Wear's my Data?
Understanding the Cross-Device Runtime Permission Model in Wearables

Doguhan Yeke[†], Muhammad Ibrahim[†], Güliz Seray Tuncay[‡], Habiba Farrukh[†§]
Abdullah Imran[†], Antonio Bianchi[†], and Z. Berkay Celik[†]
[†] Purdue University, {dyeke, ibrahi23, hfarrukh, imran8, antoniob, zcelik}@purdue.edu
[‡] Google, gulizseray@google.com [§] University of California, Irvine

Abstract—**Wearable devices are becoming increasingly important, helping us stay healthy and connected. There are a variety of app-based wearable platforms that can be used to manage these devices. The apps on wearable devices often work with a companion app on users' smartphones. The wearable device and the smartphone typically use two separate permission models that work synchronously to protect sensitive data. However, this design creates an opaque view of the management of permission-protected data, resulting in over-privileged data access without the user's explicit consent. In this paper, we performed the first systematic analysis of the interaction between the Android and Wear OS permission models. Our analysis is two-fold. First, through taint analysis, we showed that cross-device flows of permission-protected data happen in the wild, demonstrating that 28 apps (out of the 150 we studied) on Google Play have sensitive data flows between the wearable app and its companion app. We found that these data flows occur without the users' explicit consent, introducing the risk of violating user expectations. Second, we conducted an in-lab user study to assess users' understanding of permissions when subject to cross-device communication (n = 63). We found that 66.7% of the users are unaware of the possibility of cross-device sensitive data flows, which impairs their understanding of permissions in the context of wearable devices and puts their sensitive data at risk. We also showed that users are vulnerable to a new class of attacks that we call *cross-device permission phishing attacks* on wearable devices. Lastly, we performed a preliminary study on other watch platforms (i.e., Apple's watchOS, Fitbit, Garmin OS) and found that all these platforms suffer from similar privacy issues. As countermeasures for the potential privacy violations in cross-device apps, we suggest improvements in the system prompts and the permission model to enable users to make better-informed decisions, as well as on app markets to identify malicious cross-device data flows.**

1. Introduction

Wearable devices, such as smartwatches and fitness trackers, are becoming increasingly ubiquitous in our lives. They offer a wealth of practical and convenient features, such as making payments, sending voice messages, controlling smart devices, and monitoring and recording fitness activity. The operating systems for wearable devices can be developed either from scratch (e.g., Garmin OS, Fitbit) or based on existing mobile platforms (e.g., Wear OS from Android, watchOS from iOS). These platforms allow users to install wearable apps, which frequently communicate with their designated "companion" app running on the users' smartphones. These apps can share data back and forth to facilitate their use cases. For instance, a wearable app might use the camera of the paired device to capture photos via its companion app. We call such apps that operate on separate devices and collaborate for their functionality *cross-device apps*.

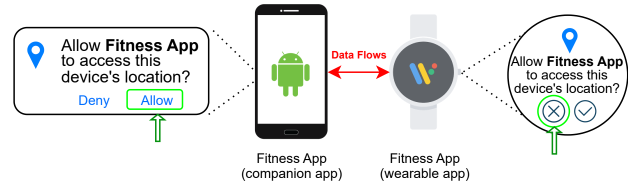


Figure 1: Illustration of two separate runtime permission models and unintended data flows on Wear OS and Android.

In the mobile-wearable ecosystem, the wearable and smartphone platforms have their own permission models that work synchronously to protect users' sensitive data, as illustrated in Figure 1. However, the interaction between a wearable app and its companion app poses unique challenges for managing user data. **In particular, the sensitive data exchange in cross-device apps that take place without the user's awareness leads to serious privacy issues.**

First, due to the complex multi-device permission model, it is difficult for users to understand the consequences of granting permission on a specific device (or both). This leads to scenarios where the user is confused regarding what data the wearable and the companion app can access. Second, this model may give users a false sense of security as they might assume that providing access to an app running on a device without internet access would mean that their sensitive data cannot be leaked online. However, the peer-to-peer connection between the devices (e.g., Bluetooth) can be used to reveal their data through the other device with an internet connection. Lastly, the permission dialog can be initiated from the wearable app to its companion app (and vice versa) using developer-managed prompts, which we call *redirection prompts*. These prompts could confuse the

users either inadvertently or with malicious intent from a developer to phish for permissions by performing what we call *cross-device permission phishing attacks*.

Prior work has rigorously examined permissions on mobile platforms through a range of studies and showed that users have many misconceptions about permissions [1], [2], [3], [4], [5], [6]. However, they have focused on apps that operate on a single device, falling short of addressing users' understanding of permissions on apps that run on multiple devices and their awareness of data flows between a wearable app and its companion app. A recent work, WearFlow [7], studied the privacy violations of the Wear OS ecosystem due to data leakages between a wearable app and a companion app. However, it does not study users' comprehension of the permission model and their mental models for access capabilities in cross-device apps.

In this paper, we focus specifically on understanding the privacy implications of the *dual permission model* adopted by Wear OS and Android to govern user data by two independent permission models. To do so, we performed the first comprehensive analysis of cross-device apps to understand their communication patterns. More specifically, we investigated if cross-device flows of permission-protected data occur in practice using a dedicated static-analysis approach. Our tool FLOWFINDER revealed 28 (out of 150) apps on Google Play with at least one data flow between the wearable app and its Android companion app.

In addition, we designed and conducted a user study (n = 63) to objectively quantify users' understanding of the Wear OS permission model when working in conjunction with Android's permission model. Our results revealed that the majority of users (66.7%) struggle with distinguishing the access capabilities of wearable apps and their companions as they are unaware of the potential data flows between them. We also demonstrated that users are susceptible to cross-device permission phishing attacks as they can be manipulated by an attacker to grant a different permission than they are initially asked for via redirection prompts. Lastly, we identified a related privacy issue in the location notice used by Wear OS during device pairing, which misleads users into believing cross-device location data transfer is impossible. We reported our findings to Google, who rewarded us with a bug bounty.

Guided by our findings, we conducted a preliminary investigation on other wearable platforms, i.e., Apple's watchOS, Fitbit, and Garmin OS. Our analysis revealed that all platforms are subject to the issues we discuss in the context of Wear OS, as they similarly adopt a dual permission model to control user data. **Our findings suggest that managing sensitive user data in the existence of data flows in cross-device apps is a challenging problem that concerns many mobile and wearable platforms**.

To mitigate the issues discussed in this paper, we suggest potential improvements to the Wear OS permission model that might help users better understand the effects of granting or denying permissions. We also recommend app markets implement strategies to identify potentially malicious or dangerous cross-device data flows.

Our contributions can be summarized as follows:

- We systematically studied the permission models of Android and Wear OS and identified privacy issues when used in the context of cross-device apps. In addition, we conducted a preliminary study on other wearable platforms and found that they also suffer from similar issues.
- We introduced FLOWFINDER, a dedicated static analysis tool, to analyze real-world wearable apps and their corresponding companion apps and show that cross-device permission-protected data flows occur in practice.
- We performed an in-lab study to assess users' understanding of the Wear OS permission model when working in conjunction with Android. Our results show that users have various misconceptions as they are unaware of the sensitive data flows between the devices. In addition, we demonstrated that users are susceptible to cross-device permission phishing.
- We found that wearable platforms fail to provide users with sufficient information to address the identified issues, and in some cases, they even provide incorrect or misleading information.
- We proposed practical countermeasures to mitigate the issues that arise from adapting a dual permission model.

Our code and data are available at https://github.com/purseclab/WearOS for public use and validation.

2. Background

Wear OS. Wear OS (formerly known as Android Wear) is primarily designed for smartwatches and provides an infrastructure for apps to run on wearable devices [8]. Wear OS apps can either be standalone (i.e., run only on the smartwatch) or can be developed as a pair of apps installed together on both the smartwatch and the phone to cooperate for their functionality [9]. We refer to the app installed on the smartwatch as *the wearable app* and the app installed on the mobile phone as *the companion app*. The wearable app can communicate with its companion app to access or control the resources on the smartphone (and vice versa).

Runtime Permission Model on Cross-device Apps. Android uses a permission model to regulate access to sensitive user data and resources. Before Android 6.0, permissions were granted during app installation [10]. With Android 6.0, the Android platform introduced the runtime permission model, which requires developers to explicitly request user consent for highly-sensitive (dangerous level) data via system-generated permission dialogs at app runtime [11]. As of September 2022, there are 41 dangerous permissions available in Android [12]. For simplicity, in this paper, we refer to them as "permissions".

We call the apps that run on multiple operating systems, such as Android and Wear OS, *cross-device apps*. On these operating systems, cross-device apps work under a *dual permission model*, where each app is subjected *only* to the permission model of the device it is running on. We refer

to this scheme with two independent permissions models as the *runtime permission model on cross-device apps*.

With this model, a permission dialog can be triggered directly by the app running on that device, or by the app running on the other device via "redirection prompts". Similar to rationale messages [13], the redirection prompts are not system-generated but instead controlled by the developer, who can provide explanations on why users need to grant permission or how their data will be accessed. Once the user grants permission, the app can access the corresponding resource on the device where the permission has been granted. While some resources, such as the microphone and location, can be available on both devices, some resources are unique to a single device. For example, the camera may only be available in the smartphone, while the bioimpedance and heart rate sensors may only be available on the smartwatch.

All permissions required by an app must be listed in the app's manifest file. The wearable app and its companion must have separate manifests, which list the required permissions for the resources on the smartwatch and the smartphone.

Wear OS-Android Communication. The communication between the mobile phone and the smartwatch requires a pairing process. To pair the mobile phone with the wearable device, the *Wear OS by Google Smartwatch* app [14] on Wear OS 2 or OEM-specific apps (e.g., *Galaxy Wearable App* [15]) on Wear OS 3 should be downloaded from the Google Play. After pairing, the apps can use local connectivity (e.g., Bluetooth) to share data or Google's Data Layer API [16], which can only be used when paired with an Android phone. Google's Data Layer API provides high-level functions such as `sendMessage` function of `MessageClient` class for the communication between devices. The Data Layer API's low-level communication is established between devices via Bluetooth, Wi-Fi, or cellular data [16].

