Uncovering Intent based Leak of Sensitive Data in Android Framework

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

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

Haipeng Cai
Washington State University
Pullman, Washington, USA
haipeng.cai@wsu.edu

ABSTRACT

To prevent unauthorized apps from retrieving the sensitive data, Android framework enforces a permission based access control. However, it has long been known that, to bypass the access control, unauthorized apps can intercept the `Intent` objects which are sent by authorized apps and carry the retrieved sensitive data. We find that there is a *new* (previously unknown) attack surface in Android framework that can be exploited by unauthorized apps to violate the access control. Specifically, we discover that part of `Intent` objects that are sent by Android framework and carry sensitive data can be received by unauthorized apps, resulting in the leak of sensitive data. In this paper, we conduct the *first* systematic investigation on the new attack surface namely the `Intent` based leak of sensitive data in Android framework. To automatically uncover such kind of vulnerability in Android framework, we design and develop a new tool named `LeakDetector`, which finds the `Intent` objects sent by Android framework that can be received by unauthorized apps and carry the sensitive data. Applying `LeakDetector` to 10 commercial Android systems, we find that it can effectively uncover the `Intent` based leak of sensitive data in Android framework. Specifically, we discover 36 exploitable cases of such kind of data leak, which can be abused by unauthorized apps to steal the sensitive data, violating the access control. At the time of writing, 16 of them have been confirmed by Google, Samsung, and Xiaomi, and we received bug bounty rewards from these mobile vendors.

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**.

KEYWORDS

Android; Static Analysis; Intent; Vulnerability

---

*The corresponding author.

1 INTRODUCTION

Smartphones have become an inseparable part of our daily lives, and they have access to a great deal of sensitive sensor data (e.g., GPS coordinates), private user data (e.g., contacts), and personal identifiable information (e.g., IMEI) [55]. To prevent these sensitive data from being retrieved by unauthorized apps, Android employs a permission based access control [24, 28, 29, 57, 63]. Specifically, Android provides sensitive framework APIs for apps to retrieve the sensitive data, and the APIs enforce permission check on their calling apps to examine whether the apps have been granted with the required permissions to retrieve the sensitive data. Only the authorized apps, having gained the necessary permissions, can successfully call framework APIs to retrieve the sensitive data.

However, it has long been known that unauthorized apps can abuse `Intent` [3], an inter-process communication mechanism, to violate the permission based access control in Android framework and result in the leakage of sensitive data to unauthorized apps [30, 31, 34, 41, 44, 45, 49, 58, 60]. In detail, as shown in Figure 1, the authorized app, having gained the required permissions, calls sensitive framework APIs to retrieve the sensitive data and then uses `Intent` to send out the retrieved sensitive data. If the `Intent` object carrying the sensitive data can be received by the unauthorized app, then the unauthorized app can get the sensitive data without the need of requesting and gaining the required permissions. In this scenario, since the `Intent` object carrying the sensitive data is sent by apps, we call it the `Intent` based leak of sensitive data in apps (short for Leak$_{App}$).

A New Attack Surface. We discover a new attack surface that can be exploited by unauthorized apps to retrieve the sensitive data based on the observation that Android framework also uses `Intent` to conduct inter-process communication for various purposes, such as notifying apps about occurrences of special system events [22]. Specifically, as presented in Figure 1, if the `Intent` object sent by Android framework, carries sensitive data and can be received by an unauthorized app, it can obtain the sensitive data in the `Intent` object, violating the access control. In this scenario, since the `Intent`
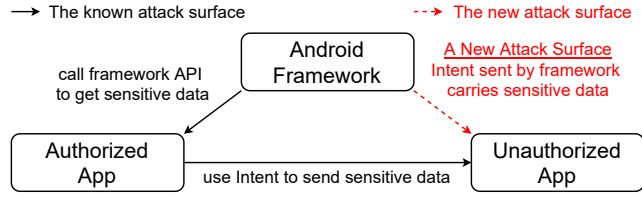
Figure 1: Attack surfaces that can be exploited by the unauthorized app to retrieve the sensitive data in Android framework.

object is sent by Android framework, we call it the `Intent` *based leak of sensitive data in Android framework* (short for Leak$_{Frm}$).

Although various work [30, 31, 34, 41, 44, 45, 49, 58, 60] has been conducted to detect Leak$_{App}$, their approaches cannot be used to identify Leak$_{Frm}$ directly, because none of them has investigated the sensitive data in Android framework so that they cannot recognize the `Intent` object that carries sensitive data and is sent by Android framework.

In order to fill the gap, in this paper, we conduct the *first* systematic investigation on Leak$_{Frm}$, and design and develop a new tool named `LeakDetector` to automatically uncover such kind of vulnerability. More precisely, `LeakDetector` first analyzes sensitive framework APIs to find the entities (i.e., fields and classes defined in Android framework) that store sensitive data, and we name such entities as sensitive entities. Then, `LeakDetector` analyzes `Intent` objects sent by Android framework. This process consists of three steps. First, to analyze sensitive framework APIs and the `Intent` objects in Android framework, `LeakDetector` builds the callgraph of Android framework by performing static analysis on JAR files and APK files of the framework. Second, based on the callgraph, `LeakDetector` finds sensitive fields defined in Android framework, which store the sensitive data, and sensitive classes whose instances encapsulate the sensitive data by conducting data flow analysis on return values of sensitive framework APIs. Third, based on the identified sensitive entities, to uncover the `Intent` based leak of sensitive data in Android framework, `LeakDetector` determines whether the `Intent` objects sent by the framework can be received by unauthorized apps and carry the sensitive data (i.e., values of sensitive fields or instances of sensitive classes) by performing data flow analysis on the `Intent` objects.

