WearFlow: Expanding Information Flow Analysis To Companion Apps in Wear OS

Marcos Tileria
Royal Holloway, University of London

Jorge Blasco
Royal Holloway, University of London

Guillermo Suarez-Tangil
King's College London
IMDEA Networks

Abstract

Smartwatches and wearable technology have proliferated in the recent years featured by a seamless integration with a paired smartphone. Many mobile applications now come with a companion app that the mobile OS deploys on the wearable. These execution environments expand the context of mobile applications across more than one device, introducing new security and privacy issues. One such issue is that current information flow analysis techniques can not capture communication between devices. This can lead to undetected privacy leaks when developers use these channels. In this paper, we present WearFlow, a framework that uses static analysis to detect sensitive data flows across mobile and wearable companion apps in Android. WearFlow augments taint analysis capabilities to enable inter-device analysis of apps. WearFlow models proprietary libraries embedded in Google Play Services and instruments the mobile and wearable app to allow for a precise information flow analysis between them. We evaluate WearFlow on a test suite purposely designed to cover different scenarios for the communication Mobile-Wear, which we release as `Wear-Bench`. We also run WearFlow on 3K+ real-world apps and discover privacy violations in popular apps (10M+ downloads).

1 Introduction

Wearable devices are becoming increasingly popular and can now run apps on appliances with large computing, storage, and networking capabilities. According to Gartner, users will spend $52 billion in wearable in 2020 [14], smartwatches being the most popular gadget. The key feature of these devices is that they are all interconnected, and provide a usable interface to interact with smartphones and cloud-based apps. In Android, wearable devices interact with the smart phone via Wear OS (previously, Android Wear). Wear OS is similar to Android in terms of architecture and frameworks but it is optimized for a wrist experience. Apps in Wear OS can run as standalone programs or companion apps.

Wearable devices provide an additional interface with the digital world, but they are also a potential source of vulnerabilities that increases the attack surface. For instance, a mobile app could access sensitive information and share it with its companion app in another device. Then, the companion app could exfiltrate that information to the Internet. This landscape expands the context of mobile applications across more than one device. Therefore, we cannot assess the security of a mobile app by just looking at the mobile ecosystem in a vacuum. Instead, we need to consider also the wearable app as part of the same execution context.

Previous studies have exposed vulnerabilities on smartwatches and their ecosystem [10, 12, 16, 30]. However, these works have mostly focused on the analysis of wearable apps in isolation [12], their Bluetooth connectivity [16] or the usage of third-party trackers [8]. To systematically study how apps use sensitive data, the security community leverages information flow analysis [3, 9, 15, 18, 21, 32]. Recent works such as COVERT [4], DidFail [18], and DialDroid [6] augment the scope of the data tracking to include inter-app data flows which use inter-component communication (ICC) methods. These works expand the execution context from one mobile app to a set of mobile apps.

In contrast to previous problems, information flow analysis in the wearable ecosystem needs to track sensitive data across apps in different devices, i.e.: the handheld and the wearable. In other words, it needs to consider that data flows can propagate from the mobile app to its companion app (and back) through the wireless connection. In Android, this communication is manged by Google Play Services, a proprietary application which handles aspects like serialization, synchronization, and transmission (among other aspects within the Android ecosystem).

In this work, we present WearFlow, a framework to enable information flow analysis for wearable-enabled applications. To achieve this, we create a model of Google Play Service by leveraging the Wear OS Application Program Interface (API). This enables WearFlow to capture inter-device flows. Thus, we run taint analysis on each app and reason about flows in an

extended context that comprises mobile and companion apps. Our results show that WearFlow can detect Mobile-to-Wear and Wear-to-Mobile data leaks with high precision and finds evidences of misuse in the wild.[1]

In summary, we make the following contributions:

1. We propose WearFlow, an open-source tool that uses a set of program analysis techniques to track sensitive data flows across mobile and wearable companion apps. WearFlow includes library modeling, obfuscation-resilient APIs identification, string value analysis, and inter-device data tracking.
2. We develop `WearBench`, a novel benchmark to analyze Mobile-Wear communications. This test suite contains examples of mobile and wearable apps sharing and exfiltrating sensitive data using wearable APIs as the communication channel.
3. We conduct a large scale analysis of real-world apps. Our analysis reveals that real-world apps use wearable APIs to send sensitive information across devices. Our findings show that developers are not using data sharing APIs following the guidelines given by Google.

The rest of the paper is structured as follows. Section 2 provides an overview of wearable companion apps and Google Play Services. Section 3 presents the security threats of the Mobile-Wear ecosystem. We describe how we model Google Play Services in Section 4. We present WearFlow in Section 5. We evaluate our solution and present the results of our large scale analysis in Section 6. We discuss the limitations of WearFlow and other related works in Sections 7 and 8. Finally, we present our conclusions in Section 9.

2 Background

This section describes the Android-Wear ecosystem, including how Wear applications communicate with their mobile or handheld counter part via Google Play Services. We also provide a motivation example to show the challenges behind tracking data usage in this ecosystem.

2.1 Wearable apps

Wear OS is a stripped version of Android optimized to run wearable apps on Android smartwatches. The capabilities of these smartwatches range depending on the hardware of the manufacturer. Apart from main components such as screen and CPU, these devices incorporate an array of sensors including accelerometers, heart-rate and GPS among others. The Wear OS provides an abstraction for apps to access those sensors.

[1]For simplicity, we refer to term Mobile-Wear when we use Mobile-to-Wear and Wear-to-Mobile interchangeably.

Wear OS adopts the same security model used to protect its mobile counterpart. In Android, applications are sandboxed and installed with minimum permissions by default. From Android 6.0, dangerous permissions are not granted at installation-time, but during run-time. Permissions still need to be declared on the app Manifest. The same permission model applies to Wear apps, however the authorization process is independent. This is, permissions are not inherited from the mobile app. The wearable app must request permission to access protected resources. These resources can be either in the smartwatch or in the smartphone (the smartwatch can also access resources in the smartphone and vice-versa provided users grant the appropriate permissions).

