort, and NLRadar represents one contribution to this effort, providing a foundation for understanding and detecting NLS abuse patterns.

7 Related Work

Research on Notification Security: This paper presents the first systematic study on the security implications of NLS in Android. Nevertheless, several studies have analyzed mobile notification security from other perspectives. For example, Liu *et al.* [46] developed a tool to detect aggressive push notifications that promote ads and malware in Android apps. Several works [29, 39, 41] investigated how malware abuses push notification services to facilitate malicious activities such as botnet command and control. Another line of research [32, 44, 47] focused on the vulnerabilities of widely-adopted mobile push notification services like Google's Firebase Cloud Messaging (FCM) and their security implications. More recently, two studies [51, 53] highlighted an inherent privacy concern: the use of push notification services like FCM inevitably exposes user messages to third-party entities, thereby potentially compromising user privacy. While these studies provide valuable insights into various aspects of notification security, they do not specifically address the security implications of NLS. Our work fills this critical gap by conducting an in-depth analysis of NLS usage patterns and their potential risks.

Research on the Security of Other Android System Components: One notable line of research closely related to our work is the security of Android Accessibility Services, which also is a dangerous and often exploited component. An app granted access to Accessibility Services can obtain every element on the app screen, including notifications. However, the security of Accessibility Services has been well-studied by previous research [36, 40, 50], while NLS has received comparatively less attention. Recent studies have also focused on the security of other Android system components, such as logcat [48], clipboard [45], intent [64], etc. Our work complements these existing studies by specifically investigating the security

8 Conclusion

This paper presents the first systematic study on NLS usage and its security implications in Android apps. By proposing NLRadar and applying it to a large-scale dataset, we demystify NLS usage on both malware and regular apps, and uncover concerning NLS abuse patterns that could lead to security and privacy risks. Our findings highlight the need for more rigorous analysis of NLS usage during app vetting processes and better developer education on responsible practices. Our research contributes to a better understanding of the security implications of NLS usage and lays the foundation for developing more effective strategies to detect and prevent NLS abuse.

Data Availability

Our artifact, including the implementation of NLRadar and the evaluation and experimental datasets, is available at https://github.com/security-pride/NLRadar.

Acknowledgment

We sincerely thank all the reviewers for their valuable suggestions and comments. This work was supported by the National Natural Science Foundation of China (grant No.62072046) and the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079).

References

[1] 2019. Expanding target API level requirements in 2019. https://android-developers.googleblog.com/2019/02/expanding-target-api-level-requirements.html.

[2] 2019. Malware sidesteps Google policy with new 2FA bypass technique. https://www.welivesecurity.com/2019/06/17/malware-google-permissions-2fa-bypass/.

[3] 2021. Beware — A New Wormable Android Malware Spreading Through WhatsApp. https://thehackernews.com/2021/01/beware-new-wormable-android-malware.html.

[4] 2022. Android target API level requirements. https://seller.samsungapps.com/notice/getNoticeDetail.as?csNoticeID=0000007234.

[5] 2023. Alien Android Banking Trojan Sidesteps 2FA | Threatpost. https://threatpost.com/alien-android-2fa/159517/.

[6] 2023. Promiscuous Permissions: Catching Your Android Apps in the Act | Keysight Blogs. https://www.keysight.com/blogs/en/tech/nwvs/2023/03/24/promiscuous-permissions-catching-your-android-apps-in-the-act.

[7] 2023. Scoped Storage. https://source.android.com/docs/core/storage/scoped.

[8] 2023. Sneaky DogeRAT Trojan Poses as Popular Apps, Targets Indian Android Users. https://thehackernews.com/2023/05/sneaky-dogerat-trojan-poses-as-popular.html.

[9] 2023. Target API level requirements for Google Play apps. https://support.google.com/googleplay/android-developer/answer/11926878.

[10] 2023. Technical analysis of SOVA android malware. https://muha2xmad.github.io/malware-analysis/sova/.

[11] 2024. BIND_NOTIFICATION_LISTENER_SERVICE | Manifest.permission | Android Developers. https://developer.android.com/reference/android/Manifest.permission#BIND_NOTIFICATION_LISTENER_SERVICE.

[12] 2024. Gift Offer Results APK (Android App) - Free Download. https://apkcombo.com/gift-offer-results/com.magis.app/.

[13] 2024. GPT-4o | OpenAI. https://openai.com/index/hello-gpt-4o/.