When implementing data flow analysis, we address the challenge of tracking data in the collection objects [13] (e.g., `List` objects, `Set` objects, and `Map` objects) by modelling the element-adding methods (e.g., `List.add`, `List.addAll`). Specifically, we perform data flow analysis on these methods' arguments to track every data added to the collection objects (detailed in §4.3).

We use `LeakDetector` to discover Leak$_{Frm}$ in 10 commercial Android systems on 2 latest Android versions (i.e., Android 11 and 12) from 5 mobile vendors, including Google, Samsung, Xiaomi, Oneplus, and Vivo. In total, we uncover 36 exploitable cases of Leak$_{Frm}$, which can be taken advantage by unauthorized apps to retrieve the sensitive data, violating the access control in the framework. We have reported these cases to the corresponding mobile vendors. At the time of writing, 16 of them have already been confirmed, and we received bug bounty rewards from Google, Samsung, and Xiaomi.

In summary, we make the following contributions:

- To the best of our knowledge, we are the *first* to reveal and investigate `Intent` based leak of sensitive data in Android framework.
- We design and develop `LeakDetector`, a new tool to automatically uncover the `Intent` based leak of sensitive data in Android framework. The source code of `LeakDetector` is available at https://github.com/moonZHH/LeakDetector.
- We extensively evaluate the performance of `LeakDetector` by applying it to 10 commercial Android systems from 5 mainstream mobile vendors. In total, we discover 36 exploitable cases of the `Intent` based leak of sensitive data in Android framework, which can be abused by unauthorized apps to violate the access control to retrieve the sensitive data.

2 BACKGROUND

This section introduces the access control in Android framework for restricting the retrieval of sensitive data in §2.1. To explain Leak$_{Frm}$ (see §3), we provide the necessary knowledge about `Intent` in §2.2.

2.1 Access Control

Android framework provides interfaces (i.e., sensitive framework APIs) for apps to retrieve the sensitive data. To prevent unauthorized apps from accessing the sensitive data, Android framework enforces access control on these sensitive APIs, such as permission check, UID check, and User ID check [24, 28, 29, 32, 57, 63]. Since the framework mainly employs permission check to protect `Intent` [43] (see §2.2), in this section, we introduce the permission based access control in Android framework.

- **Permission based Access Control**. It requires apps to gain the necessary permissions to retrieve the sensitive data (e.g., device ID) from Android framework [20].

```
// class "com.android.phone.PhoneInterfaceManager"
                                        required permission
01 public String getDeviceId(*) {              ↓
02    if (mContext.checkPermission(READ_PRIVILEGED_PHONE_STATE,*, *)) {
03        return PhoneFactory.getPhone(0).getDeviceId();
04    } // if apps have been granted with the required permission, return device ID
05    return null; // if apps do not have the required permission, return null
06 } /* irrelevant code is omitted */
```

Figure 2: An example of permission based access control enforced on the sensitive framework API for protecting the sensitive data.

More specifically, when an app invokes sensitive framework APIs to retrieve the sensitive data, these APIs call permission-checking methods [28, 29, 57] (e.g., `Context.checkPermission` listed in Table 1) to enforce permission check on the app. For example, as shown in Figure 2, when an app invokes the sensitive framework API `getDeviceId` to retrieve the device ID, the API calls the permission-checking method `checkPermission` in Line 2 to examine whether the calling app has been granted with the required permission `READ_PRIVILEGED_PHONE_STATE`. If so, the API returns the sensitive data (i.e., device ID) to the app in Line 3.

2.2 Intent

`Intent` is an inter-process communication mechanism in Android [3]. An `Intent` object is a message sent by Android framework or

Table 1: A partial list of permission-checking methods.

Class	Method
Context	checkPermission(*), enforcePermission(*) checkCallingPermission(*), enforceCallingOrSelfPermission(*) checkCallingOrSelfPermission(*), enforceCallingPermission(*)
ActivityManagerService	checkPermission(*), checkCallingPermission(*) enforcePermission(*), enforceCallingOrSelfPermission(*)
PermissionManagerService	checkPermission(*), checkUidPermission(*)

apps to request an action from its recipients, i.e., app components (including activities, services, content providers and broadcast receivers) [12]. Android framework provides APIs to create and send `Intent` objects. We list some necessary APIs in Table 2 for explaining $Leak_{Frm}$ in §3, and detail their functionality as follows.

Table 2: A partial list of Intent related APIs.

| Functionality	API
Set Intent action.	Constructors of Intent, i.e., Intent(ACTION_*) Intent.setAction(ACTION_*)
Set Intent recipient.	Constructors of Intent, i.e., Intent(ACTION_*, "class name") Intent.setClass(*) / setClassName(*) / setComponent(*)
Set Intent data.	Intent.putExtra(data) / putParcelableArrayListExtra(data)
Send Intent.	Context.startActivity(intent) / sendBroadcast(intent) PendingIntent.getActivity(intent) / getBroadcast(intent)

• **Setting Intent Action**. The `action` of an `Intent` object is a string that provides an abstract description of the action to be performed by its recipients [12]. Android framework provides APIs to set the action of an `Intent` object, such as the API `setAction` defined in the class `Intent`, which takes the action string as its argument.