Wear devices are also equipped with network connectivity like Bluetooth, NFC, WiFi, or even access to cellular networks. Most watches require a phone pairing process via Bluetooth or WiFi. The pairing process establishes a low-level channel that can be used by mobile apps to communicate with a companion app in the smartwatch. Note, however, that wearable apps can run standalone apps (i.e., no mobile app needed) from Wear OS 2.0. Figure 1 illustrates the interplay between a mobile phone, a smartwatch and the network. We next describe how Wear OS enable apps to communicate with each other (including to how they communicate with the mobile companion app).

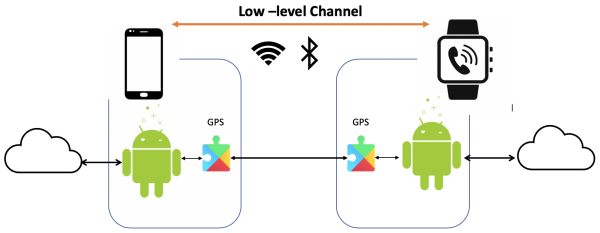

Figure 1: Communication between a mobile app, its companion, Google Play Services (GPS) and the network.

2.2 Google Play Services

While Android is an open-source OS, most “stock” Android devices run proprietary software from manufacturers (OEMs) and third-parties [13]. To access the Google Play Store, Google requires phone manufacturers to include other core modules such as Google Mobile Services (GMS). These services include Google apps (Maps, Youtube, etc.) and background services, also known as Google Play Services.

The Android ecosystem suffered a fragmentation problem as OEMs were unable to keep up with Google updates [33]. In response to the security issues underlying the fragmentation problem, Google moved the most critical components of Android to the Google Play Services bundle. This library receives automated updates from the Play Store without involving OEMs or users. Google Play Services has two core

components: i) a proprietary app that embeds the logic of the different services offered by Google, and ii) a client library that provides an interface to those services. Developers must include the client library in their apps when accessing Google-dependent services, including those regarding Wear. Figure 2 shows how the Google Play Services app interacts with the client library using standard inter-process communication (IPC) channels.

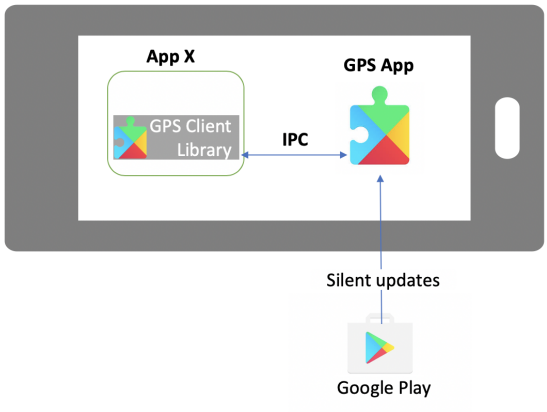

Figure 2: Google Play Services (GPS) architecture and update process.

As of March of 2020, Google provides 19 different packages[2] that allow developers to interface with all the Google Play Services like Google Analytics, Cloud Messaging, Mobile Ads, or Wear OS among others. In particular, the package *com.google.android.gms.wearable* gathers all the interfaces exposed for wearable apps, including the APIs that enable the communication between mobile and wearables apps. This package is commonly referred as the `Data Layer` API.

2.3 Data Layer

The `Data Layer` API provides IPC capabilities to apps. This API consists of a set of data objects, methods, and listeners that apps can rely on to send data using four types of abstraction:

1. `DataItem` is a key-value style structure that provides automatic synchronisation between devices for payloads up to 100KB. The keys are strings values, and the payload could be integers, strings or other 16 data types. The `DataClient` APIs offers support to send `DataItems` which are uniquely identified by a path (string value) in the system.

2. `Assets` are objects that support large binaries of data like images or audio. Assets are encapsulated into `DataItems` before being sent. The `Data Layer` takes care of transferring the data, bandwidth administration, and caching the binaries.

3. `Message` are short bytes of text message that can be used for controlling media players, starting intents on the wearable from the mobile, or request/response communication. The `MessageClient` object provides the APIs to send this type of asynchronous messages. Each message is also identified by a path in the same way as `DataItems`.

4. A `ChannelClient` offers an alternative set of API methods to send large files for media formats like music and video (in streaming as well) which save disk space over `Assets`. `ChannelClient` are also identified by a unique path.

The Wearable API also provides the callbacks to listen for events receiving one of these four data types. Table 1 shows a summary of these objects and their corresponding callbacks. We omit the list of API methods due to space constrains. The 16 data types supported by `DataItems` can be found in the API documentation.[3]

2.4 Mobile-Wear Communication

Once two devices are paired, a mobile and its companion apps can talk to each other through the `Data Layer` as long as they are signed with the same certificate. This is a restriction introduced for security reasons.

Apps can use the `Data Layer` to open synchronous and asynchronous channels over the wireless channel. Table 1 shows the channel type corresponding to each abstraction of the `Data Layer`.

The `MessageClient` (asynchronous API) exposes the methods to put a message into a queue without checking if the message ever reaches its destination. This abstraction encapsulates the context of messages into a single API invocation, for instance, destination and payload. In contrast, synchronous channels (`DataClient`, `ChannelClient`) provide transparent item synchronization across all devices connected to the network. Moreover, synchronous channels rely on many APIs to provide context to one transmission. From now on, we will use synchronous channels to explain the operation of the `Data Layer` as these are more complex than asynchronous.

The context of one transmission consists of: node identifier, channel type (table 1), channel path (string identifier), and the data that will be transferred. Node identifier correspond to string that represents a node in the Wear OS network. A channel path represent a unique address which identifies each open channel within a node. Finally the data is the payload of the transmission.

An app can create many channels of the same type to send different payloads to the companion app. Developers often

[2] https://developers.google.com/android/guides/setup

[3] https://tinyurl.com/y4dwopqk

Table 1: Map between the different data types and the available channels in the `Data Layer` API.