3. Threat Model and Motivation

3.1. Threat Model

We consider an adversary whose goal is to obtain permission-protected data without the user's awareness. For this purpose, the adversary can either (1) abuse the cross-device data flows by obtaining permission on one device and leaking the data to the other device or (2) they can perform cross-device permission phishing to access the user's sensitive data by leveraging their lack of attention. We assume that the victim's smartphone and smartwatch are paired, and a connection is set up through communication channels such as Bluetooth. In addition, we assume the victim grants at least one permission on one of the devices and internet access is available on at least one device.

3.2. Cross-device Sensitive Data Flows

The runtime permission models running separately on the wearable device and the smartphone allow cross-device sensitive data flows without the users' explicit consent. To

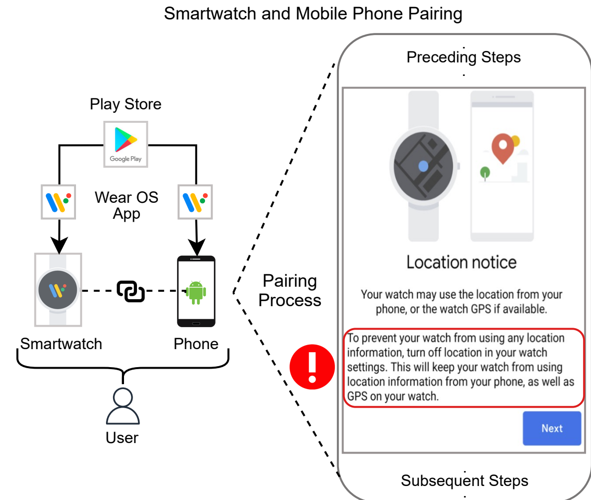


Figure 2: The location prompt shown on Wear OS 2 at the pairing process misinforms the users.

illustrate, we consider a scenario where a user installs a *Fitness Tracking* app that requires location permission on the wearable app and its companion app. The Android companion app requests the location permission to find the nearby running tracks, and the wearable app requests it to track the number of miles the user has run.

Turning to the scenario in Figure 1, the user grants the location permission on the companion app as they only want to see the nearby tracks. However, they turn off the phone's Wi-Fi and cellular data to prevent the location data from being transmitted over the Internet. Additionally, the user does not grant the location permission on the wearable app as they think their LTE-enabled watch is connected to the Internet and can transmit the location data. Hence, they expect they can search the nearby running tracks without their location leaving their smartphone. Overall, the user expects that the smartphone can access their location but cannot transmit it over the Internet, and the smartwatch cannot access their location since they denied permission to it.

The companion app, however, can send the user's location to the wearable app via the various communication methods between apps (e.g., via Bluetooth), and the wearable app can then transmit this data over the Internet. In the scenario above, the platform does not explicitly inform users about these communication methods. As we will show in Section 5, users are generally unaware that sensitive permission-protected data can be transmitted between the devices.

Misinforming Users. To make this problem worse, we found that Android misinforms the users about the possibility of such sensitive data flows while pairing the smartwatch with the smartphone. Figure 2 shows the location notice, which states that after a user revokes location permission on the smartwatch settings, the smartwatch cannot access the smartwatch's and mobile phone's location data. However, due to the *cross-device sensitive data flows*, the location data is still accessible on the smartwatch.

Upon further investigation, we found that the flawed

```
# send the request to the companion app
def requestPermissionFromPhone(input):
  # show the prompt
  text = "This app needs your Camera access."
  textView.setText(text)
  permission = Android.Manifest.READ_EXTERNAL_STORAGE
  # send the permission request to the phone
  sendMessage(nodeId, path="Mypath", permission)
  return
```

```
# handle the received request from the wearable app
def handleRequest(input):
  if input.path == "Mypath": # check the path
    if ActivityCompat.checkSelfPermission(input.receivedPerm):
      sendMessage(nodeId, path, data) # send data to watch
    else:
      # show the user the permission dialog
      startActivity(Intent, input.requestedPermission)
  return
```

Listing 1: Sender code block on the wearable app (Left), and Receiver code block on the companion app (Right).

location notice in Figure 2 exists on the pairing and device control app named "Wear OS by Google Smartwatch" used across devices on Wear OS 2. Starting with Wear OS 3, OEMs abandoned using this app and switched to developing their own apps for pairing and device control. However, a similar issue still exists in the updated location notices in the recent Samsung Galaxy Wearable app and the Pixel Watch app (See Figure 8 in Appendix A). Additionally, as we will detail in Section 5, the location toggle itself continues to be a point of confusion for users, as turning this toggle off on the watch does not prevent a wearable app from being able to retrieve the location from the companion app.

We responsibly disclosed our findings regarding the location notice to Google, who acknowledged our findings and awarded us with a bug bounty.

3.3. Cross-device Permission Phishing

When a wearable app requires permission to be granted on the companion app or vice versa, the developer shows a *redirection prompt* that asks the user to grant the permission on the paired device as shown in Figure 3.

As redirection prompts are not system-controlled, developers can customize them for malicious purposes. To perform a cross-device permission phishing attack, an adversary can exploit these redirection prompts to misguide the user and attempt to obtain permissions without their awareness. More specifically, the adversary designs a redirection prompt to intentionally misguide the user by informing them about the need for a different permission than the one that will be requested on the other device. Therefore, the user may inadvertently grant the permission without being fully aware of which permission they granted. This leads the app to have access to sensitive data without the user's informed consent.

Figure 3 demonstrates the attack steps. The user installs both the wearable app and the companion app on their respective devices and starts interacting with the wearable app. First, the user clicks the "Camera Permission on Phone" button on the wearable app's user interface (❶). This action opens a redirection prompt stating: "This app needs your Camera access". When the button "Open on phone" is clicked, it navigates the user to the companion app (❷). Lastly, the user sees the permission dialog on the phone screen and grants it since she expects that the related permission request is sent by the companion device (❸).

In this scenario, although the correct permission shown to the user should be the camera permission (`CAMERA`) based on

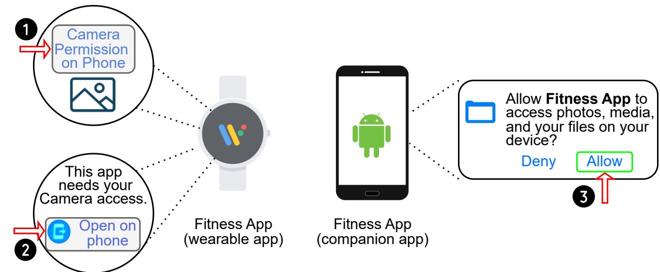


Figure 3: Demonstration of the cross-device permission phishing attacks on Wear OS 2. Notice the permission presented in step ❶ and ❷ (`Camera`) differs from the permission shown in ❸ (`External Storage`).

the redirection prompt, the permission dialog shown to the user is the storage permission (`WRITE_EXTERNAL_STORAGE` and `READ_EXTERNAL_STORAGE`). Since the redirection prompt asks for the camera permission, it is possible that the user grants the permission for storage access while thinking that they are granting the camera permission. If the user is not aware that they granted the storage permission, then we consider the permission phishing attack to have been successful, as the app now has access to permission-protected data without the user's informed consent.

Listing 1 provides a proof-of-concept (PoC) for the cross-device permission phishing scenario we described above. For this PoC, we modified the sample code provided by Android [17]. The code block on the left runs on the wearable app. It prompts the user with information about camera use (Lines 4-5) and sends a request for different permission (i.e., storage) to the companion app (Lines 6 and 8). The code block on the right runs on the companion app. Since the Activity component is optional in Android, the adversary can remove it from the companion app and show it as a redirection prompt on the wearable app.

We confirmed with Google that the app we developed could bypass Google's vetting process in the app review process. We also conducted a user study to show that users are indeed vulnerable to such permission phishing attacks.

4. Data Flow Extraction

We perform a taint analysis to show and quantify the existence of permission-protected sensitive data flows between the wearable app and its companion app. This analysis aims to demonstrate if real-world cross-device apps already

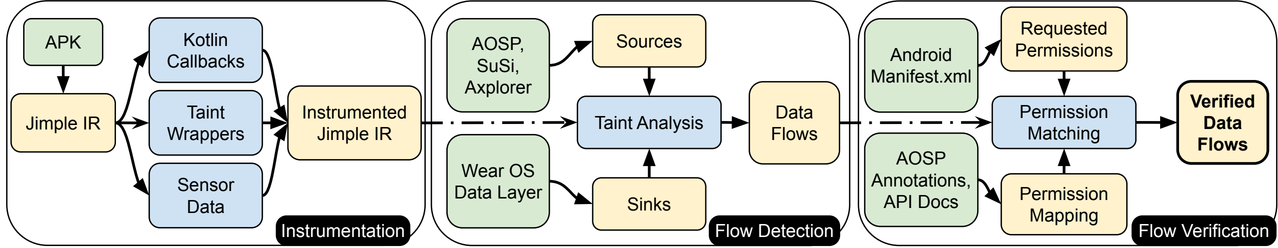


Figure 4: Overview of FLOWFINDER architecture.

share permission-protected sensitive data, which may lead to privacy concerns when users are unaware of this sharing.

Regarding cross-device apps, standard taint analysis tools have two main limitations. First, standard taint tracking tools cannot track the taints across devices to deal with both wearable and companion apps. Second, these tools use APIs as sources, but in the case of sensitive data in Wear OS, often there are no such APIs that can be directly used as sources, which is the case for sensitive data such as sensor data. To overcome these limitations, we develop our taint analysis tool, FLOWFINDER, and perform taint analysis of 150 real-world cross-device apps.

As data sources, FLOWFINDER uses any API that gives access to permission-protected data. These data sources include APIs such as `Location.getLatitude`, `TelephonyManager.getDeviceId`, and `SQLiteDatabase.query`. As data sinks, it uses any API involved in inter-device communication from the Wear OS Data Layer. These data sinks include APIs such as `DataMap.putAsset`, `MessageClient.sendMessage`, and `ChannelClient.sendFile`.