• **Setting Intent Recipient**. The `Intent` object whose recipient is specified is called an explicit `Intent` object [12], and it can only be received by the specified recipient, i.e., the specified app component. Android framework provides APIs to specify the recipient of an `Intent` object, such as the APIs `setClass`, `setClassName`, and `setComponent` defined in the class `Intent`. When calling these APIs, the package name, class name, `Class` object [14], or `ComponentName` object [6] of the specified app component is passed to these APIs.

The `Intent` object whose recipient is unspecified is called an implicit `Intent` object [12], and it can be received by multiple recipients, i.e., multiple app components. To receive an implicit `Intent` object, app components need to declare that they can perform the action of this `Intent` object by setting the `intent-filter` tag in the manifest files that registered the components or programmatically setting the components' `IntentFilter` objects [12].

• **Setting Intent Data**. Since `Intent` is commonly used to share data across processes [3], to set the data carried by an `Intent` object, Android framework provides a serials of `putExtra` APIs, which take the transferred data as their arguments.

• **Sending Intent**. The framework provides various APIs to send an `Intent` object to its recipient, such as the APIs `startActivity` and `sendBroadcast` defined in the class `Context`, `getActivity` and `getBroadcast` defined in the class `PendingIntent`. When calling some of these `Intent` sending APIs (e.g., `sendBroadcast`), the permission requirement for the recipient to receive the `Intent` object
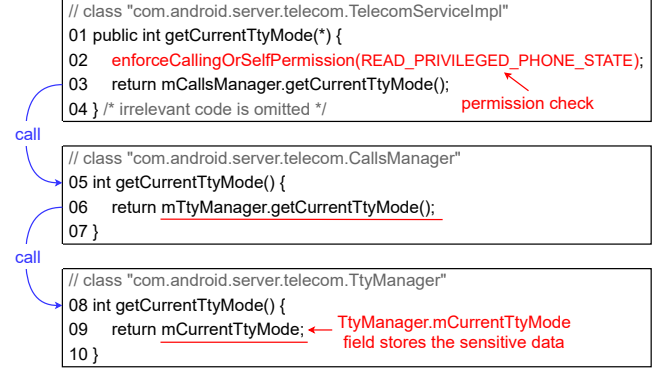
Figure 3: A sensitive field in Android framework.
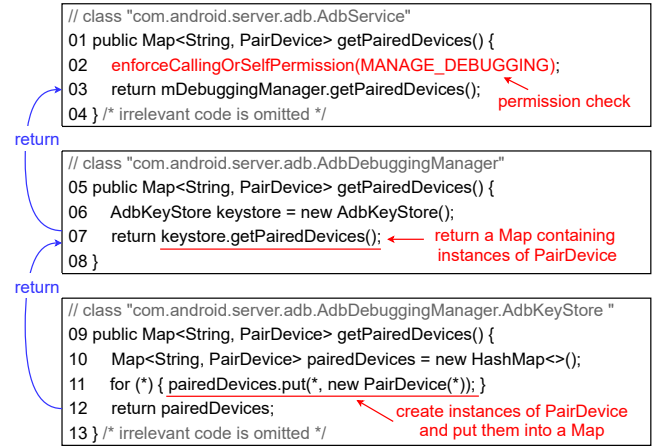
Figure 4: A sensitive class in Android framework.

can be set, and it is taken as the APIs' arguments. Once set, only the recipients, whose hosts (e.g., the apps that register the recipients) have been granted with the required permissions, can receive the `Intent` object at runtime.

3 INTENT BASED LEAK OF SENSITIVE DATA

This section investigates the sensitive data in Android framework in §3.1. Then, we explain $Leak_{Frm}$, introduce two types of $Leak_{Frm}$, and provide the threat model in §3.2. After that, for each type of $Leak_{Frm}$, we demonstrate a motivating example in §3.3.

3.1 Sensitive Entity in Android Framework

Since sensitive framework APIs retrieve the sensitive data and return the data to their callers (as introduced in §2.1), we study the implementations of these APIs to investigate the sensitive data in Android framework. Specifically, we discover two types of entities in Android framework that store the sensitive data, including the sensitive field and the sensitive class defined in Android framework.

• **Sensitive Field in Android Framework**. A few sensitive framework APIs (e.g., `TelecomServiceImpl.getCurrentTtyMode`) access the fields of Java classes defined in Android framework to retrieve

the sensitive data and return the data to their callers. We call these fields, storing the sensitive data, *sensitive fields*.

For example, Figure 3 shows the simplified code snippet of the API `TelecomServiceImpl.getCurrentTtyMode`. This API retrieves the sensitive data (i.e., the sensitive phone state information about the enabled teletypewriter mode on device [9]) from the sensitive field `TtyManager.mCurrentTtyMode` in Android framework. More specifically, to access the sensitive data, the API `getCurrentTtyMode` invokes the method `CallsManager.getCurrentTytMode` in Line 3, which invokes the method `TtyManager.getCurrentTtyMode` in Line 6 to access the sensitive field `mCurrentTtyMode` storing the sensitive data in Line 9. In order to prevent the sensitive data from being retrieved by unauthorized apps that have not been granted with the permission `READ_PRIVILEGED_PHONE_STATE`, the sensitive API enforces permission check in Line 2.