Data Type - Channel	Channel Type	Information	Listeners
`Messages` - `MessageClient`	Asynchronous/not-reliable	Bytes	OnMessageReceived
`DataItems` - `DataClient`	Synchronous/reliable	16 types	OnDataChanged
`Assets` - `DataClient`	Synchronous/reliable	Binaries	OnDataChanged
`Channel` - `ChannelClient`	Synchronous/reliable	Files	OnChannelOpened

use path patterns to create a hierarchy that matches the project structure to identify different channels. For instance, the path *example.message.normal* can be used to request a normal update, while the path *example.message.urgent* could indicate an urgent request.

To initiate a Mobile-Wear communication, the sender app needs to create the context of the channel through a sequence of APIs calls. Then, the Google Play Services app in the phone performs the transmission, handling the encapsulation, serialization, and retransmission (if needed). In the smartwatch, Google Play Services receives the communication and processes the data before handing it over to the wearable app. The receiver app implements a listener that captures events from Google Play Services. The listener could be defined in a background service or an activity where the data is finally processed.

2.5 Motivation Example

In this section, we describe an example of a data leak using the `DataItem` channel. Here, a wearable app sends sensitive information to the Internet after a mobile app transfers sensitive information through this channel. Listing 1 shows the mobile app sending the geolocation and a constant string to the companion wearable app.

First, the channel is created (line 4) with its corresponding path. Then the geolocation and a string "hello" are added to the channel in line 5, and 6 respectively. Finally, the app synchronizes all the aggregated data in one API call (`synchronizeData`) in line 7.

Listing 1: Simplified example of mobile app exfiltrating data to the companion app.

```
1 nodeID = getSmartwatchId()
2 location = getGeolocation()
3 text = "Hello"
4 channel = WearAPI.createChannel("path_x")
5 WearAPI.put(channel, "sensitive", location)
6 WearAPI.put(channel, "greetings", text)
7 WearAPI.synchronizeData(nodeID, channel)
```

Listing 2 shows how the wearable app receives this transmission with the location data.

Listing 2: Example of companion app exfiltrating sensitive data.

```
1  event = WearAPI.getSynchronizationEvent()
2  if (event.path == "path_x"){
3      data = parseEvent(event)
4      location = data["secret"]
5      hello = data["greetings"]
6      exposeToInternet(location)
7      exposeToInternet(hello)
8  }else if (event.path == "path_y"){
9      do_something_else
10 }
```

First, the app fetches the event from the channel using the `Data Layer` API (line 1). The developer uses a conditional statement to execute actions depending on the event's path. Paths are the only way to characterize events that trigger different data processing strategies when exchanging data through a channel. Here, *path_x* (lines 3 to 7) corresponds to the branch that handles the data sent by listing 1, which includes the geolocation. In this case, the geolocation is sent via a sink in the companion app to the Internet. The branch *path_y* is used to process a different event.

On the receiver side, it also possible to specify the channel path in the Manifest using an intent-filter. In this case, the service only receives events which path is equal to the path specified in the Manifest. However, it also possible to specify a path prefix and then trigger different branches in the code. For listener in activities, developers rely on indirect references to the path on the code, like the example.

3 Security Threats in Wearable Ecosystem

The exchange of sensitive data between mobile and wearable applications introduces risks in relation on how that data may be handled by both the mobile and/or the wearable app. In our case, we assume that the smartphone and smartwatch will contain sensitive information like Personal Identifiable Information (PII), contacts information, and biomedical data that could be exfiltrated either from the smartphone or from the smartwatch.

3.1 Threat Model

We identify the following security risks that arise from the transmission of PII in the mobile-wear ecosystem:

1. Re-delegation: The permission model in Android requires developers to declare the permissions of their mobile and wearable apps separately. This enables mobile and wearable apps to engage in colluding behaviors [27]. For instance, a mobile app that requests the `READ_CONTACTS` permission, can use the `Data Layer` APIs to send the contact information to a wearable application that does not have this permission. Similarly, a wearable application could share sensor data such as heart rate or other sensors with its corresponding mobile app without requiring access to the Google Fit permissions.

2. Wearable data leaks: Wear OS includes APIs to perform HTTP and other network requests to Internet facing services. This means that wearable apps have exactly the same capabilities to exfiltrate data as regular mobile apps. However, as already mentioned in Section 1, information flow tools available today only account for data leaks that happen directly via the mobile app (or via other apps in the case of collusion). As of today, there are no methods to detect data leaks through wearable interfaces.

3. Mobile data leaks: In a similar way, the mobile app could exfiltrate sensitive data leaked from the wearable app environment. An example of sensitive data unique to the wearable is the heart rate. A mobile application can pull this data and sent it over the network. Note that while this threat can be materialized through a permission re-delegation attack, it is not strictly bound to this attack. Instead, both apps can request permission to access specific sensitive data, but the taint is lost when data is transmitted from the companion to the mobile app.

4. Layout obfuscation: Developers are increasingly using obfuscation techniques to prevent reverse engineering and to shrink the size of their apps [17]. Obfuscation presents a challenge to information flows analysis when it modifies the signature of relevant classes and methods. In our case, the APIs from the `Data Layer` might be obfuscated, and we cannot merely look at the signatures of the API methods.

To the best of our knowledge, this is the first framework that models Mobile-Wear communication. As a consequence, current frameworks fail to detect the situations above. This happens either when developers are not following good coding practices or when miscreants intentionally try to evade detection mechanism that rely on data-flow analysis.

For simplicity, we do not discuss how permissions are assigned. However, we note that the Motivation Example in section 2.5 relates to a re-delegation attack when the companion app does not require the geolocation permission. While taint tracking tools are able to identify sensitive data flows in the mobile app, they can not propagate the tracking to the companion app. The simplest way to solve this is to consider the execution context of the mobile (sender) and companion (receiver) app as a single context. Thus, enabling us to reason about existing Mobile-Wear communication and to track non-sensitive message individually.

Note that a Mobile-Wear taint tracking needs to consider that data flows are combined in a single point when the sender transmits the DataItem, and it separates again when the receiver app parses the event. This is shown in Listing 1, where the `Data Layer` aggregates data (i.e., the geolocation and a constant value) into a single channel. Finally, we note that an attacker may use any other channel described in Section 2.3 to leak sensitive data, although the technical procedure will defer.