As illustrated in Figure 4, FLOWFINDER takes an APK as input and outputs the data flows from the sources to the sinks. In implementing FLOWFINDER, we address the following additional challenges, which are not handled by existing taint analysis tools:

- **Source and Sink List:** Our taint analysis requires a comprehensive list of APIs as data sources and data sinks. However, prior work does not provide a comprehensive list of such APIs. FLOWFINDER semi-automatically generates a list of APIs that are sources of permission-protected data. As sinks, it uses a comprehensive manually curated list of Wear OS Data Layer APIs that covers all possible ways to leak data between devices.
- **Complex Data Flows (False Negatives):** Real-world cross-device apps involve complex data flows that previous work on taint analysis fails to detect. For instance, data flows that originate from sources without a defined method, e.g., sensor data, and those that involve data modifications, e.g., operations on JSON objects. FLOWFINDER improves the taint analysis by instrumenting the app code to handle such data flows; specifically, we improve code reachability and taint propagation on data flows.
- **App Permissions (False Positives):** Due to the high complexity of real-world cross-device apps, taint analysis approaches over-approximate taint propagation which results in the detection of high false positive data flows. FLOWFINDER mitigates false positives by automatically discarding flows from permission-protected data sources for which an app does not have the required permissions.

4.1. FLOWFINDER Implementation

FLOWFINDER leverages existing static analysis frameworks including Soot [18] and FlowDroid [19] to perform the app analysis. Specifically, it is built on top of Soot for instrumentation and FlowDroid for the taint analysis. FlowDroid requires a list of sources and sinks as input for performing taint analysis on an APK. We provide the details of compiling source and sink lists below.

Source List Extraction. Our goal is to identify possible sensitive data sources that an app can exploit. For our analysis, we define a data source as sensitive if it is guarded by dangerous permissions. This is because they require user interaction, and we focus on evaluating the user's interaction with the devices.

To extract such sensitive data sources, we systematically analyze the Android Open Source Project (AOSP) source code. Specifically, we leverage the `@RequiresPermission` method annotation in the AOSP to identify sensitive data sources. This method annotation specifies the permission an app requires to access that method. FLOWFINDER extracts all annotated methods from the AOSP and filters the ones requiring permissions.

However, not all sensitive data sources have `@RequiresPermission` annotation. To include such data sources in our analysis, FLOWFINDER also uses data sources lists compiled by previous works [20], [21] and includes the sources that require permissions. Finally, we also manually analyzed the Android documentation to find additional sensitive data sources.

Sink List Extraction. Our goal is to identify all possible APIs in the Wear OS Data Layer used for data communication between devices. To achieve this goal, we systematically analyzed the Wear OS Data Layer, extracted such APIs, and used them as sinks in our taint analysis.

By using the generated lists of sources and sinks, FLOWFINDER performs the following steps on each APK.

Instrumentation. In this step, FLOWFINDER performs instrumentation necessary to detect sensitive Data Layer data flows. Specifically, we leverage Soot to convert app code to Jimple IR (Intermediate Representation) and instrument the Jimple IR to prepare it for taint analysis using FlowDroid.

We perform the instrumentation of the SensorEvent API in Android. Android mobile phones and most Wear OS watches have sensors to gather data (e.g., heart rate and movement speed). This sensor data is accessible to the apps by the SensorEvent API which is guarded by permissions.

The SensorEvent API is pertinent to our analysis since watch apps primarily use the sensors on the watch and can possibly leak the sensor data to the phone. Yet, we cannot use these APIs as sources directly because the SensorEvent API does not have an explicit method that can be used as a source for the taint analysis, and an app gets access to this data in a callback method (`SensorEventListener.onSensorChanged`) which is unreachable in the Soot call-graph. Thus, we need to instrument the SensorEvent data access and make the callback method reachable.

To this aim, FlowFinder leverages Soot to instrument the Android SensorEvent API. Specifically, it detects the access to SensorEvent data and instruments the data access by replacing it with an invocation to a dummy method. It then uses this dummy method as a source in the taint analysis. Furthermore, FlowFinder adds an edge to the SensorEvent callback in the `onCreate` method of the classes registering to the `onSensorChanged` callback. This makes the callback reachable during the taint analysis.

FlowFinder leverages FlowDroid to perform the taint analysis of apps. FlowDroid tries to model the Android app life-cycle by modifying the app call-graph. However, modern Android apps use callbacks that FlowDroid does not model. For example, modern apps increasingly use Kotlin [22] methods which are not handled by FlowDroid. This issue results in FlowDroid computing an incomplete call-graph that leads to false negative data flows. Since many sources and sinks can be inside these callback methods, we designed FlowFinder to detect and parse an extensive list of callback methods, including Kotlin and Data Layer API callbacks.

Furthermore, real-world apps use APIs that can fail taint propagation, e.g., due to saving data in JSON objects. FlowDroid addresses this by defining taint propagation rules using taint wrappers. However, the list of FlowDroid taint wrappers is insufficient for handling taint propagation in real-world apps since real-world apps use various APIs that FlowDroid does not address. To address this issue, FlowFinder adds an extensive list of taint wrappers necessary for detecting data flows in real-world apps.

Flow Detection. In this step, FlowFinder leverages FlowDroid to perform the taint analysis on the instrumented apps. FlowDroid requires a list of sources and sinks to perform taint analysis. FlowDroid provides a list of sinks and sources; however, that list is insufficient for detecting permission-protected data flows across devices. For this reason, we use the source and sink lists complied by FlowFinder.

Flow Verification. In this step, FlowFinder extracts the permissions requested by the app from the app's manifest file. Then, it checks if the app has requested the permissions required for accessing the data sources in the data flows detected by FlowFinder in the previous step. If the app has not requested the required permissions, we consider the data flow impossible and discard it. For example, the `getLatitude` method requires an app to have permission for the `Location`. If FlowFinder detects a flow from `getLatitude` in an app that has not requested the `Location` permission, then the detected flow is discarded.

4.2. App Collection

With FlowFinder, we explore and analyze the data flows in real-world apps in a large-scale study. Although current third-party websites (e.g., AndroZoo [23]) provide a comprehensive and structured dataset for Android apps, filtering only cross-device apps from these datasets is challenging. This is mainly because these websites do not include wearable apps, or when they do, they do not specifically categorize them as wearables. In addition, Google Play does not provide the full list of wearable apps; as of August 2022, it only provides a small subset of 104 wearable apps.

To overcome these challenges, we thus, scrape two third-party websites, Goko Store [24] and Android Wear Center [25], and the official Google Play app market to obtain the package names of all wearable apps. Overall, we find 3691 unique package names. Out of these, only 1892, including free and paid apps, have an up-to-date Google Play Store link. Since we analyze wearable apps with their companion apps, we filter this list based on whether a companion app is compatible with an Android device. This leaves us with 336 package names.

Following this, we download both the Wear OS and Android APKs for these package names. As stated in the official Google documentation [26], wearable apps that have a companion Android app can be distributed in two ways. Some wearable APKs are packaged inside their companion Android APK while others use Google's multiple-apk delivery method [27]. Previous work [7] downloaded the corresponding APKs from AndroZoo [23] and extracted the watch APK from the downloaded Android APK residing in the "`/res/raw`" path of the Android APK. However, this approach only works on a subset of apps as this method does not cater to apps using the multiple-apk delivery method.

We, therefore, follow a different approach to curate our dataset. We first use the package names to find the corresponding Google Play Store app link. We then use this link to download and install the app on an Android phone and a Wear OS watch. In this way, Google Play Store automatically installs the compatible APK for the device type. Lastly, we use the adb [28] command line tools to extract the APK from both devices. We detail the number of apps after each step in Appendix B.

From the 336 collected apps, we removed the apps having code obfuscation and are left with a total of 150 apps. We note that determining flows in obfuscated apps requires a different type of analysis (See Section 8).

4.3. Data Flow Results

Among the collected 150 apps, FlowFinder found 28 apps (with the combined number of downloads $>$ 13M) that

TABLE 1: Permission groups and the number of apps with corresponding data flows. P → W is for data being sent from phone to watch. W → P is for from watch to phone.

Permission Group	# of Apps (P → W)	# of Apps (W → P)	Total
Location	7	2	9
Sensors	0	1	1
Storage	4	0	4
Contacts	8	0	8
Physical Activity	9	1	10
Total	28	4	32

shared permission-protected sensitive data between devices. Table 1 shows an overview of the data flow directions and their corresponding permission groups. The permission groups are based on the permissions required by the data source APIs in the detected data flows. These permission groups represent the following data types:

- **Location:** Data that can reveal the user's physical location, such as latitude and longitude.
- **Sensors:** Data accessed from the onboard device sensors, such as the heart rate sensor.
- **Storage:** Files stored on the device's external storage.
- **Contacts:** User account information on the device.
- **Physical Activity:** Health data provided by the Google Fitness API [29], such as steps and calories.

Among the 32 detected data flows, 28 apps send sensitive data from the smartphone to the smartwatch. One app sends data from two permission groups (Location and Storage) to the smartwatch. We found four wearable apps that send sensitive data from the smartwatch to the smartphone. 3 apps send data (i.e., location and physical activity) in both directions. We show the details of each app in Table 2.

As shown in Table 1, most data flows (28%) occur because companion apps share physical activity data with wearable apps. All apps sending this data from the smartphone to the smartwatch are the watch face apps. Watch face apps change the wallpaper on the watch's lock screen and show information such as time, steps, and health data of the user. These apps access health data from the Google Fitness API on the user's smartphone and show it on the user's smartwatch. One reason to access the physical activity data from the smartphone rather than the smartwatch is to save battery, bandwidth, and other resources on the smartwatch. Among the detected flows from watch face apps, we found one app (`watch.richface.androidwear.valiant`) sending physical activity data in both directions; yet, we noticed no apparent use case of physical activity data on the companion app.

We note that apps targeting an SDK level lower than 29 do not need permission to access the physical activity data. Thus, they do not need to show a runtime permission dialog before accessing this data. There are only six such apps in our dataset. We note that we include all apps accessing physical activity data in our results, as newer Android versions require apps to explicitly request this permission.

Among the other detected flows, seven apps send location data from the smartphone to the smartwatch. Two apps track users' trail runs and send the user's location on the trail from the smartphone to the smartwatch. Four apps sending the smartphone's location to the smartwatch are watch face apps.

TABLE 2: The groups and directions of 32 data flows in the 28 apps detected by FlowFinder.