• **Sensitive Class in Android Framework**. Other sensitive framework APIs (e.g., `AdbService.getPairedDevices`) encapsulate the sensitive data using instances of Java classes defined in Android framework and return these instances to their callers. We call these classes, encapsulating the sensitive data, *sensitive classes*.

For example, Figure 4 shows the simplified code snippet of the API `AdbService.getPairedDevices`. This API internally creates instances of the sensitive class `PairDevice` to encapsulate the sensitive data (i.e., the sensitive information about the paired devices for wireless ADB debugging [7]) retrieved from a system file. More specifically, in Line 11, instances of the sensitive class `PairDevice` are created and put into a collection object [13] (i.e., the `Map` object initialized in Line 10), which is returned to the method `AdbDebuggingManager.getPairedDevices` in Line 12 and the API `getPairedDevices` in Line 7, respectively. Then, in Line 3, the API returns the collection object, storing the instances of `PairDevice`, to its caller. In order to prevent the sensitive data from being retrieved by unauthorized apps that have not gained the permission `MANAGE_DEBUGGING`, the API enforces permission check in Line 2.

It is worth mentioning that, if the type of a sensitive field is a Java class defined in Android framework, this class is also considered as a sensitive class because the instance of the class, being stored in the sensitive field, contains sensitive data.

3.2 Two Types of Leak$_{Frm}$

In this section, we introduce two types of Leak$_{Frm}$ and present the threat model of Leak$_{Frm}$.

• **Definition of Leak$_{Frm}$**. If an `Intent` object sent by the framework, carrying the sensitive data, is received by the component of an unauthorized app that has not requested and gained any permissions, the sensitive data is leaked to the unauthorized app via the `Intent` object. We call this scenario the `Intent` based leak of sensitive data in Android framework (short for Leak$_{Frm}$). Note that, the unauthorized app cannot get the sensitive data by calling sensitive framework APIs due to lacking the required permissions.

Accordingly, there are three entities involved in Leak$_{Frm}$ as shown in Figure 5. (1) An `Intent` sender, i.e., Android framework. (2) An `Intent` object sent by the `Intent` sender, which carries the sensitive data. (3) An `Intent` recipient, i.e., the component of an

unauthorized app that receives the `Intent` object and gets the sensitive data carried by the `Intent` object, resulting in the leakage of the sensitive data.
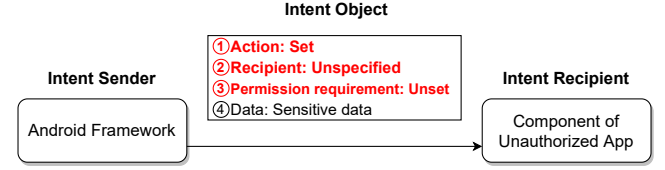

Figure 5: Entities of Leak$_{Frm}$.

To let the component of the unauthorized app receive the `Intent` object, the `Intent` object needs to meet three requirements. ① The action of the `Intent` object is set; ② The recipient of the `Intent` object is unspecified, so that the unauthorized app can register a component and declare that the component can perform the action of the `Intent` object in order to receive the `Intent` object; ③ The permission requirement for receiving the `Intent` object is unset when calling `Intent` sending APIs, so that the `Intent` object can be received by the component of the unauthorized app that does not request and gain any permissions.

Since the sensitive data carried by `Intent` object can be either values of sensitive fields or instances of sensitive classes in Android framework (see §3.1), we divide Leak$_{Frm}$ into two types. For each type of Leak$_{Frm}$, we detail the requirement for the sensitive data carried by involved `Intent` object (i.e., requirement ④) as follows.

Figure 6: Entities of Type-1 Leak$_{Frm}$.

• **Type-1**. Figure 6 illustrates the details about the entities involved in Type-1 Leak$_{Frm}$. In particular, the `Intent` object sent by Android framework carries the sensitive data stored in sensitive fields in Android framework (i.e., requirement ④). That is, the sensitive data stored in sensitive fields is leaked to the unauthorized app.

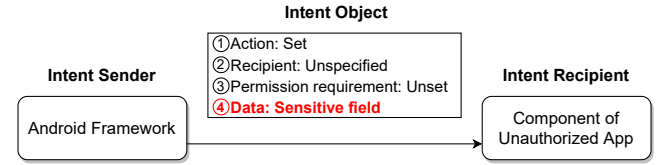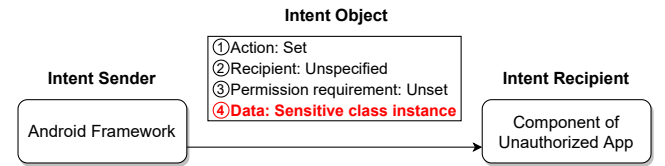**Figure 7: Entities of Type-2 Leak$_{Frm}$.**

• **Type-2**. Figure 7 shows the details about the entities involved in Type-2 Leak$_{Frm}$. Specifically, the `Intent` object sent by Android framework carries the instances of sensitive classes in the framework (i.e., requirement ④). That is, the sensitive data encapsulated in instances of sensitive classes is leaked to the unauthorized app.