Next, we show how we address this problem for all channels.

4 Modeling Google Play Services

WearFlow expands the context of taint-tracking analysis from a single application to a richer execution environment that includes the wearable ecosystem (i.e., the Wear OS).

Mobile-Wear taint tracking presents a different set of constrains and characteristics than Inter-Application and Inter-Component communication analysis. In Wear OS, the communication between the smartphone and the wearable involves the mobile app, Google Play Services, and the wearable app. As Google Play Services library acts as a bridge between the two, we need to model its behavior to track information between the two apps.

As seen in the examples shown in Listings 1 and 2, wearable APIs are designed to send and receive data in batches. This means that developers first insert the different items they want to transmit between apps and then execute a synchronization API call. From a data analysis perspective, this means that multiple data flows join into a single point when an app invokes the synchronization API to send data. One possible solution would be to taint all the information exchanged. However, this overestimation would result in a high number of false positives. There is another challenge behind tracking individual data flows in Wear OS, i.e.: Google Play Services is not open source and it is implemented in native code, which makes the data tracking more difficult [28].

In order to track these flows, we have created a model of the `Data Layer` to generate a custom implementation of the wearable client library. To create the model, we manually inspected the wearable-APIs from the `Data Layer`, and built a sequence of possible invocations and the effect of these APIs on the context of the communication. This model allows us to extract the context of each communication, such as the

path and the data added into a *channel*. Then, we can use this information to replace the invocations to the original APIs with invocation to our instrumented APIs.

Note that we do not know the details of how Google Play Services implements the communication, but we do know the result of the communication, and we can reason about the context of communication by looking at relevant points where the apps invoke wearable APIs.

The result of our model is a mapping between the original methods from the `Data Layer` client library to a modified implementation template that facilitates the matching of individual data flows between apps. This modified implementation is generated as follows:

1. We identify all relevant classes from the gms-wearable library and generate custom signatures for each method.
2. For each app, we identify all invocations of synchronous and asynchronous APIs from the `Data Layer`. For each invocation we run a taint analysis to extract the context of the transmission. This involves:
   (a) Identifying the channel creation.
   (b) Searching the items that have been added into the channel variable (data sent across the channel).
   (c) Evaluating strings from the context (path and keys).
   (d) Generating custom API calls using the extracted context and corresponding method template.
   (e) Replacing original method invocation with a custom API invocation.

By doing this, we can simulate the propagation of data flows across apps on different devices while keeping the semantics of the different data flows intact. As an example, whenever we find a call to <*DataClient: putDataItem(PutDataRequest request)*>, the model will tell us that this is a synchronous communication which in sending a DataItem (encapsulated in the PutDataRequest). In this case, the model also specified that the `PutDataRequest` object required a previous API call that creates a channel, and other APIs that add data to the `DataItem`. We use this information to do a backward and forward inter-procedural analysis to extract such information. Finally, the model provides the rules to match entry and exit points once we have the results of the data flow analysis.

5 WearFlow

We design a pipeline of five phases that result in the detection of Mobile-Wear data leaks. Figure 3 shows a high-level overview of our system. Phase 1 converts the app to a convenient representation and extract relevant information. Phase 2 deobfuscates (if necessary) the Google Play Services client library and relevant app components. Phase 3 performs a context extraction and instrumentation for every invocation to a wearable API. Phase 4, runs the information flow analysis and export the results. Finally, we match data flows according to the model of the `Data Layer` in phase 5 to obtain all Mobile-Wear flows.

Phase 1: Pre-Processing

Android packs together the wearable and the mobile app into a single package file (namely, APK). WearFlow first splits both apps and then uses Soot to pre-process each executable separately. In particular, we convert the Dalvik bytecode into the Jimple Intermediate Representation (IR), and parse the relevant configuration files (e.g., the wearable and mobile app Manifests). Jimple simplifies the different program analysis techniques we use in the following phases.

WearFlow then searches for Wear OS components (services and callbacks as in Table 1), subject to an optional deobfuscation phase (Phase 2). We leverage the Manifests to understand the relationship between paths and services by looking at the intent filters declared as *WearableService*. We then inspect the Jimple to obtain all variables of the data types listed in Table 1, including those that appear in callbacks. These data types are used to open Mobile-Wear channels. We will instrument all these components as described in Phase 3.

Phase 2: Deobfuscation

We use a simple heuristic to detect if the app is obfuscated. First, we assume that all Mobile-Wear applications would use any of the methods from the classes of the `Data Layer` API shown in Table 1. Thus, we search for these methods in the client libraries of the APK. If no method is found, we consider the app may be obfuscated and perform a type signature brute-force search. This signature models the type of inputs and outputs of a function.

In addition to the type signature, we further look at local variables declared using system types in the method and compute their frequency per method (when the method is not a stub). The rationale behind including context from the method itself is to reduce the number of false positives when performing the signature search. The signature model uses only system types and abstract types from the `Data Layer` to generate the obfuscation-resilient signatures. We refer the reader to Section 7 for a discussion on our choices and how this may impact our results.