App Package Name	P → W	W → P
com.smartartstudios.digitalforcefree.interactive.watchface	PA†	
fr.thema.wear.watch.octane	C	
fr.thema.wear.watch.destroy	C	
com.rockgecko.dips	L	L
fr.thema.wear.watch.futuristicgui	C	
watch.richface.androidwear.valiant	PA	PA
com.bravetheskies.ghostracer	L, S	
com.watch.richface.smartdrive	PA	
de.esymetric.rungps_trial	L	L
com.thehoodiestudio.rwrk	L	
fr.thema.wear.watch.venom	C	
com.skimble.workouts		SE
com.bosenko.watchfaceblack	L	
com.smartartstudios.extremefree.interactive.watchface	PA	
com.turndapage.navmusic	S	
ga.westpoint.gaugewatchface	L	
com.thehoodiestudio.mustachewatchface	L	
fr.thema.wear.watch.feisar	C	
com.watch.richface.power	PA	
com.smartartstudios.minimusfree.interactive.watchface	PA	
fr.thema.wear.watch.rapier	C	
fr.thema.wear.watch.guardian	C	
com.watch.richface.neo	PA	
com.watch.richface.rolling	PA	
uk.co.chunkybacon.harmonywear.pro	S	
com.moletag.gallery	S	
com.smartartstudios.hexane.interactive.watchface	PA	
fr.thema.wear.watch.master	C	

† PA: Physical Activity, C: Contacts, L: Location, S: Storage, SE: Sensors

The apps sending contact information from the smartphone to the smartwatch are also watch face apps. These watch face apps send information about all the Google Accounts registered on the smartphone. This information includes account email and user name, which are considered sensitive information guarded by permissions. One app sends data from the smartwatch's onboard sensors, such as heart rate, to the smartphone.

To contextualize the discovered data flows, we manually checked the data flows of 28 apps. We discovered that some of these data flows occur due to the apps' capabilities and the device's functionality. For instance, watch face apps are not explicitly listed on the smartwatch menu; thus, users cannot open the wearable app. Therefore, to request permission from users, the watch face apps ask for location permission on the smartphone and transmit this data to the smartwatch. Two watch faces that transmit the smartphone's location to the smartwatch use the location to calculate sunrise and sunset times. The other two watch faces display weather data, which could potentially justify the use of location data. Similarly, as an example of how device capability affects the data flows, a fitness app (`com.skimble.workouts`) requests sensor permission on the smartwatch since the sensor data (e.g., heart rate) is available on the smartwatch. The developers then transmit this data from the smartwatch to the smartphone to display the user's health-related data. These examples demonstrate that app capabilities and device functionalities results in data flows between devices.

There are also pairs of wearable and companion apps that do not transfer permission-protected data across devices. One example of such an app (`com.mydiabetes`) is designed

to monitor insulin injections for diabetes patients. The app sends logs of insulin injections across devices. Since the user manually inputs this data, it is not permission protected and not considered a sensitive data flow under our analysis. In this case, the companion app requests permission-protected data, including location and contacts. This data only provides functionality on the companion app, so there is no need to send data to the wearable app. Another example is an app (`com.opl.transitnow`) used for locating public transport such as buses. The companion app uses the location data to get to nearby bus stops. The schedules of incoming buses and nearby bus stops are sent to the wearable app.

To determine if these 28 apps inform users about data transfers, we conducted an analysis of their app descriptions on the Google Play Store and the privacy policies provided by the developers. Only two apps mentioned data transfers across devices in their app descriptions. One of these apps (`com.moletag.gallery`) is used for syncing images and videos, while the other (`com.turndapage.navmusic`) is for syncing music across devices.

In the privacy policies, only one app explicitly stated which data is transferred and provided reasons for the transfers (`com.bosenko.watchfaceblack`). Ten apps mentioned that user data is collected to provide app functionality, but did not clarify if data is transferred across devices. 12 apps' privacy policies denied any data transfers across devices and assured the data remains on the device. Additionally, five apps either did not disclose how they use data in their privacy policies or did not have a privacy policy. Our analysis shows that apps do not adequately inform users about data transfers across devices, potentially posing privacy concerns.

Summary of Results. FlowFinder's app analysis results show that both wearable and companion apps share permission-protected data with their respective counterparts. The type of permission-protected data being shared includes location, sensor, health data, account information, and data stored in the device's external storage. This problem poses serious privacy concerns since users are generally unaware of the possibility of these cross-device data flows, as demonstrated by our user study in Section 5.

4.4. Evaluation of FlowFinder

We evaluate FlowFinder on 40 apps and compare our results with an existing tool called WearFlow [7], which also performs cross-device app taint analysis. The evaluated apps also include the WearBench [30] dataset created by the WearFlow authors. The complete list of evaluated apps is available in Table 7 in Appendix B. FlowFinder detects all flows correctly on the WearBench apps and yields 37% less false negatives than WearFlow. For brevity, we detail below a subset, shown in Table 3, of the evaluated apps.

For our first evaluation, we develop a set of five benchmark apps. These apps have cross-device app data flows originating from various permission-protected sources. FlowFinder can correctly detect all the flows in the benchmark apps without any false positives or negatives. On the contrary, WearFlow does not detect any data flows.

TABLE 3: Data flow comparison of FlowFinder against WearFlow for selected apps. ● means a flow is found, ○ means no flow is found.

App/Package Name	WearFlow	FlowFinder
Benchmark Apps*		
Location	○	●
DataNetwork	○	●
NetworkType	○	●
DownloadedFile	○	●
AuthToken	○	●
Real-World Apps with Flows		
com.sparkistic.photowear	○	○
com.estela	○	●
com.ammarptn.willow.digital.watch.face	○	○
com.moletag.gallery	●	●
com.skimble.workouts	○	●
Real-World Apps with No Flows		
com.sousoum.droneswear	○	○
com.appfour.wearweather	○	○
com.kjsk.watchface.jesus	○	○
ch.sooon.hub	○	○
huskydev.android.watchface.atlas	○	○

* Synthetic apps developed by the authors for evaluation.

For false positive evaluation, we analyze real-world apps with no data flows. To find apps without data flows, we manually check four properties: (1) we analyze the apps' manifest file to see if the app is a non-standalone app, (2) if there is any permission listed in the manifest file, (3) if the app is using the relevant permission-protected API and Wear OS Data Layer APIs, and (4) if the app is not sending the permission-protected data across devices. Out of the candidate apps, we randomly select five apps. We then run both FlowFinder and WearFlow on these apps; both tools report zero false positives.

To further compare FlowFinder with WearFlow, we also run WearFlow on our dataset of 150 real-world apps. WearFlow is able to identify the permission-protected sensitive data flow in only one of these apps. In contrast, FlowFinder detects the flow in three out of these five apps with permission-protected sensitive data flows.

WearFlow is not able to detect flows due to its following three main technical limitations: (1) *Insufficient Sources/Sinks:* WearFlow does not use a comprehensive list of sources and sinks (Section 4.1) required for detecting permission-protected data flows in cross-device apps. (2) *Broken Taint Propagation:* WearFlow does not consider complex data flows in real-world apps which require additional instrumentation for detection. (3) *Restricted Call-graph:* WearFlow does not consider the life-cycle of modern real-world Android apps, and its generated call-graph does not reach code locations where sources and sinks are invoked.

FlowFinder addresses these technical challenges (as explained in Section 4.1) and performs better by detecting three flows. The data flows are not detected in the other two apps for two reasons. First, the call-graph generated by Soot (and used by FlowDroid) is not able to reach the code locations where the source and sink functions are invoked. Second, FlowFinder is not able to track tainted data when the data is stored in a file and accessed later by the app.

5. Users' Understanding of Runtime Permissions on Cross-device Apps

Our analysis of real-world apps revealed the existence of sensitive data flows in cross-device apps. These sensitive data flows violate users' privacy if they are unaware or not explicitly informed about such flows. We conduct a user study to investigate users' understanding of permissions in the existence of cross-device data flows. Our study aims to answer the following two research questions:

RQ1 How do users perceive permissions to work, and what are their mental models around apps' access capabilities in the existence of cross-device data flows?

RQ2 Can an adversary misguide users into unknowingly granting access to sensitive data?

To answer these research questions, we conducted an in-lab user study with 63 participants. First, we explored user awareness by asking participants about their understanding of which app(s) (i.e., the wearable app and/or the companion app) could access data when the relevant permission was granted only on one of the apps. Second, we investigated the feasibility of a cross-device permissions phishing attack by showing the participants malicious prompts and gauging their understanding of app capabilities. We evaluated the participants' understanding through categorical questions and semi-structured interviews for both research questions.

5.1. Participant Recruitment

We recruited participants by advertising our study on Slack channels, mailing groups, and distributing flyers. To participate in our study, we stipulated that participants must be over 18 years old, fluent in English, and active Android-based mobile device users. To avoid priming the participants, we do not reveal that the study is related to the security and privacy of mobile devices. Instead, we state that our study is about users' interaction with Android and Wear OS devices.

We designed an online screening questionnaire to determine if the participants had experience using Android and/or Wear OS devices. 150 participants completed our screening questionnaire. Among these participants, 92 had prior experience with Android and/or Wear OS devices. We invited these participants for an in-lab study. All 63 participants who completed the study were compensated with a $10 Amazon gift card. We provide the demographics of the participants in Appendix C.

Ethical Considerations. Our study was approved by Purdue University's IRB office. Prior to recording the interviews, we obtained participants' consent for the audio recording of the session. We provided participants with devices and did not collect any personal data during the study.

5.2. Interview Design

To answer **RQ1** and **RQ2**, we designed an in-lab user study where we invited participants to interact with a Google Pixel 3a smartphone running Android 9.0 and a Huawei Watch 2 running Wear OS 2. We note that we do not expect the minor UI changes in Wear OS 3 and the later Android versions to affect our results.

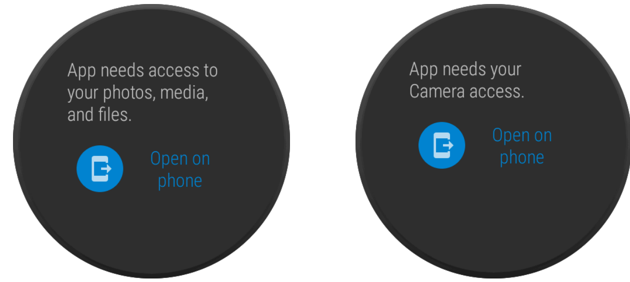


Figure 5: Redirection prompts shown to the participant on the smartwatch in Scenario-1: (Left) the benign prompt, (Right) the phishing prompt.