• **Threat Model**: According to the details about the entities of Leak$_{Frm}$, we assume that attackers can induce victims into installing and launching the unauthorized app, whose component can receive the `Intent` object that is sent by Android framework and carries the sensitive data. Since the unauthorized app does not request any permissions, it is very hard for the existing permission based malicious app detection tools [25, 26, 37, 48] to detect it and warn the users.

3.3 Motivating Examples

For each type of Leak$_{Frm}$, we present a real case we found in the recently released Android 12 of the official Google Android system. The two cases have been confirmed and patched by the Android security team of Google.

• **Type-1**. The code fragment in Figure 8 presents a real case of Type-1 Leak$_{Frm}$.

```
// class "com.android.server.telecom.TtyManager"
01 private void updateCurrentTtyMode() {
02    String action = "*.CURRENT_TTY_MODE_CHANGED"; // action string
03    Intent intent = new Intent(action); ① // action of the Intent object is set
      ② // recipient of the Intent object is unspecified
04    int ttyMode = TtyManager.mCurrentTtyMode // the value of the sensitive field
05    intent.putExtra(*, ttyMode); ④ // the value of the sensitive field is carried
06    sendBroadcastAsUser(intent, null); ③ // permission requirement is unset
07 } /* irrelevant code is omitted */
```

Figure 8: A real case of Type-1 Leak$_{Frm}$.

The `Intent` object created and sent in the framework method `updateCurrentTtyMode` meets all requirements of the `Intent` object involved in Type-1 Leak$_{Frm}$. Specifically, ① in Line 3, the framework calls the constructor method of the class `Intent` to create and set the action of the `Intent` object. ② The framework does not call any APIs to specify the recipient of the created `Intent` object. ③ In Line 6, when calling the API `sendBroadcastAsUser` to send the `Intent` object, no permission requirement for receiving the `Intent` object is set. ④ In Line 4, `updateCurrentTtyMode` directly accesses the sensitive field `TtyManager.mCurrentTtyMode` (see §3.1) to retrieve the sensitive data instead of calling the sensitive framework API `getCurrentTtyMode` (introduced in §3.1). Then, in Line 5, the API `putExtra` is called to make the `Intent` object carry the sensitive data. Accordingly, an unauthorized app can register a broadcast receiver to receive the `Intent` object and then get the value of the sensitive field `mCurrentTtyMode` carried by the `Intent` object.

Note that, the framework only allows the authorized app, having gained the permission `READ_PRIVILEGED_PHONE_STATE`, to retrieve the data stored in the sensitive field `mCurrentTtyMode` by calling the sensitive framework API `TelecomServiceImpl.getCurrentTtyMode` (see §3.1). However, exploiting this case of Type-1 Leak$_{Frm}$, the unauthorized app can violate the access control to get the sensitive data stored in `mCurrentTtyMode`. As a result, the sensitive phone state information about the enabled teletypewriter mode is leaked to the unauthorized app.

• **Type-2**. The code fragment in Figure 9 presents a real case of Type-2 Leak$_{Frm}$.

```
// class "com.android.server.adb.AdbDebuggingManager$AdbDebuggingHandler"
01 private void onPairingResult(*) {
02    String action = "*.WIRELESS_DEBUG_PAIRING_RESULT"; // action string
03    Intent intent = new Intent(action); ① // action of the Intent object is set
      ② // recipient of the Intent object is unspecified
04    PairDevice device = new PairDevice(*); // an instance of the sensitive class
05    intent.putExtra(*, device); ④ // the instance of the sensitive class is carried
06    sendBroadcastAsUser(intent, null); ③ // permission requirement is unset
07 } /* irrelevant code is omitted */
```

Figure 9: A real case of Type-2 Leak$_{Frm}$.

The `Intent` object created and sent in the framework method `onPairingResult` meets all requirements of the `Intent` object involved in Type-2 Leak$_{Frm}$. Specifically, ① in Line 3, the framework calls the constructor method of the class `Intent` to create and set the action of the `Intent` object. ② The framework does not call any APIs to specify the recipient of the created `Intent` object. ③ In Line 6, when calling the API `sendBroadcastAsUser` to send the `Intent` object, no permission requirement for receiving the `Intent` object is set. ④ Instead of calling the sensitive framework API `getPairedDevices` to get instances of the sensitive class `PairDevice` (see §3.1), in Line 4, `onPairingResult` creates a new instance of `PairDevice` to encapsulate the sensitive data. Then, in Line 5, the API `putExtra` is called to make the `Intent` object carry the sensitive data. Accordingly, an unauthorized app can register a broadcast receiver to receive the `Intent` object and get the instance of the sensitive class `PairDevice` carried by the `Intent` object.

It is worth noting that Android framework only allows the authorized app, having gained the permission `MANAGE_DEBUGGING`, to retrieve instances of the sensitive class `PairDevice` by calling the sensitive framework API `AdbService.getPairedDevices` (see §3.1). However, exploiting this case of Type-2 Leak$_{Frm}$, the unauthorized app can retrieve instances of `PairDevice`. As a result, the sensitive information about the paired devices for wireless ADB debugging is leaked to the unauthorized app.