We extract signatures for all relevant methods that model Mobile-Wear IPC (see Section 2.3). Overall we extract 63 signatures capturing methods that exchange `Messages`, `DataItems`, `Assets` and `Channels`. We then search for methods in app's components that match against these signatures. When we find a match, we identify the corresponding wearable API of our interest. As we show in Section 6.4, we can

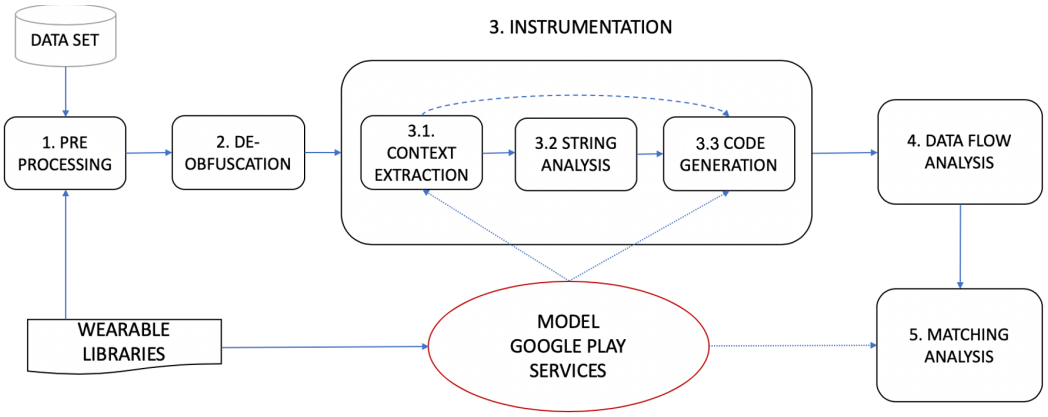

Figure 3: Overview of WearFlow.

identify all the methods used by the model with the above features.

Phase 3: Instrumentation

This phase aims at instrumenting the apps under analysis and it has three steps: context extraction, string analysis, and code generation. It takes as input a model of Google Play Services library. Our attempt to model this library is described in Section 4.

For the context extraction, we search invocations to wearable APIs that send or receive data in each of the components seen in the pre-processing step. Once that one API is identified, WearFlow performs an inter-procedural backward analysis to find the creation of the corresponding channel, and then a forward analysis to find invocations to APIs which add data into the channel. We then evaluate strings of relevant API methods; for instance, the method <*PutDataMapRequest.create(String path)*> for DataItems. For this we perform an inter-procedural and context-sensitive string analysis. For asynchronous APIs (MessageClient), the context extraction is limited to evaluate the variable which contains the channel path.

The next step is to instrument the app. On the one hand, we add our custom methods to the client library. In particular, we add the declaration of method that we use as entry/exit points in their corresponding classes. On the other hand, for each invocation to APIs methods acting as entry/exit points, we generate the corresponding invocation to our custom APIs. We use the output of the context extraction, string analysis, and the model of the `Data Layer` to generate such invocations. The resulting code will replace the invocations to the original methods in the wearable library.

The Listing 3 shows the instrumented code corresponding to the motivation example in Listing 1. This code replaces the lines [4 - 7] from the example. Note that the code below illustrates a notion of the instrumentation which is done in the Jimple IR.

Listing 3: Simplified instrumented code.

```
1 nodeID = getSmartwatchId()
2 text = "hello"
3 location = getGeolocation()
4 channel = WearAPI.createChannel("path_x")
5 WearAPI.syncString(nodeID, channel,
      "greetings", text)
6 WearAPI.syncString(nodeID, channel,
      "sensitive", location)
```

Phase 4: Data Flow Analysis

This phase performs data flow analysis of the Mobile-Wear ecosystem as a whole. First, we add callbacks from the Wear OS libraries as given by our model to enable data flow analysis across devices (see Section 2.3) . Note that we add sources and sinks that are not detected by state-of-the-art well-maintained projects in Android [2, 3]. More importantly, we add data wrappers that can capture how data flows propagate through objects of the `Data Layer`. One limitation of existing data flow frameworks like FlowDroid [3] is that they use simplified wrapper models that only abstract the semantics of the Android framework for well-known cases.

The next step is to compute the call graph of both apps and perform a taint tracking analysis as a single context. We do this by first running the taint analysis separately on each app and then matching the results using the instrumented APIs as connectors between data flows. We add the APIs that send data as sinks (wearable-sinks) and the APIs that receive data as source (wearable-sources). Then, we also add the wearable-sources and the wearable-sinks in the list of sources and sinks.

Finally, WearFlow reports the results of the taint tracking. At this point, we are only interested in data flows with wearable-sources or wearable-sinks. It is worth noting that taint analysis still detects data flows that end in a non-wearable

sink, but they are irrelevant to the matching step. Our approach is agnostic to the underlying method used to compute data flows. We refer to Section 6.1 for implementation details.

Phase 5: Matching Analysis

The final step consists of matching exit points with entry points; that is to say, wearable-sinks with wearable-sources. We consider three values to match data flows: channel path, API method, and key. If the value of the path or key could not be calculated during the context extraction, then we use a wildcard value that matches any value. To match the API methods, we built a semantic table that provides information to match wearable-sinks with its corresponding wearable-sources. We present a summary in the Table 2 due to space limitation. The table contains thirty-four entries in total and it can be found in the project repository.

6 Evaluation and Results

We evaluate WearFlow against other Android information flow analysis tools currently available and perform a large-scale analysis of 3.1K Android APKs with wearable components looking for sensitive data leaks. Our evaluation uses a specifically crafted set of apps that presents different data exfiltration cases using the `Data Layer` API. We conduct our experiments on a machine with 24 cores Intel Xeon CPU E5-2697 v3 @ 2.60GHz and 32 GB of memory.

6.1 Implementation

WearFlow relies on the Soot framework [29] to perform the de-obfuscation, context extraction and app instrumentation (Phases 2, 3.1 and 3.3). Our implementation leverages FlowDroid [3] with a timeout of 8 minutes per app for the information flow analysis (Phase 4) and Violist [19] for the string analysis (Phase 3.2). We use FlowDroid and Soot because previous works report that they provide a good balance between accuracy and performance on real-world apps [6, 24, 25]. We customize FlowDroid to run on wearable apps by adding callbacks from the Wear OS libraries and by extending the SuSi [2] sources and sinks as discussed in Section 5. We also perform several optimizations to Violist to reduce the execution time while keeping the accuracy for the APIs we were interested in. For instance, we reuse the control flow graph generated by Soot, and we limit the evaluation of the strings to relevant methods. With this, WearFlow adds, overall, around 6000 LoC to these frameworks. We make the implementation of WearFlow open source in `https://gitlab.com/s3lab-rhul/wearflow/`.