We divide our study into three scenarios to study users' understanding of the runtime permission model on cross-device apps and the consequences of granting permissions to sensitive data on the two devices. For each scenario, we provide step-by-step instructions for the participants to follow. After executing the steps for each scenario, we ask the participants a series of questions and collect their responses via audio recording (The screening and interview questions are available at our project GitHub repository). As a data collection method, audio recording is chosen over a survey to capture more detailed, explanatory, and contextually rich responses. We conclude the study by asking the participants to self-report their demographics.

Pairing Process. Each participant is first instructed to follow the standard procedure for pairing an Android smartphone with a Wear OS smartwatch. This involves navigating through a sequence of prompts on the smartphone and the smartwatch until the two devices are paired. After the devices are paired, we ask the participants if they have performed a similar pairing process before and how they generally respond to the prompts displayed during the process.

Scenario-1: Phishing. To investigate whether the runtime permission model on cross-device apps makes users susceptible to cross-device permission phishing attacks (as detailed in Section 3), we designed and developed a *Travel App* and installed it on both the smartphone and the smartwatch. The *Travel App* on the smartwatch shows a redirection prompt related to camera usage; when clicked, this prompt navigates the user to the companion app to grant storage permission.

To eliminate any associations or confounding factors involving the camera and storage permissions, we divided the 63 participants into two groups, i.e., `G.1` (control) and `G.2` (experimental). We compare the results of both groups to test our null hypothesis that malicious redirection prompts do not increase the number of users with a wrong perception of the granted permission. We define phishing as the inconsistency between the permission shown in the redirection prompt on one device and the actual permission requested on the other device. The scenario above highlights the impact of this inconsistency on users.

Figure 5 demonstrates the redirection prompts shown to the participants in the groups. The benign prompt on the left shows the user information about storage access, and the phishing prompt on the right shows the user information about camera access. `G.1` is presented with a benign version of *Travel App*, which shows information about storage permission on the redirection prompt, and the permission requested on the smartphone is for storage access as well. In contrast, `G.2` is presented with the phishing version of the same app, which displays that app needs camera permission on the redirection prompt while the permission requested on the smartphone is for storage access.

Since the participants in our study do not use their personal devices, we do not evaluate whether they grant or deny permissions. Following prior work [31], we evaluated their awareness of the consequences of different permission choices by asking them to read the prompts and grant the permission shown on the smartphone. Thereafter, we first asked both groups to describe the capabilities that the *Travel App* gains on the smartwatch and smartphone after granting permission. We then asked whether the *Travel App* on the smartphone is able to (1) access photos and media, (2) view SMS messages, and (3) take pictures and videos.

Scenario-2: Permissions Subjected to Cross-device Flows. To assess users' understanding of permissions in the existence of cross-device flows and their awareness of sensitive data exchange between a wearable app and its companion Android app, we ask 63 participants to interact with a real app, *Swim.com*. This app is used for "tracking the swimming performance and comparing stats with teammates and friends" [32] and is available on the Google Play Store.

We install the *Swim.com* app on both devices and ask the participants to interact with the apps for a few minutes to get familiar with the app's features. We then ask the participants to click the search icon on the *Swim.com* companion app to search for nearby pools. This action triggers a location permission prompt on the companion app. To understand if participants make different choices on two devices, we ask the participants to allow or deny the permission as they prefer. Next, we ask the participants to navigate to the smartwatch and launch the *Swim.com* wearable app. The app shows a location permission prompt once it opens, and we ask the participants to allow or deny this permission based on their preference. After these steps, we ask the participants whether they grant the location permission on the two devices.

Additionally, we ask participants about their perceptions when the location permission is granted on *Swim.com* companion app but denied on its wearable app. Based on this permission setup, we ask participants which *Swim.com* app(s) can access the smartphone's and smartwatch's locations.

Scenario-3: Location Notice. We also investigate whether the runtime permission model on cross-device apps impacts users' understanding of the smartwatch permission settings. For this, we ask the participants to navigate to the *Settings* → *Connectivity* → *Location from Phone & Watch* on the smartwatch, as in Figure 2. We then ask them to describe their understanding of this setting's functionality. We next instruct them to turn off the setting and ask them how the change of this setting impacts location access for companion and wearable apps.

TABLE 4: Participants' responses on apps' location access capability when the location permission is granted on the smartphone but denied on the smartwatch†.

Data	Choices	Participants
Phone's Location	Both Apps	19 (31.7%)
	Phone App	40 (66.7%)
	Watch App	0 (0%)
	Neither of the Apps	1 (1.7%)

† Three participants said "Not Sure", not shown here for brevity.

Pilot Study. After developing the first draft of the interview scenarios, we conducted pilot studies with four participants to evaluate the clarity of our interview design. We eliminated redundant questions and rectified any ambiguous language or unclear wording in the questionnaire based on participants' feedback. In the interviews, we introduced the phishing scenario and asked the related questions before the data flow scenario to mitigate bias and avoid unnecessarily alerting the users about the permission prompts.

Data Analysis. We used inductive coding to thematically analyze users' responses to our open-ended questions. Two authors first familiarized themselves with responses and generated initial codes independently. The two independent coders next grouped the codes into themes, ensuring that they accurately represent users' responses. Coders met to reconcile differences before generating the final codebook[1]. We note that coders achieved substantial agreement (Cohen's-kappa > 0.8).

5.3. Understanding of Permissions with Cross-device Flows

To assess users' mental models of permissions in the existence of cross-device data flows **[RQ1]**, we focus on their understanding of which device can access sensitive data when they grant or deny the relevant permission on the wearable and/or companion app. Specifically, we note participants' responses in Scenario-2 on which app(s) could access the smartphone and smartwatch's location data after they granted location permission on the smartphone but denied the location permission on the smartwatch.

Table 4 presents participants' understanding of access to location data by *Swim.com* companion and wearable apps. Among the 63 participants, only 19 (31.7%) state that both apps can access the smartphone's location data when the location permission is only granted to the companion app. The remaining 40 (66.7%) participants specify that only the companion app can access the smartphone's location data. This highlights that although the *Swim.com* app on both smartphone and smartwatch can access the smartphone's location through data flows, the majority of the participants (40 out of 63) are *unaware* of this data exchange.

Most participants are confused by the separate location permission requests on two devices. Participants consider that

1. The codebook is available at https://github.com/purseclab/WearOS.

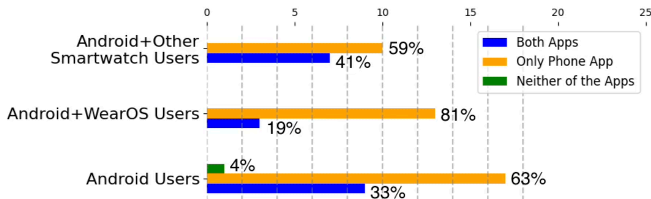


Figure 6: Participants' understanding of which app(s) can access the smartphone's location based on their prior experience with Android, Wear OS, and other smartwatches (Three participants that answered "Not Sure" are not included).

the apps on the two devices operate independently, regardless of the pairing process. Based on their understanding of app usage scenarios, participants make different choices for a given permission on the two devices. For instance, one participant stated, *"I prefer disabling location permission on the phone since I can search the nearby pools by myself, but I would prefer granting location permission on the watch since it may need to measure the distance I swam."* Moreover, they believe if permission to access sensitive data is granted on only one device, only that device can access it. For instance, one participant noted that if location permission is granted on the companion app, only this app can access the smartphone's and smartwatch's location. They stated *"The watch app won't be able to get the phone's location. I think the phone app can access the watch's location since I granted the location permission on the phone and the devices are paired. However, the watch app will be unable to get the phone's location since we denied the location permission on the watch."* Similarly, another participant highlighted *"Apps are on independent devices, so I would think that two apps do not depend on each other."*

Among the 19 participants who believe both apps can access location data, 16% believe data exchange is possible since they logged in to apps on both devices with the same account. For instance, one participant stated that *"Phone app has the location. The watch app will get the location data from the phone because the account in the app is the same. If there is no account, then the watch app will not get the location data."* However, this understanding is false since apps can exchange data without relying on a user account.

Among the participants, three participants were uncertain about which app(s) could access the location data. One of these participants explained that *"I am not sure. I just want to say phone app only. But I do not know if they are communicating, although I did the pairing steps."*

Interestingly, when asked about the smartwatch's location, most participants (49, 77.8%) correctly stated that none of the apps could access the smartwatch's location. They highlight that denying the location permission on the smartwatch limits apps' access to the smartwatch's location. Among the remaining 14 participants, five mentioned both apps could access the smartwatch's location, while eight believed only the companion app could access it. These participants reasoned that if devices are paired and in close proximity, both apps or at least the companion app can access the smartphone's location. Lastly, a participant stated only the wearable app could access the smartwatch's location.

Impact of Prior Device Usage. We investigated whether prior experience with Wear OS devices affects users' understanding of permissions when subjected to cross-device data flows in Scenario-2. 16 participants identified as active Android and Wear OS device users, 30 as only Android users, and 17 as Android users with experience using smartwatches other than the Wear OS-installed watches.

Figure 6 demonstrates the participants' responses when asked about which app(s) can access the smartphone's location after they granted location permission on the smartphone but denied it on the smartwatch in the data flow scenario. We observe that regardless of their experience with devices, participants are unaware that location data could be shared between the *Swim.com* apps on two devices. We perform the Fisher-Freeman-Halton test [33] on the responses from the three groups and observe no significant difference ($p = .40$). This shows that prior device usage does not change users' understanding of permissions across cross-device apps.

Impact of Smartwatch Settings. 51 out of 63 participants stated that a wearable app could not access the user's location after following the steps stated in the location notice and disabling the setting. These participants believe that turning off the location setting prevents any app on the smartwatch from accessing location data from the smartwatch or smartphone. For instance, one participant stated *"The setting should turn off getting data from phone and watch, that's why the watch app cannot know the user's location."* This understanding aligns with the location notice shown to the participants during pairing. Yet, our analysis shows that a wearable app can still access the user's location through the companion app with location access. Therefore, the information provided by Google during pairing further exacerbates users' confusion about sensitive data exchange between cross-device apps.

Summary of Findings. Our study showed that 66.7% of the participants believe that only the companion app can access the smartphone's location. This observation demonstrates that participants are confused by the separate permission requests on the smartphone and smartwatch. They are *unaware* of the data flows between the wearable app and its companion app and do not fully understand the consequences of granting permissions to sensitive data on cross-device apps.