Although various work has been conducted to investigate and detect the `Intent` based data leak [30, 31, 34, 41, 44, 45, 49, 58, 60], to the best of our knowledge, they all focus on studying Leak$_{App}$, leaving Leak$_{Frm}$ not studied. The existing approaches cannot be adapted to identify Leak$_{Frm}$ because the sensitive data carried by the `Intent` object sent by Android apps and Android framework is retrieved using different methods. Specifically, since apps call sensitive framework APIs to retrieve the sensitive data, the existing approaches analyze these API calls in apps to determine whether the `Intent` object carries sensitive data. However, the framework does not rely on sensitive framework APIs to retrieve the sensitive data, and it can directly access sensitive fields to get the sensitive data or create instances of sensitive classes to encapsulate the sensitive data (see the motivating examples). Therefore, the existing approaches cannot find out that the `Intent` objects sent by the framework carry the sensitive data. Thus, they cannot uncover Leak$_{Frm}$.

To fill the gap, we design and develop `LeakDetector`, a new tool for uncovering Leak$_{Frm}$. Specifically, for the motivating example in Figure 8, `LeakDetector` first determines that `mCurrentTtyMode` is a sensitive field in Android framework. Then, `LeakDetector` analyzes the action and the recipient of the `Intent` object created in

updateCurrentTtyMode, the permission requirement for receiving the Intent object, and finds that they meet the requirement for the Intent object in Type-1 data leak. Moreover, LeakDetector analyzes the data carried by the Intent object and notices that the data refers to the value stored in the sensitive field mCurrentTtyMode (i.e., this Intent object carries the sensitive data). Since all requirements for the Intent object in Type-1 data leak are met, LeakDetector uncovers a case of Type-1 Leak$_{Frm}$.

4 LEAKDETECTOR

In §4.1, we present the overview and workflow of LeakDetector. Then, we introduce design details of LeakDetector in §4.2-§4.4.
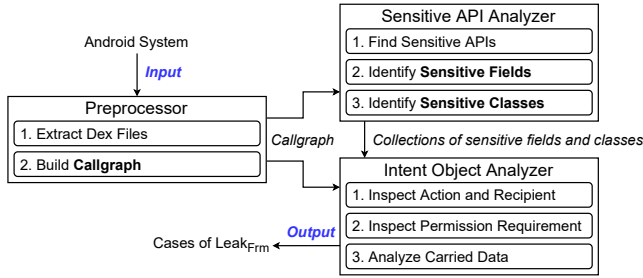

Figure 10: The overview and workflow of LeakDetector.

4.1 Overview and Workflow

Figure 10 illustrates the overview and workflow of LeakDetector, which consists of three modules, including preprocessor, sensitive API analyzer, and Intent object analyzer.

• **Preprocessor** (§4.2): For analyzing the sensitive framework APIs and the Intent objects sent by Android framework, LeakDetector extracts the Dex files, containing the implementations of Android framework [40], from the JAR files and APK files of the framework, and then builds the callgraph of Android framework.

• **Sensitive API Analyzer** (§4.3): For determining whether the data carried by Intent objects sent by the framework is sensitive data (i.e., values stored in sensitive fields or instances of sensitive classes), LeakDetector analyzes sensitive framework APIs. More specifically, LeakDetector first finds the sensitive APIs in Android framework, which enforce access control and return the retrieved sensitive data to their callers. Then, LeakDetector performs data flow analysis on the return values of the identified sensitive framework APIs to identify sensitive fields and sensitive classes in Android framework.

• **Intent Object Analyzer** (§4.4): In order to uncover Leak$_{Frm}$, LeakDetector analyzes the Intent objects sent by the framework to check whether they meet the requirements of the Intent object involved in Leak$_{Frm}$ introduced in §3.2. Specifically, for each Intent object sent by Android framework, LeakDetector conducts data flow analysis on it to inspect the action and recipient of the Intent object, and the permission requirement for receiving the Intent object. Moreover, LeakDetector analyzes the data carried by the Intent object to determine whether it is the value stored in a sensitive field or the instance of a sensitive class.

4.2 Preprocessing

To build the callgraph of Android framework, which is further used to analyze sensitive framework APIs (§4.3) and Intent objects sent by Android framework (§4.4), LeakDetector extracts Dex files of the framework and then analyzes the bytecode of Dex files.

• **Extracting Dex Files**. Depending on whether or not the complete stock ROMs of Android systems are published by the mobile vendors, LeakDetector employs two ways to extract the Dex files of Android framework. If the complete stock ROMs are unavailable, LeakDetector dumps the Dex files from smartphone at runtime. Otherwise, LeakDetector extracts the Dex files from stock ROMs.

Towards dumping the Dex files at runtime, LeakDetector uses ADB [2] to retrieve the JAR files from the system directory /system/framework, where most of the sensitive framework APIs are implemented [5]. In addition, since a few sensitive framework APIs (e.g., the APIs related to Bluetooth, Telecom, and TeleService) are implemented in the APK files placed in the system directories /system/app and /system/priv-app [59], we also dump these APK files.

To extract the Dex files from stock ROMs, LeakDetector follows the approaches introduced in the previous studies [40, 42]. Specifically, LeakDetector uses the tools simg2img [4] and 7-Zip [1] to unzip the ROMs and extract JAR files and APK files.

After getting the JAR files and APK files, LeakDetector decompresses them to extract the Dex files.

• **Building Callgraph**. Taking in the extracted Dex files of Android framework, LeakDetector analyzes the bytecode of Dex files to build the callgraph of the framework using Soot [21], a static bytecode analysis framework. Note that, due to the complexity and the large code base of Android framework, LeakDetector follows the existing work [28, 29] to adopt the Class Hierarchy Analysis based algorithm [39] to build the callgraph.