6.2 Evaluation results

As community lacks on a test suite that include Mobile-Wear information flows for Android, we create `WearBench`[4]. WearBench has 15 Android apps with 23 information flows between the mobile app and the wearable companion (18 of them sensitive). Our test suite covers examples of all APIs from the `Data Layer`. It also contains challenges for the instrumentation like field sensitivity, object sensitivity, and branch sensitivity for listeners.

Our suite is inspired by Droid-Bench[5] and ICC-Bench[6], which are standard benchmarks to evaluate data flow tools. Note that these benchmarks evaluate the effectiveness of the taint analysis, and some Inter-App communication cases using ICC methods. Instead, we are evaluating Inter-App communication between mobile and wearable apps (using the `Data Layer` API). Therefore, we cannot use these benchmarks alone to evaluate WearFlow. In our evaluation, we compare our results against FlowDroid. For this, we add the Data Layer APIs as sources and sinks, execute FlowDroid on both the mobile and wearable companion and look for matches. We run FlowDroid with a context sensitive algorithm twice: first with high precision, we set the access path length to 3. Then we reduce the precision by setting the access path to 1. With this, FlowDroid truncates taints at level 1. This configuration increases the number of false positives but catches situations where FlowDroid fails to propagate taint abstraction correctly.

Table 3a shows the result of our evaluation against the test suite. WearFlow detects all the 18 exfiltration attempts with two false positives. These two false positives stem from a branching sensitivity issue present in FlowDroid, which result in false positives during the data flow analysis (Phase 4). Conversely, FlowDroid with high precision detects 6 out of 18 exfiltrations — these are only matches communicating with the MessageClient API. This is because FlowDroid fails to propagate taints on complex objects from the `Data Layer`. When reducing the precision, FlowDroid identifies matches with `MessageClient` and `DataClient` but still fails to identify sensitive flows with the `ChannelClient` API. In this case, FlowDroid produces 12 false positives. This results from an overestimation of taints that uses `DataItem`.

> Our results show that WearFlow performs better than FlowDroid by a clear margin. This exemplifies how the modeling, instrumentation and matching analysis can improve information flow analysis in wearable applications.

6.3 Analysis of Real-World Apps

We use WearFlow to search the presence of potential data leaks on around 3.1K real-world APKs available in the Google

[4] `https://gitlab.com/s3lab-rhul/wearbench/`
[5] https://github.com/secure-software-engineering/DroidBench
[6] https://github.com/fgwei/ICC-Bench

Table 2: Selection of Sink-Source matches in Data Layer API. A full list can be found in WearFlow repository.

Library	Wearable sink signature	Wearable source signature
DataClient	DataClient: void putString(String,String)	DataMap: String getString(String)
MessageClient	MessageClient: Task sendMessage(String,String,byte[])	MessageEvent: byte[] getData()
DataClient	DataClient: void putAsset(String,Asset)	DataMap: Asset getAsset(String)
ChannelClient	ChannelClient sendFile(Channel,Uri)	Task receiveFile(Channel,Uri,Boolean,String)

Table 3: Summary of our results.

(a) Results for our test-suite between WearFlow and FlowDroid. HP = high precision, LP = low precision.

	Existing Data Flows			Found Data Flows		
Library	**Apps**	**Total**	**Sensitive**	**WearFlow**	**Flowdroid-HP**	**Flowdroid-LP**
DataItem	9	16	13	14 (1 FP, 1 FN)	0 (13 FN)	22 (6 FP)
Message	5	6	4	5 (1 FP)	6	10 (6 FP)
Channel	1	1	1	1	0 (1 FN)	0 (1 FN)

(b) Results for real-world apps (* sensitive data flows).

	Apps	**APKs**
Number of apps	220	3,111
With flows	47	293
With sensitive *	6	50

Play Store (downloaded from AndroZoo [1]). From an initial set of 8K APKs, around 5K refer to standalone (only wear) APKs, and 3.1K include mobile and wearable components. We execute WearFlow against this set which corresponds to 220 different package names. Table 3b shows a summary of the results. Note that the dataset contains multiple versions of the same app. Thus, we refer to apps as APKs with unique package name.

Figure 4 shows a summary of the different APIs used as exit/entry points of sensitive data flows. Although we found the occurrence of the `ChannelClient` API in the dataset, we did not find any case where this API was used to send sensitive information. WearFlow identifies sensitive information flows that include the transmission of device contacts (via Cursor objects), location, activities, and HTTP traffic. We also found that in several occasions sensitive data ended up in the device logs (17% of overall sinks) or SharedPreference files (20%). A more detailed analysis of these flows for a selection of apps is provided in Section 6.5.

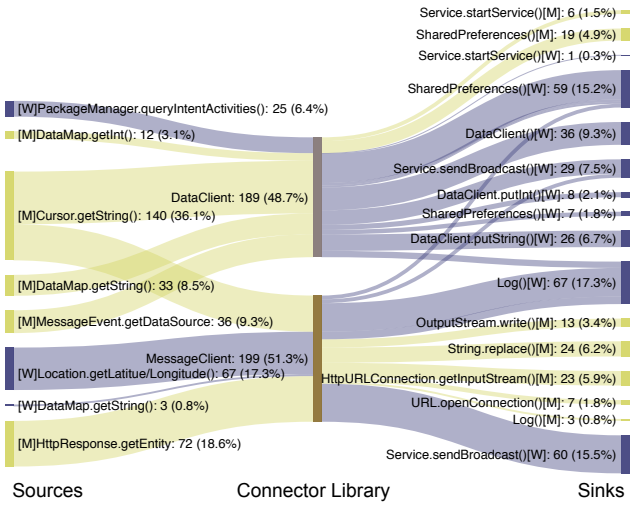

Figure 4: Sensitive information flows found. [M] refers to Android and [W] refers to Wear OS.

WearFlow is capable of finding 4,896 relevant data flows in all the analyzed APKs. Out of those, 388 relate to Mobile-Wear sensitive information flows in 6 apps (or 50 APKs, when considering all versions and platforms). The results indicate that 70% of the flows are from the mobile to the wear platform, while 30% are wear to mobile.

6.4 Applicability

We next see how we perform when dealing with obfuscation and what is the runtime overhead.