5.4. Feasibility of Cross-device Permission Phishing

Using the participant responses in Scenario-1, we evaluate whether deceptive explanations on redirection prompts can impact the users' perception of which permission is granted **[RQ2]**. We note that Android users typically have an understanding that an app should access storage (i.e., photos, media, and files) to take pictures and videos (i.e., the camera permission) [5]. Therefore, to determine whether users' understanding of storage access is affected by the malicious redirection prompt or their prior misunderstandings about permissions, we consider the difference in the number

TABLE 5: Participants' understanding of app's storage access after interacting with benign (G.1) and phishing (G.2) apps.

	App can access storage	App cannot access storage
G.1†	33 (100%)	**0 (0%)**
G.2	24 (83%)	**5 (17%)**

† One participant in G.1 answered "Not Sure".

of participants that believe the app has storage access in the benign (G.1) and phishing (G.2) groups.

Table 5 shows participants' responses in G.1 (presented with a benign app) and G.2 (presented with a phishing app) when asked about whether the *Travel App* can access device storage (i.e., photos, media, files). While all the participants in G.1 stated that the *Travel App* could access storage, a significant portion of the participants in G.2 (5 out of 29) said that the app could not access it. Comparing results for G.1 and G.2, we find that phishing redirection prompts effectively deceive the participants into unintentionally granting permission for storage access. Specifically, the percentage of participants who mistakenly believe that the app cannot access storage after seeing the redirection prompt increases from 0% in G.1 to 17% in G.2.

To determine whether the cross-device permission phishing attack causes statistically significant results, we tested the null hypothesis H_0: "Malicious redirection prompt does not increase the number of participants with a wrong perception of which permission is granted." To evaluate the hypothesis, we performed Fisher's exact test (one-tailed) [34]. Fisher's exact test shows a significant difference between the control and phished groups ($p = .018$). Thus, we reject the null hypothesis and conclude that malicious redirection prompts lead to an increased number of participants with an incorrect perception of the granted permissions.

When we asked participants why they believed the *Travel App* could access storage, most participants noted that the permission dialog mentioned storage and therefore the app gains storage access. However, a few participants explained that they consider the redirection prompts as permission dialogs; thus, when they click to proceed, they think that the permission is granted. For instance, one stated that *"I gave the permission on watch...Because I started giving the permission on watch."* Moreover, a few participants in G.2 who believed the app could access storage highlighted that this was possible since the app had camera access. For instance, one participant stated that *"It asked for the camera permission. So, it can also access photos and videos."* Similarly, another participant said that *"It needs to save the image somewhere, so it can access the storage."* This aligns with users' misunderstanding about permissions, where they believe granting camera permission to an app automatically grants it storage access [5].

Among the phished participants in G.2 who believed app could not access storage, we noted that participants did not pay attention to the permission dialog they saw after being directed from the redirection prompt and immediately granted the permission without reading. For example, one participant said that *"I don't remember what I saw on the phone, but I only remember some information about the camera."* It demonstrates that these participants were phished as they believed they granted a different permission (i.e., camera) than the one that was actually requested (i.e., storage). Lastly, we did not observe any significant correlation between participants' device usage and their phishing susceptibility.

Summary of Findings. Our study showed that malicious redirection prompts could mislead participants with an incorrect perception of the granted permission, revealing users' sensitive data without their awareness.

6. Other Wearable Platforms and Ecosystems

We conducted a preliminary study on other watch platforms and ecosystems (i.e., watchOS, Fitbit, Garmin OS) to understand the extent of the identified issues across different mobile and wearable platforms.

iOS and watchOS. We investigated the Apple ecosystem to understand how it deals with the cross-device transfer of permission-protected data for wearable devices. Figure 9 in Appendix A illustrates permission dialogs and toggles on watchOS. We found that Apple also adopts a "dual permission model", but it implements further improvements, possibly to mitigate the issues discussed in this paper. Specifically, Apple implements permission synchronization, where the decisions made by the user on one device will be propagated to the other paired device. Interestingly, while apps can request permissions on either device through permission dialogs, users are allowed to revoke permissions only on their phones as the permission settings are disabled on the watch.

This strategy takes an initial step at addressing the found issues; however, we argue that it is far from being fully effective due to the limited flexibility, control, and information provided to users over their permissions. First, we observed that Apple does not provide any information directly via the system prompts to the users regarding adopting the permission synchronization strategy; hence, users might be unaware that their permission decisions on one device are being copied to the other device. Second, once permissions are granted, the user cannot revoke them on the watch. These design decisions significantly limit the users' control over their own data and may potentially create serious privacy issues for the users.

To illustrate these issues, we consider a scenario where a user installs an app on their iPhone and Apple watch and goes for a run, leaving their phone at home. Suppose the user does not intend to be tracked during their run. First, if the user granted permission to this app on their phone, they might not be aware that the wearable app can also access their location. Second, the user might decide to stop sharing their location with the app during their run when they suddenly realize they cannot do so as they no longer have access to their phone. They would have to turn off the location on their watch, which, undesirably, disables location access to all the wearable apps. We note that the strategy to turn off location completely would not work for other data types as only a subset has designated toggles.

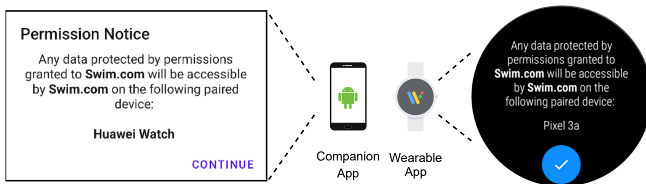


Figure 7: Information prompts to better inform the user about the possible cross-device sensitive data flows.

Fitbit and Garmin OS. Fitbit and Garmin OS have their own permission models working in conjunction with the mobile platform they are paired with. For both platforms, the watch permissions can be granted only on the phone via the designated companion apps of these platforms (i.e., Garmin Connect and Fitbit apps), as shown in Figure 10 in Appendix A. For Garmin OS, permissions are granted at installation and can never be revoked. Fitbit also uses install-time permissions but allows users to revoke them after installation.

Two separate permission models operating independently on user data in Fitbit and Garmin OS make both watch platforms vulnerable to the issues discussed in this paper. However, we argue that the strategy to enable permission control only on the phone at installation may exacerbate the problem. This is because it makes it less likely for users to be conscious of the interactions between the two devices that may have access to sensitive data.

Summary. Our study revealed that watchOS, Fitbit, and Garmin OS have similar privacy issues we discussed thoroughly for Wear OS. These findings suggest that the management of sensitive data and permissions in the existence of cross-device data transfer is a complex problem across mobile and wearable platforms, and there is a need for a careful design to address all use cases and scenarios while balancing usability and privacy.

7. Countermeasures

To address the found issue, we introduce several possible improvements to the current permission model in Wear OS, and we present them in order of implementation complexity. These improvements aim not only to increase the security of privileged data but also to improve users' understanding regarding the use of sensitive data by apps. We note that the solutions we provide are intended as recommendations that provide guidance on the possible directions to take to address the issues. All the proposed solutions require rigorous design and user testing before adoption in the real world.

Additional Information Prompts. For users to correctly understand the consequences of granting permission, an additional information prompt can be shown to them by the Android or Wear OS platform, as depicted in Figure 7. Before the user grants permission on one device, a system prompt should inform the user that by granting the mentioned permission, the permission-protected data can be potentially accessed by the other paired device. To improve user experience, this new prompt could be shown to the user only once every few minutes per app instead of displaying it before requesting permission. Another possible alternative is to integrate this new text into permission requests without introducing a new prompt.

Unified Permission Model. A possible alternative design for a permission model for cross-device apps would be to allow the user to control the permissions of a cross-device app using a unified permission model. In this design, the permission decisions taken by the user on one device will be synchronized by default to the other device. However, to allow for flexibility, the user will be given the option to overwrite their decisions on either device. As can be seen, this design corresponds to what is currently implemented in the Apple smartphones/smartwatches ecosystem [35]. However, by giving the privacy-conscious user the option to change their permissions on either device, we prevent the issues discussed previously in Section 6.

To securely manage permissions on both devices in the aforementioned way necessitates a permission management protocol that enables cross-device permission synchronization. This is easily achieved for devices like the Apple watch, which is only designed to be paired with an iPhone, and both devices share the same, close ecosystem since they are products of the same company. On the other hand, Wear OS is designed to be used by watches from multiple vendors and is meant to be used with Android smartphones, as well as with smartphones from other vendors including iPhones. To make matters worse, many existing wearable devices (e.g., Garmin, Fitbit) can be paired with both iPhones and Android devices. Therefore, it is challenging to develop a standardized cross-device permission management protocol supported by all the operating systems and wearable devices.

A solution that would work for all wearable platforms and ecosystems is a highly challenging, if not infeasible, task as it requires the cooperation of many smartphone and wearable device vendors. For this reason, we limit our focus to solving the problem for vendors that simultaneously control both the mobile and the paired wearable device (e.g., when a Wear OS watch is working with Android). On Wear OS, we propose using a unified permission model that synchronizes permissions by default when paired with an Android phone and gives the conscious user the option to override this synchronization to provide flexibility. Since Android and Wear OS are managed by Google, developing and deploying a unified permission model is practical.

Cross-device Data Flow Detection. App markets such as Google Play can use program analysis techniques to detect potentially malicious cross-device data flows. For this, the techniques we implemented in FLOWFINDER can be used to detect any unauthorized flows that use the Data Layer API on Android and Wear OS. Although this approach does not work for sideloaded apps, it can serve as a complimentary strategy to the two aforementioned countermeasures.

8. Discussion and Limitations

First, we used Wear OS 2 and Android 9 in our user study. While changes exist in the permission UIs of the later versions of Android and Wear OS, we expect their effect on our results to be marginal since the changes are minimal and not directly related to the phenomena tested in our study.

Second, our main focus is on the permission model of Android and Wear OS in the context of cross-device apps. We have only conducted a preliminary investigation on other wearable platforms (i.e., watchOS, Fitbit, and Garmin OS); however, we were able to demonstrate the existence of similar issues on all these platforms due to the dual permission model in place to guard user data. We leave a more thorough investigation of other wearable platforms to future work.