[14] 2024. Jelly Bean | Android Developers. https://developer.android.com/about/versions/jelly-bean.

[15] 2024. Market Distribution of the Regular App Dataset | NLRadar. https://github.com/security-pride/NLRadar/tree/master/ApkInfo.

[16] 2024. NotificationListenerService | Android Developers. https://developer.android.com/reference/android/service/notification/NotificationListenerService.

[17] 2024. Notifications overview | Android Developers. https://developer.android.com/develop/ui/views/notifications.

[18] 2024. Prompting Questions for Assessing NLS Usage Security and Chain-of-thought Reasoning Example | NLRadar. https://github.com/security-pride/NLRadar/tree/master/NLRadar/LLM_Evaluation.

[19] 2024. Save data in a local database using Room - Android Developers. https://developer.android.com/training/data-storage/room.

[20] 2024. SharedPreferences | Android Developers. https://developer.android.com/training/data-storage/shared-preferences.

[21] 2024. StatusBarNotification | Android Developers. https://developer.android.com/reference/android/service/notification/StatusBarNotification.

[22] 2024. Using Binder IPC | Android Open Source Project. https://source.android.com/docs/core/architecture/hidl/binder-ipc.

[23] 2025. Code Snippets of Identified NLS Abuse Examples | NLRadar. https://github.com/security-pride/NLRadar/tree/master/NLRadar/LLM_Evaluation/Abuse_Example.

[24] 2025. deepseek-ai/DeepSeek-R1. https://github.com/deepseek-ai/DeepSeek-R1.

[25] 2025. Fiddler B. https://www.telerik.com/fiddler-b.

[26] 2025. GitHub - skylot/jadx: Dex to Java decompiler. https://github.com/skylot/jadx.

[27] 2025. Intent | API reference | Android Developers. https://developer.android.com/reference/android/content/Intent.

[28] Huda Abualola, Hessa Alhawai, Maha Kadadha, Hadi Otrok, and Azzam Mourad. 2016. An Android-based Trojan Spyware to study the notificationlistener service vulnerability. *Procedia Computer Science* 83 (2016), 465–471. doi:10.1016/J.PROCS.2016.04.210

[29] Mansour Ahmadi, Battista Biggio, Steven Arzt, Davide Ariu, and Giorgio Giacinto. 2016. Detecting misuse of google cloud messaging in android badware. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. 103–112. doi:10.1145/2994459.2994469

[30] 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

[31] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269. doi:10.1145/2666356.2594299

[32] Yangyi Chen, Tongxin Li, XiaoFeng Wang, Kai Chen, and Xinhui Han. 2015. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1260–1272. doi:10.1145/2810103.2813652

[33] Yongliang Chen, Ruoqin Tang, Chaoshun Zuo, Xiaokuan Zhang, Lei Xue, Xiapu Luo, and Qingchuan Zhao. 2024. Attention! Your Copied Data is Under Monitoring: A Systematic Study of Clipboard Usage in Android Apps. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623317

[34] Kyle Denney, A Selcuk Uluagac, Kemal Akkaya, and Shekhar Bhansali. 2016. A novel storage covert channel on wearable devices using status bar notifications. In *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 845–848. doi:10.1109/CCNC.2016.7444898

[35] Kyle Denney, A Selcuk Uluagac, Hidayet Aksu, and Kemal Akkaya. 2018. An Android-Based Covert Channel Framework on Wearables Using Status Bar Notifications. *Versatile Cybersecurity* (2018), 1–17. doi:10.1007/978-3-319-97643-3_1

[36] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and XiaoFeng Wang. 2019. Kindness is a Risky Business: On the Usage of the Accessibility {APIs} in Android. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 261–275.

[37] Zikan Dong, Tianming Liu, Jiapeng Deng, 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 *33rd USENIX Security Symposium (USENIX Security 24)*. 4535–4552.

[38] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[39] Sangwon Hyun, Junsung Cho, Geumhwan Cho, and Hyoungshick Kim. 2018. Design and Analysis of Push Notification-Based Malware on Android. *Security and Communication Networks* 2018, 1 (2018), 8510256. doi:10.1155/2018/8510256

[40] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. 2014. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 103–115. doi:10.1145/2660267.2660295

[41] Hayoung Lee, Taeho Kang, Sangho Lee, Jong Kim, and Yoonho Kim. 2014. Punobot: Mobile botnet using push notification service in android. In *Information Security Applications: 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers 14*. Springer, 124–137. doi:10.1007/978-3-319-05149-9_8