Obfuscation. WearFlow detects 282 obfuscated APKs in the dataset. The deobfuscation phase successfully unmangles all these APKs. On the one hand, we find 71 data flows using the `Data Layer` within these APKs. WearFlow did not find relevant APIs in 651 APKs. This can either because these APIs are not used at all or because developers use more complex obfuscation techniques. We discuss this in Section 7.

On the other hand, we find around 2K non-obfuscated APKs in our dataset. WearFlow instruments 4.8 components on average per APK (excluding library classes). From all the wearable APIs, around 48% are `DataClient` APIs, 51% |MessageClient| APIs, and less than 1% `ChannelClient` APIs. This number shows that developers are aggregating multiple data into `DataClient` before synchronizing `DataItems` and shows the benefits of instrumenting the APKs to track individual data flows.

Running time. Running our tool on the real-world dataset took 115 hours. WearFlow analyzes over 95% of the APKs before the 8 minutes timeout lapses. The average time per APK is 3.1 minutes. Note that wearable apps are considerably smaller in size than mobile apps, and WearFlow evaluates most wearable apps in less than 1 minute. The time distribution per phase analysis is the following: pre-processing 13%, string analysis 9%, deobfuscation and instrumentation 2%, and data flow analysis 76%.

WearFlow failed to complete the analysis for a small number of APKs. In most cases, this is due to unexpected bytecode that Soot fails to handle, errors while parsing APK resources, or because the analysis reached an extended timeout.

Overall, WearFlow extracts data flows for an additional of 282 Mobile-Wear APKs. Without the deobfuscation phase, these flows would not otherwise be extracted. The deobfuscation phase only takes 2% of the running time.

6.5 Case Studies

This section describes issues found by WearFlow in specific apps in relation to the threat model presented in Section 3.

Companion Leak. We first study the case of Wego (*com.wego.android*), a travel app to book flights and hotels with more than 10 million downloads. We find a sensitive flow that starts in the watch with source *getLatitude()* from the Location API, and it is sent to the mobile app with the `MessageClient` API using the path "request-network-flights". Then, the mobile app sends out this data through URL using the *HttpURLConnection* object, and write it to a file system using the *java.io.OutputStream* class. In this case, both the wearable and the mobile declare the location permission in the Manifest. However, this alone is not enough to comply with the guidelines.[7] In this case, the wear app must send the user to the phone to accept the permission. This case shows that it is possible to bypass the permission system using `Data Layer` APIs.

Permission re-delegation. Venom (*fr.thema.wear.watch.venom*) is a Watch Face customized for a watch user interface. The mobile version of this app uses the *android.database.Cursor* class to store sensitive information such as the call history or unread messages in a database. The app aggregates all information in a `DataItem` object and synchronizes it with the wearable app. However, the wearable app does not declare the relevant permissions. Interestingly, the string analysis has been key to uncover the type of information the app is retrieving from the database and trace it back to API sources that relate to the sensitive information discussed (e.g., missed calls and text messages).

Sensitive Data Exposed Finally, we observe evidence of apps exposing sensitive data through a wide range of sinks, including Android Broadcast system and Shared Preferences. For instance, Talent (*il.talent.parking*) — which is used for car parking — reads data related to the last parking place and its duration from a database and synchronizes it with the watch using a `DataItem`. Then the wearable app writes the data to Shared Preferences. Another example is the app *com.mobispector.bustimes*, which shows bus and tram timetables, and has more than 4 million downloads. The app reads data from an HTTP response, then send it to the wearable through the `MessageClient` API, and finally executes a system Broadcast exposing the content of the HTTP response.

All these cases show how developers leverage the `Data Layer` API to send sensitive information. While it is unclear whether or not these cases intentionally use Google Play services to hinder the detection of data leakages, we see that WearFlow is effective at exposing bad practices that can pose a threat to security and privacy.

7 Limitations

This section outlines the limitations of our work. These may arise from WearFlow's implementation or the dataset we used.

Data Transfer Mechanisms. WearFlow inherits the limitations of static analysis, i.e.: it is subject to constrains of the underlying flow and string analysis techniques. This means that it fails to match data flows with native code, advanced reflection, or dynamic code loading. Still, WearFlow can be used together with other frameworks [22,31] that handle these issues to improve the accuracy of the analysis.

WearFlow considers obfuscation while performing the analysis of apps. There are four trivial techniques and seven non-trivial techniques commonly used in the wild, according to a large scale study of obfuscation in Android [17]. WearFlow type-signature deobfuscator is resilient to all trivial techniques and five non-trivial, but it fails to deobfuscate APKs with class or package renamed and reflection. As mentioned before, WearFlow did not found relevant APIs in 20% (651) of the APKs, many of which correspond to APKs with these obfuscation techniques.

A more robust deobfuscation technique should rely on other invariant transformations such as the hierarchical structure of the classes and packages. Many methods of the Data Layer library are stubs, therefore using features of the method body will not improve the accuracy drastically. Additionally, one could obfuscate those features (e.g., using reflection together with string encryption) introducing false negatives.

[7] https://developer.android.com/training/articles/wear-permissions

In our case, we have successfully leveraged system types to disambiguate methods with the same signature. We chose to only use system types as they are less prone to obfuscation than other data types and have helped reducing the number of false positives. One could take into account the threat model, and chose to use a more coarse type of signatures when certain traits (e.g., reflection) appear in the app.

Branching. Another potential source of false negative data flows stems from the backward analysis, in the context extraction. WearFlow stops the backtracking when encounters a definition of a channel, but this definition could be part of the branch of a conditional statement. Depending on the scope of the variable, the channel could be defined in another method or even another component. Finally, if the string analysis is unable to calculate the value of a key or path (e.g., when there are multiples values due to branching) we use a wildcard value, i.e.: we match any string. This means that the matching step will overestimate the potential flows between the entry and the exit point.