Third, FLOWFINDER builds upon existing static analysis tools (e.g., Flowdroid [19]) and therefore inherits their limitations. We implemented several techniques to address various limitations of these tools, as discussed in Section 4. However, further improvements are possible, such as propagating taints across file operations and obfuscated APIs.

Fourth, we do not evaluate users' prior understanding of permission models. Previous work extensively analyzed users' comprehension of runtime permissions on a single device [3], [2], [36], [6], [5]. However, users' understanding of multi-device permission models remains largely unexplored. Hence, we extend prior literature by focusing on users' understanding of runtime permissions for cross-device apps.

Lastly, our proposed improvements require changing the Android and Wear OS permission systems. To evaluate these changes, we plan to implement them and conduct a user study to compare the results of user perception between the current permission model and our proposed models.

9. Related Work

Extracting Sensitive Data Flows. A line of prior work has studied data flows in Android apps [37], [38], [39], [40], [19], [7]; however, they do not consider Android apps' communication with Wear OS and do not propagate taints for the Data Layer APIs used for cross-device communication. A recent work proposed WearFlow [7] that models the DataLink API functions to detect data flows between the wearable app and its companion app. However, as detailed in Section 4, WearFlow cannot accurately detect the sensitive data flows in real-world apps. In contrast, we detect sensitive data flows by identifying the relevant source and sink functions and performing instrumentation on the apps.

Users' Comprehension of Permission Models. Another line of prior work has investigated users' comprehension of Android permissions granted at app installation [41], [1], [4]. With the introduction of the runtime permission model in Android 6.0, several recent works have explored different factors that affect users' decisions for granting runtime permissions [3], [2], [36], [6], [5]. While these works revealed users' comprehension of permission models, they focused on permissions on a single device. In contrast, we investigate the user's understanding of permissions in the existence of cross-device data flows for wearables.

Attacks on Android Permissions. Prior research has demonstrated a variety of attacks on Android permissions to obtain users' sensitive data from a single mobile device [42], [43], [44], [45], [31], [46]. In contrast, we demonstrate that an adversary can leverage developer-generated redirection prompts, triggered by permission requests from the wearable app to its companion app (and vice versa), to phish users to unintentionally grant permissions in a cross-device setting.

Improvements for Runtime Permission Model. Recent work has proposed improvements on Android permissions to provide more comprehensive information, enabling users to make better-informed decisions [6], [47]. Yet, we propose additional rationale messages to help the users understand the runtime permission model on cross-device apps.

10. Conclusions

In this paper, we systematically studied the permission model of Wear OS and demonstrated the privacy issues caused by the possibility of cross-device sensitive data transfer between wearable devices and their smartphone companions. Via taint analysis, we demonstrated that there are apps on Google Play that have sensitive data flows between the wearable app and its companion app. Subsequently, we conducted an in-lab user study to assess users' understanding of permissions and their awareness of cross-device sensitive data flows. Our findings revealed that users struggle with determining the access capabilities of apps in the existence of cross-device data flows due to a lack of awareness of such flows and are also vulnerable to cross-device phishing attacks. Although our main focus was on Android and Wear OS, we also conducted a preliminary study on other watch platforms and showed that they suffer from similar privacy issues, demonstrating that cross-device sensitive data transfer is a complex problem across mobile and wearable platforms. Lastly, we proposed potential improvements (1) in the app markets to identify cross-device data flows and (2) on the watch and mobile platforms to better inform users about sensitive data flows, creating transparency regarding the handling of user data. Informing users about the sensitive data flows through these improvements could potentially improve the security and privacy of cross-device apps and ultimately lead to increased user trust in the mobile ecosystem.

Acknowledgments

This work has been partially supported by the National Science Foundation (NSF) under grant CNS-2144645 and by Google with an ASPIRE Research Award. We thank Ehsan Nourbakhsh, Bjorn Kilburn, Mario Kosmiskas, Kamran Mustafa, and Arpit Midha of Wear OS, Dave Kleidermacher, Michael Specter, and Sudhi Herle of Android Security and Privacy of Google, as well as our anonymous reviewers and shepherd for their feedback.

References

[1] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David A Wagner, et al. How to ask for permission. In *HotSec*, 2012.

[2] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. Exploring decision making with android's runtime permission dialogs using in-context surveys. In *SOUPS*, 2017.

[3] Weicheng Cao, Chunqiu Xia, Sai Teja Peddinti, David Lie, Nina Taft, and Lisa M Austin. A large scale study of user behavior, expectations and engagement with android permissions. In *USENIX Security*, 2021.

[4] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[5] Bingyu Shen, Lili Wei, Chengcheng Xiang, Yudong Wu, Mingyao Shen, Yuanyuan Zhou, and Xinxin Jin. Can systems explain permissions better? understanding users' misperceptions under smartphone runtime permission model. In *USENIX Security*, 2021.

[6] Yusra Elbitar, Michael Schilling, Trung Tin Nguyen, Michael Backes, and Sven Bugiel. Explanation beats context: The effect of timing & rationales on users' runtime permission decisions. In *USENIX Security*, 2021.

[7] Marcos Tileria, Jorge Blasco, Guillermo Suarez-Tangil, et al. Wearflow: Expanding information flow analysis to companion apps in wear os. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[8] Wear OS by Google. https://wearos.google.com/, 2022. [Online; accessed 01-March-2023].

[9] Standalone versus non-standalone wear os apps. https://developer.android.com/training/wearables/overlays/standalone-apps, 2022. [Online; accessed 01-March-2023].

[10] Android permission overview. https://developer.android.com/guide/topics/permissions/overview, 2022. [Online; accessed 01-March-2023].

[11] Request runtime permissions. https://developer.android.com/training/permissions/requesting, 2022. [Online; accessed 01-March-2023].

[12] Manifest.permission. https://developer.android.com/reference/android/Manifest.permission, 2022. [Online; accessed 01-March-2023].

[13] Explain access to more sensitive data. https://developer.android.com/training/permissions/explaining-access, 2022. [Online; accessed 01-March-2023].

[14] Wear OS by Google Smartwatch. https://play.google.com/store/apps/details?id=com.google.android.wearable.app&hl=en_US&gl=US, 2022. [Online; accessed 01-March-2023].

[15] Galaxy wearable app. https://play.google.com/store/apps/details?id=com.samsung.android.app.watchmanager, 2022. [Online; accessed 01-March-2023].

[16] Data layer api. https://developer.android.com/training/wearables/data/data-layer#send-and-sync-with-API, 2022. [Online; accessed 01-March-2023].

[17] Android runtimepermissionswear sample. https://github.com/android/wear-os-samples/tree/main/RuntimePermissionsWear, 2022. [Online; accessed 01-March-2023].

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, 2010.

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *SIGPLAN*, 2014.

[20] Susi: Source and sink. https://github.com/secure-software-engineering/SuSi, 2022. [Online; accessed 01-March-2023].

[21] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting Android permission specification analysis. In *USENIX Security*, 2016.

[22] Android's kotlin-first approach. https://developer.android.com/kotlin/first, 2023. [Online; accessed 12-April-2023].

[23] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of Android apps for the research community. In *Int. Conference on Mining Software Repositories (MSR)*, 2016.

[24] Goko store. https://goko.me/, 2022. [Online; accessed 01-March-2023].

[25] Android wear center. https://www.androidwearcenter.com/, 2022. [Online; accessed 01-March-2023].

[26] Package and distribute wear apps. https://developer.android.com/training/wearables/packaging, 2022. [Online; accessed 01-March-2023].

[27] Multiple apk support. https://developer.android.com/google/play/publishing/multiple-apks, 2022. [Online; accessed 01-March-2023].

[28] Google Android debug bridge (adb). https://developer.android.com/studio/command-line/adb, 2022. [Online; accessed 01-March-2023].

[29] Google fit. https://www.google.com/fit/, 2023. [Online; accessed 12-April-2023].

[30] Marcos Tileria, Jorge Blasco, Guillermo Suarez-Tangil, et al. Wearbench, 2020.

[31] Güliz Seray Tuncay, Jingyu Qian, and Carl A Gunter. See no evil: phishing for permissions with false transparency. In *USENIX Security*, 2020.

[32] Swim.com app. https://www.swim.com/, 2022. [Online; accessed 01-March-2023].

[33] GH Freeman and John H Halton. Note on an exact treatment of contingency, goodness of fit and other problems of significance. *Biometrika*, 1951.

[34] Peter Sprent. Fisher exact test. In *International Encyclopedia of Statistical Science*, 2011.

[35] Watchos permissions. https://developer.apple.com/documentation/watchos-apps, 2023. [Online; accessed 1-April-2023].

[36] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. Permissions snapshots: Assessing users' adaptation to the android runtime permission model. In *IEEE WIFS*, 2016.

[37] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 2014.

[38] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[39] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *SIGPLAN*, 2014.

[40] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *ICSE*, 2015.

[41] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM CCS*, 2011.

[42] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

[43] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A Gunter. Resolving the predicament of Android custom permissions. In *NDSS*, 2018.

[44] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *USENIX Security*, 2019.

[45] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the Android user interface. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[46] Xueqing Liu, Yue Leng, Wei Yang, Wenyu Wang, Chengxiang Zhai, and Tao Xie. A large-scale empirical study on android runtime-permission rationale messages. In *IEEE VL/HCC*, 2018.

[47] Lynn Tsai, Primal Wijesekera, Joel Reardon, Irwin Reyes, Serge Egelman, David Wagner, Nathan Good, and Jung-Wei Chen. Turtle guard: Helping android users apply contextual privacy preferences. In *SOUPS*, 2017.