4.3 Analyzing Sensitive Framework APIs

As introduced in §3.1, sensitive framework APIs either return the data stored in sensitive fields or instances of sensitive classes to their callers. Accordingly, LeakDetector first finds the sensitive APIs in Android framework, and then conducts data flow analysis on their return values to identify the sensitive fields and sensitive classes in the framework, which are further used to uncover Leak$_{Frm}$ (§4.4).

• **Finding Sensitive Framework APIs**. Since sensitive framework APIs call permission-checking methods to enforce permission check on their calling apps (see §2.1), LeakDetector locates the callers of permission-checking methods from the callgraph of Android framework and treats them as sensitive framework APIs. Moreover, for each sensitive framework API, LeakDetector determines the permission p under check, which is further associated to the identified sensitive field or sensitive class for analyzing the Intent objects sent by Android framework (see §4.4). Since the permission under check is passed as an argument to the permission-checking method and each permission is represented as a string constant in Android framework [28, 29, 57], LeakDetector conducts interprocedural data flow analysis on arguments of the permission-checking method called by the sensitive API under analysis.

For example, since both of the method getCurrentTtyMode in Figure 3 and getPairedDevices in Figure 4 call permission-checking

method `enforceCallingOrSelfPermission` in Line 2, they are identified as sensitive framework APIs. By analyzing the arguments of `enforceCallingOrSelfPermission`, LeakDetector finds that these APIs enforce check on permissions `READ_PRIVILEGED_PHONE_STATE` and `MANAGE_DEBUGGING`, respectively.

• **Identifying Sensitive Fields**. Sensitive framework APIs can retrieve the sensitive data stored in sensitive fields and then return the data to their callers (see §3.1). Accordingly, to identify sensitive fields, LeakDetector conducts interprocedural use-def analysis on each sensitive API's return value to check whether the return value is defined as the field of a Java class defined in Android framework. If so, the field is identified as a sensitive field in Android framework.

For each identified sensitive field S_f, we associate it with the permission p being checked in the sensitive framework API, and store them in a field-permission map $M_{fp} := \{S_f \rightarrow p\}$. The constructed field-permission map is further used by LeakDetector to query whether the data carried by the `Intent` objects sent by Android framework is sensitive data (see §4.4).

For example, since the return value of the sensitive framework API `TelecomServiceImpl.getCurrentTtyMode` in Figure 3 is defined as the field `TtyManager.mCurrentTtyMode`, this field is identified as a sensitive field in Android framework. Then, we store the mapping `TtyManager.mCurrentTtyMode` → `READ_PRIVILEGED_PHONE_STATE` to the field-permission map M_{fp}.

Figure 11: An example of identifying sensitive class.

• **Identifying Sensitive Classes**. Sensitive framework APIs can create instances of sensitive classes to encapsulate the sensitive data and then return the instances to their callers (see §3.1). Accordingly, to identify sensitive classes, LeakDetector conducts interprocedural use-def analysis on each sensitive API's return value to check whether the return value is defined by the new expression of a Java class defined in Android framework. If so, the class is identified as a sensitive class in Android framework.

For each identified sensitive class S_c, we associate it with the permission p being checked in the sensitive framework API, and store them in a class-permission map $M_{cp} := \{S_c \rightarrow p\}$, which is further used by LeakDetector to query whether the data carried by the `Intent` objects sent by Android framework is sensitive data.

For example, since the return value of the sensitive framework API `getNetworkInfo` in Figure 11 is defined by the new expression of the class `NetworkInfo`, whose instances encapsulate the sensitive information about network connectivity on device [8], this class is identified as a sensitive class in Android framework. Then, the mapping `NetworkInfo` → `ACCESS_NETWORK_STATE` is stored to the class-permission map M_{cp}.

▷ **Handling Collection Objects**. As shown in Figure 4, sensitive framework APIs can put the sensitive data stored in sensitive fields or the instances of sensitive classes to collection objects [13] (e.g., `List` objects, `Set` objects, and `Map` objects), and then return

the collection objects to their callers (see §3.1). To handle this case, LeakDetector further analyzes each element added to the collection objects to identify sensitive fields and sensitive classes. Precisely, since each element stored in collection objects is passed as an argument to the element-adding methods (e.g., `add`, `append`, `put` listed in Table 3), LeakDetector performs use-def analysis on the arguments at calls to these methods. If the element is defined by the value of a field or the instance of a class in Android framework, the corresponding field or class is identified as a sensitive field or a sensitive class. Additionally, if the element is defined by another collection object, LeakDetector recursively analyzes the element added to that collection object to identify sensitive fields and sensitive classes in Android framework.

Table 3: A partial list of classes of collection objects.

Type	Class	Functionality	Method
List	ArrayList	Add an element.	add(*), push(*)
	LinkedList	Add elements from a collection object.	<init>(*), addAll(*)
Set	ArraySet	Add an element.	add(*)
	HashSet	Add elements from a collection object.	<init>(*), addAll(*)
Map	ArrayMap	Add a pair of elements.	append(*, *), put(*, *)
	HashMap		
	LinkedHashMap	Add elements from a collection object.	<init>(*), putAll(*)

For example, since the return value of the sensitive framework API `getPairedDevices` in Figure 4 is defined by a collection object (i.e., a `HashMap` object), whose elements are added by calling the method `put` in Line 11, LeakDetector performs use-def analysis on the arguments of `put`. LeakDetector finds that the element is defined by the new expression of the class `PairDevice`, and thus `PairDevice` is identified as a sensitive class. Then, the mapping `PairDevice` → `MANAGE_DEBUGGING` is stored to M_{cp}.