Dataset. Our dataset is limited to 3,111 APKs and 220 package names after considering different APK versions. There are more than 220 apps available for Wear OS, however, identifying them is a challenging task. Google Play does not offer an exhaustive list of apps with Wear OS components, nor it is always featured in the description of the app. This restriction limits our ability to query Wear OS apps in Google Play. Furthermore, datasets like Androzoo do not provide information about whether a app has wearable components or not. Thus, we need to download apps as the rate limit allows. Given the low density of these kind of apps in the overall set of Android apps, the amount of apps that can be obtained this way is very limited.

Model Accuracy. The precision of the analysis also depends on the accuracy of the `Data Layer` model. The `Data Layer` model used by WearFlow replicates the `Data Layer` model, as described in Google's Wear OS documentation. If the `Data Layer` APIs were to transfer data trough undocumented components of the OS or even through the cloud (e.g., via backups), WearFlow would not detect such flows. Also, our model is based on Wear OS versions 1 and 2. Wear OS under active development. Thus, any new APIs introduced in future versions will need to be modeled.

8 Related Work

Mobile-Wear communication can be seen as a kind of inter-app communication where one of the apps is being executed in a wearable device. Several works have focused on app collusion detection [4, 5, 18, 23, 26, 34]. These works model ICC methods to identify sensitive data flows between applications running on the same device. WearFlow complements these, extending the analysis of these apps into the Mobile-Wear ecosystem, increasing the overall coverage of these solutions to all current app interactions in the Android-Wear OS ecosystem. One may argue that these tools could be extended to cover for Wear OS interactions. As an example, works such as DialDroid [6] uses entry and exit points to match ICC communication between mobile apps. In our case, we consider wearable APIs as sources and sinks, which could be easily replicated in DialDroid. However, These APIs aggregate multiple data into a single API call, and we need to match data types on the sender and receiver side, which would lead to inaccuracies in DialDroid and many other tools [11].

ApkCombiner [20] combines two apps into one allowing to run taint tracking on a single app. This approach does not allow us to reason about individual items aggregated into a single API call.

There has been a recent interest of the community in expanding the scope of data tracking to more platforms outside the Android ecosystem. Zou *et al.* [35] studied the interaction of mobile apps, IoT devices and clouds on smart homes using a combination of traffic collection and static analysis. They discoverd several new vulnerabilities and attacks against smart home platforms. Berkay *et al* proposed a taint tracking system for IoT devices [7]. WearFlow could have been implemented following the same approach (analysing WiFi and Bluetooth communications between Android and Wear OS). This would have required us to reverse the different communication protocols and data exchanged in both wireless protocols. Our approach is simpler, and doesn't require additional hardware to execute.

9 Conclusion

In this work, we have presented WearFlow, a static analysis tool that systematically detects the exfiltration of sensitive data across the Mobile-Wear Android ecosystem. WearFlow augments the capabilities of previous works on taint tracking, expanding the scope of the security analysis from mobile apps to smartwatches. We addressed the challenge of enabling inter-device analysis by modeling Google Play Services, a proprietary library. Our analysis framework can deal with trivial obfuscation and most of the non-trivial obfuscation techniques commonly used in the wild.

We have created WearBench, the first benchmark for analyzing inter-device data leakage in Wear OS. Our evaluation shows the effectiveness of WearFlow over other approaches. We also analyze apps in Google Play. Our results show that our system scales and can uncover privacy violations on popular apps, including one with over 10 million downloads. As a future work, we want to extend our deobfuscation phase to cover additional forms of obfuscation (e.g., the two — out of seven — non-trivial obfuscations we discuss), and extend the scope of our analysis to the entire Google Play app market.

10 Acknowledgements

This research has been partially sponsored by the Engineering and Physical Sciences Research Council (EPSRC) and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 468–471, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013*, 114:108, 2013.

[3] 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 *Acm Sigplan Notices*, volume 49, pages 259–269. ACM, 2014.

[4] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering*, 41(9):866–886, 2015.

[5] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. Android inter-app communication threats and detection techniques. *Computers Security*, 70:392–421, 2017.

[6] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[7] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th {USENIX} Security Symposium*, pages 1687–1704, 2018.

[8] Jagmohan Chauhan, Suranga Seneviratne, Mohamed Ali Kaafar, Anirban Mahanti, and Aruna Seneviratne. Characterization of early smartwatch apps. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2016.

[9] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–12, 2015.

[10] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. Is the data on your wearable device secure? an android wear smartwatch case study. *Software: Practice and Experience*, 47(3):391–403, 2017.

[11] Karim O Elish, Danfeng Yao, and Barbara G Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*, 2015.

[12] HP Fortify. Internet of things security study: smartwatches. Accessed March 2020, 2015. https://www.ftc.gov/system/files/documentspublic_comments/2015/10/00050-98093.pdf.

[13] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. *arXiv preprint arXiv:1905.02713*, 2019.

[14] Gartner. Gartner says global end-user spending on wearable devices to total $52 billion in 2020. Accessed March 2020, 10 2019. https://perma.cc/MR8J-PUUK.

[15] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[16] Rohit Goyal, Nicola Dragoni, and Angelo Spognardi. Mind the tracker you wear: a security analysis of wearable health trackers. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 131–136, 2016.

[17] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering*, pages 421–431, 2018.

[18] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[19] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 661–672. ACM, 2015.

[20] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference*, pages 513–527. Springer, 2015.

[21] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[22] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329, 2016.

[23] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Yao, Karim O Elish, and Barbara G Ryder. Mr-droid: A scalable and prioritized analysis of inter-app communication risks. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 189–198. IEEE, 2017.

[24] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341, 2018.

[25] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186, 2018.

[26] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, pages 1–10, 2014.

[27] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[28] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342, 2016.

[29] 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*, pages 214–224. IBM Corp., 2010.

[30] He Wang, Ted Tsung-Te Lai, and Romit Roy Choudhury. Mole: Motion leaks through smartwatch sensors. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 155–166, 2015.

[31] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[32] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[33] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.

[34] Mengwei Xu, Yun Ma, Xuanzhe Liu, Felix Xiaozhu Lin, and Yunxin Liu. Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web*, pages 143–152, 2017.

[35] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th {USENIX} Security Symposium*, pages 1133–1150, 2019.