[42] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291. doi:10.1109/ICSE.2015.48

[43] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Reflection-Aware Static Analysis of Android Apps. In *The 31st IEEE/ACM International Conference on Automated Software Engineering, Demo Track (ASE 2016)*. doi:10.1145/2970276.2970277

[44] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. 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*. 978–989. doi:10.1145/2660267.2660302

[45] Keke Lian, Lei Zhang, Guangliang Yang, Shuo Mao, Xinjie Wang, Yuan Zhang, and Min Yang. 2024. Component Security Ten Years Later: An Empirical Study of Cross-Layer Threats in Real-World Mobile Applications. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 70–91. doi:10.1145/3643730

[46] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. 2019. Dapanda: Detecting aggressive push notifications in android apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 66–78. doi:10.1109/ASE.2019.00017

[47] Jiadong Lou, Xiaohan Zhang, Yihe Zhang, Xinghua Li, Xu Yuan, and Ning Zhang. 2023. Devils in Your Apps: Vulnerabilities and User Privacy Exposure in Mobile Notification Systems. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 28–41. doi:10.1109/DSN58367.2023.00017

[48] Allan Lyons, Julien Gamba, Austin Shawaga, Joel Reardon, Juan Tapiador, Serge Egelman, and Narseo Vallina-Rodríguez. 2023. Log:{It's} Big,{It's} Heavy,{It's} Filled with Personal Data! Measuring the Logging of Sensitive Information in the Android Ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2115–2132.

[49] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639187

[50] Mohammad Naseri, Nataniel P Borges Jr, Andreas Zeller, and Romain Rouvoy. 2019. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. In *PETS 2019-The 19th Privacy Enhancing Technologies Symposium*. doi:10.2478/popets-2019-0031

[51] Thomas Neteler, Sascha Fahl, and Luigi Lo Iacono. 2024. "You received $100,000 from Johnny": A Mixed-Methods Study on Push Notification Security and Privacy in Android Apps. *IEEE Access* (2024). doi:10.1109/ACCESS.2024.3439095

[52] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, Vol. 14. 1125. doi:10.14722/ndss.2014.23039

[53] Nikita Samarin, Alex Sanchez, Trinity Chung, Akshay Dan Bhavish Juleemun, Conor Gilsenan, Nick Merrill, Joel Reardon, and Serge Egelman. [n. d.]. The Medium is the Message: How Secure Messaging Apps Leak Sensitive Data to Push Notification Services. *Proceedings on Privacy Enhancing Technologies* 2024, 4 ([n. d.]). doi:10.56553/popets-2024-0151

[54] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2020. Taming Reflection: An Essential Step Towards Whole-Program Analysis of Android Apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020). doi:10.1145/3440033

[55] Xiaoyu Tan, Yongxin Deng, Xihe Qiu, Weidi Xu, Chao Qu, Wei Chu, Yinghui Xu, and Yuan Qi. 2024. Thought-Like-Pro: Enhancing Reasoning of Large Language Models through Self-Driven Prolog-based Chain-of-Though. *arXiv preprint arXiv:2407.14562* (2024). doi:10.48550/arXiv.2407.14562

[56] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224. doi:10.1145/1925805.1925818

[57] Liu Wang, Haoyu Wang, Ren He, Ran Tao, Guozhu Meng, Xiapu Luo, and Xuanzhe Liu. 2022. MalRadar: Demystifying android malware in the new era. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–27. doi:10.1145/3530906

[58] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34. doi:10.1145/3386252

[59] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[60] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 226–237. doi:10.1145/2970276.2970312

[61] HanXiang Xu, ShenAo Wang, Ningke Li, Yanjie Zhao, Kai Chen, Kailong Wang, Yang Liu, Ting Yu, and HaoYu Wang. 2024. Large language models for cyber security: A systematic literature review. *arXiv preprint arXiv:2405.04760* (2024). doi:10.48550/arXiv.2405.04760

[62] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. Iccbot: fragment-aware and context-sensitive icc resolution for android applications. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 105–109. doi:10.1145/3510454.3516864

[63] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic uncovering of hidden behaviors from input validation in mobile apps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1106–1120. doi:10.1109/SP40000.2020.00072

[64] Hao Zhou, Xiapu Luo, Haoyu Wang, and Haipeng Cai. 2022. Uncovering Intent based Leak of Sensitive Data in Android Framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3239–3252. doi:10.1145/3548606.3560601

Received 2024-10-29; accepted 2025-03-31