Appendix A. Wearable Platforms and Ecosystems

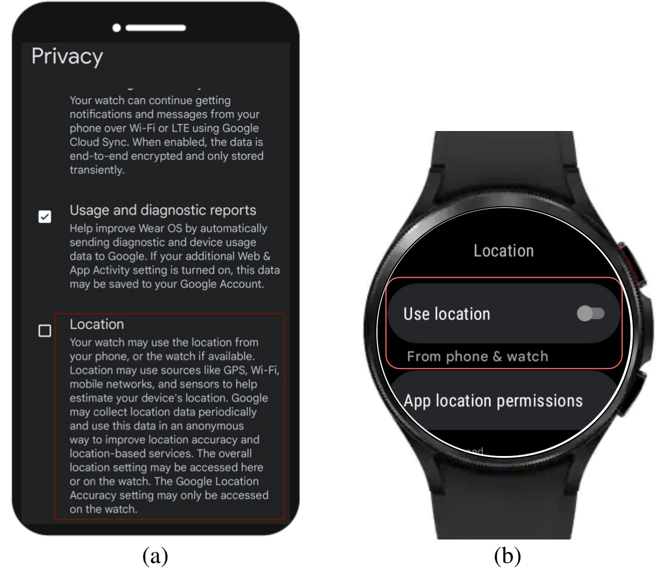
Wear OS. Figure 8 demonstrates the location notice shown in the pairing process of the Google Pixel Watch App and Galaxy Wearable App, and the watch setting on the Google Pixel Watch App for Wear OS 3. On the Galaxy Wearable App, users can revisit this setting by navigating to Galaxy Wearable App→Watch Settings→About Watch→Legal Information→Google Privacy Options. We note that the location notice and watch setting are synchronized. Specifically, when the user turns off the watch location setting, the choice for the location notice on the smartphone is also updated. Yet, this change does not affect its location setting ("on").

iOS and watchOS. Figure 9 shows the permission model in the iOS-watchOS ecosystem. Within this ecosystem, the user can interact with a permission dialog on any app (wearable or companion app). Upon interacting with this dialog, the user's decision is applied to the companion device (or vice versa). We note that once this decision has been made, it cannot be modified via the watch settings. Instead, users seeking to change their permission choices must navigate to the phone settings.

Fitbit and Garmin OS. Figure 10 depicts the install-time permission models for Fitbit and Garmin OS. The user needs to grant multiple permissions within the same prompt. After granting the permissions, the app can access the system resources to provide its functionality.

Appendix B. Dataflow Details

Table 6 presents the number of apps after each intermediary step in the app collection presented in Section 4. After scraping the apps, we have 5,415 package names. Following that, we filter out the duplicate ones since two third-party websites and the Google Play Store have some common package names listed on their websites. We then



(a) (b)

Figure 8: (a) Location setting shown in the pairing on Android and (b) watch location setting shown on Wear OS.

TABLE 6: The number of apps after each intermediary step in creating the cross-device app dataset.

App Dataset Details	
Number of scraped package names	5415
Number of unique package names	3691
Number of existing package names	1892
Number of downloaded package names	336

check whether the app is still listed in Google Play Store. As a last step, we filter only the wearable apps, having 336 wearable and companion apps in our dataset.

Table 7 demonstrates the full results of data flow comparison of FlowFinder with WearFlow on 40 apps (See Section 4). Overall, FlowFinder yields 37% lower false negatives than WearFlow. We also test against WearFlow's benchmark dataset called WearBench. Since the WearBench apps do not transfer permission-protected data, we added the required data-source methods to evaluate FlowFinder on WearBench. With the required methods, FlowFinder detects all of the flows in the WearBench dataset.

Appendix C. Participant Demographics

In Table 8, we show the participant demographics. We note that our screening and interview questions are available at our project GitHub repository, https://github.com/purseclab/WearOS.

Figure 9: Illustration of permission model on iOS-watchOS. (a) The user grants permission on the watch. (b) The watch setting is disabled against any permission decision change. (c) The user can modify permission settings via the phone setting.

TABLE 7: Data flow comparison results on the validation dataset. ● means flow is found, ○ means flow is not found.

App/Package Name	WearFlow	FlowFinder
Benchmark Apps		
Location	○	●
DataNetwork	○	●
NetworkType	○	●
DownloadedFile	○	●
AuthToken	○	●
WearBench Apps		
Asset	●	●
DataItem1	●	●
DataItem2	●	●
InterDataItem	●	●
InterDataItem2	●	●
SimpleChannel	●	●
Messages	●	●
SimpleDataItem	●	●
SimpleDataItem2	●	●
SimpleDataItem3	●	●
SimpleDataItemOld	●	●
SimpleMessage	●	●
SimpleMessage2	●	●
SimpleMessage3	●	●
SimpleMessageOld	●	●
Real-World Apps with Flows		
com.sparkistic.photowear	○	○
com.estela	○	●
com.ammarptn.willow.digital.watch.face	○	○
com.moletag.gallery	●	●
com.skimble.workouts	○	●
com.thehoodiestudio.rwrk	○	●
com.sparkistic.photowear	○	○
com.rockgecko.dips	○	●
com.smartartstudios.hexane.interactive.watchface	○	●
de.esymetric.rungps_trial	○	●
Real-World Apps with No Flows		
com.sousoum.droneswear	○	○
com.appfour.wearweather	○	○
com.kjsk.watchface.jesus	○	○
ch.sooon.hub	○	○
huskydev.android.watchface.atlas	○	○
com.mydiabetes	○	○
com.hole19golf.hole19.beta	○	○
com.opl.transitnow	○	○
com.watch.richface.neo	○	○
adarshurs.android.vlcmobileremote	○	○

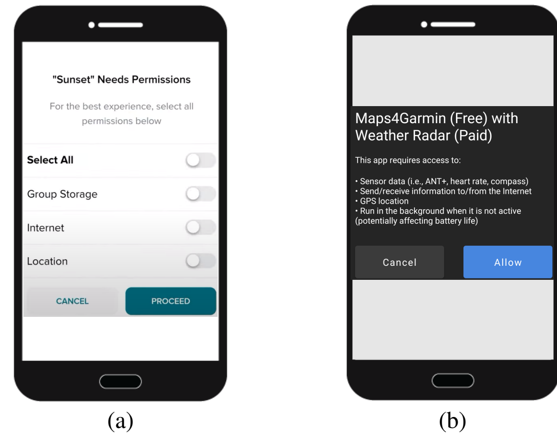


Figure 10: Illustration of the install-time permission models of (a) Fitbit and (b) Garmin OS.

TABLE 8: Demographics of participants.

	Choices	Participants
Age	Between 18 and 25	29 (46%)
	Between 26 and 34	28 (44%)
	Between 35 and 50	6 (10%)
Gender	Man	43 (68%)
	Woman	19 (30%)
	Prefer not to say	1 (2%)
CS Related	Yes	40 (63%)
	No	23 (37%)
Education	Up to high school	5 (8%)
	Some college (1-4 years, no degree)	6 (10%)
	Bachelor's degree	31 (49%)
	Graduate degree	21 (33%)
Smartwatch Usage Time	More than 1 year	18 (29%)
	A few months	13 (21%)
	Less than one month	2 (3%)
	I don't have a watch	30 (48%)
Smartwatch Usage Purposes	Messaging	20 (61%)
	Activity tracking	31 (94%)
	Phone calls	13 (39%)
	Music	14 (42%)
	Navigation	8 (24%)
	Payments	8 (24%)
	Other	7 (21%)
Similar Pairing Experience	Yes	38 (60%)
	No	25 (40%)

Appendix D.
Meta-Review

D.1. Summary

The paper develops a static analysis tool to study the interaction between wearable apps on Wear OS devices and their companion apps on Android devices. An analysis of 150 real-world apps finds that in 28 apps, presumable unintended, protected data flows between those apps. A user study with 63 participants indicates that most users are unaware of this data transfer and that users can be phished into granting permissions.

D.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability

D.3. Reasons for Acceptance

1) The paper confirms data leakage between the Wear OS app and the companion app, similar to the study by Tileria et al. (WearFlow), by developing a tool that extends the state-of-the-art solution to increase coverage of flows (for example, an augmented list of sources and sinks)
2) The paper complements those results with a user study that identifies gaps in the users' understanding of the permission model in the cross-device app setting.

D.4. Noteworthy Concerns

The methodology for studying the phishing attacks could have been improved: Phished and benign permission may be too close to each other to have a strong enough effect size to capture users' susceptibility to phishing accurately. Users typically have an understanding that an app should access storage to take pictures and videos. Choosing a phished permission unrelated to the phishing scenario (camera remote control) would have been a better choice. Additionally, the study design asked users "what they think the app can do" instead of whether they knowingly granted the storage permission. This type of question may trigger users to rationalize app behavior (e.g., creating these aforementioned associations like "if the app can take pictures, it also can access storage").

Appendix E.
Response to the Meta-Review

We split the noteworthy concern into two parts regarding our phishing study (Section 5.2): (1) semantically-related permissions and (2) direct questions vs. general queries of app capabilities. We answer each concern below.

Semantically-related Permissions. It is true that the permissions we used in our study to assess the feasibility of cross-device permission phishing (i.e., camera and storage permissions) are semantically related from the user's perspective, which has the potential to mislead them into not being able to correctly differentiate the capabilities of these two permissions [5]. This might make it difficult to distinguish in a user study such as ours if users fell victim to the phishing attack or are simply confused about the capabilities of the two permissions due to their apparent semantic connection.

We, however, accounted for this phenomenon and eliminated its effects in our study by checking if the likelihood of misperception of the granted permission among the participants increases under the phishing scenario. More specifically, we formed the null hypothesis as "Malicious redirection prompts do not increase the number of participants with a wrong perception of which permission is granted".

We consequently conducted our study in two phases, where 1) we had an experimental group presented with the phishing scenario in which the redirection prompt shows camera-related information and the requested permission is storage and 2) a control group presented with a benign scenario, where the redirection prompt shows storage-related information and the requested permission is storage. For the purposes of this study, we define phishing in terms of a misconception of the granted permission: If a user believes that the storage permission has not been granted, despite them having just granted it, such a user is categorized as "phished". Our findings indicate that the phishing prompt does indeed increase the likelihood of users misinterpreting which permissions they have granted.

Direct Questions vs. General Queries of App Capabilities. The preliminary findings in our pilot study show that participants are unable to articulate their reasoning with technical terms, such as "permission". For this reason, rather than posing direct questions about the permissions granted, we were careful to use terminology that is familiar to Android users. Specifically, in the phishing part of our user study, we first asked participants generic questions about the app's functionality. Subsequently, we asked more targeted questions regarding app capabilities after the permission grant. While framing our questions, we aligned them with the terminology used in the system dialogs. For instance, the Android system prompts show the following prompts:

- Allow [Appname] to access photos and media on your device? [storage permission]
- Allow [Appname] to take pictures and record videos? [camera permission]

and we framed our questions as follows:

- Do you think the [Appname] on the phone is able to access photos and media?
- Do you think the [Appname] on the phone is able to take pictures and videos?