4.4 Analyzing Intent Objects

As introduced in §3.2, the `Intent` objects involved in both types of Leak$_{Frm}$ need to meet four requirements. Accordingly, to uncover Leak$_{Frm}$, LeakDetector first finds the `Intent` objects created in the framework by locating the new expression of the class `Intent` in framework methods. Then, for each `Intent` object, LeakDetector conducts data-flow analysis on it to determine whether it meets the four requirements.

• **Inspecting Action of Intent Objects (Requirement ①)**. To inspect whether the action of an `Intent` object sent by Android framework is set, LeakDetector conducts use-def analysis on the `Intent` object to find all the statements related to this object, and then it examines each statement to see whether an API for setting `Intent` action is called. If so, the action of the `Intent` object is set.

For example, since the constructor method of `Intent`, taking in the action string as its argument, is invoked in Figure 8 and Figure 9, LeakDetector determines that both of the action of the `Intent` object created in the method `updateCurrentTtyMode` and that of the `Intent` object created in the method `onPairingResult` are set by Android framework.

• **Inspecting Recipient of Intent Objects (Requirement ②)**. To check whether the recipient of an `Intent` object sent by Android framework is unspecified, LeakDetector performs use-def analysis

on the `Intent` object to find all the statements associated with this object, and then `LeakDetector` analyzes each statement to see whether an API for specifying `Intent` recipient is invoked. If not, the recipient of the `Intent` object is unspecified.

For example, since none of APIs for setting `Intent` recipient is called in Figure 8 and Figure 9, `LeakDetector` identifies that both of the recipient of the `Intent` object created in `updateCurrentTtyMode` and that of the `Intent` object created in `onPairingResult` have not been specified by Android framework.

• **Inspecting Permission Requirement for Receiving Intent Objects (Requirement ③).** To examine whether the permission requirement for receiving an `Intent` object sent by the framework is unset, `LeakDetector` collects all the statements related to the `Intent` object by applying use-def analysis on this object, and then `LeakDetector` analyzes the statements that call `Intent` sending APIs to see whether the permission requirement is passed as the argument to these APIs. Specifically, `LeakDetector` conducts def-use analysis on each argument of the called `Intent` sending APIs to see whether the permission strings (e.g., those stored in the field-permission map M_{fp} and the class-permission map M_{cp}) are assigned to the argument. If not, `LeakDetector` determines that the permission requirement for receiving the `Intent` object is unset.

For instance, since no permission strings are assigned to the arguments of `Intent` sending API `sendBroadcastAsUser` in Figure 8 and Figure 9, `LeakDetector` considers both of the permission requirement for receiving the `Intent` object created in the method `updateCurrentTtyMode` and that for receiving the `Intent` object created in `onPairingResult` have not been set by Android framework.

• **Analyzing Data Carried by Intent Objects (Requirement ④).** To check whether the data carried by an `Intent` object sent by Android framework is the data stored in a sensitive field (i.e., requirement ④ for Type-1 Leak_{Frm}) or an instance of a sensitive class (i.e., requirement ④ for Type-2 Leak_{Frm}), `LeakDetector` first applies use-def analysis on the `Intent` object to collect all the statements related to this object. Then, `LeakDetector` further analyzes the statements that call the series of `putExtra` APIs to see whether the data carried by the `Intent` object meets the requirement. Precisely, for each statement, `LeakDetector` performs use-def analysis on the argument of the `putExtra` API to see whether the definition of the argument is either a field in the field-permission map M_{fp} or an instance of the class in the class-permission map M_{cp}. If so, `LeakDetector` considers that the `Intent` object sent by the framework carries the sensitive data.

For example, since the definition of the data carried by the `Intent` object sent in the method `updateCurrentTtyMode` in Figure 8 is the sensitive field `TtyManager.mCurrentTtyMode`, `LeakDetector` determines that this `Intent` object carries the sensitive data stored in a sensitive field. In addition, since the definition of the data carried by the `Intent` object sent in the method `onPairingResult` in Figure 9 is an instance of the sensitive class `PairDevice`, `LeakDetector` determines that this `Intent` object carries the sensitive data encapsulated in an instance of a sensitive class.

5 EVALUATION

We implement `LeakDetector` in around 6.7k SLOC in Java. We evaluate it by answering the following three research questions (RQs).

RQ1: Can `LeakDetector` precisely identify the sensitive fields and sensitive classes in Android framework?
RQ2: Can `LeakDetector` effectively uncover Leak_{Frm} in commercial Android systems?
RQ3: How is the efficiency of `LeakDetector`?

Table 4: Details about Android systems under analysis.

	Name	Version	Vendor	ROM	Date	#Class	#Method
1	AOSP	Android 11	Google	✓	10/2021	48,487	399,301
2	AOSP	Android 12	Google	✓	11/2021	64,868	485,794
3	ColorOS	Android 11	Oneplus	✗	12/2021	130,467	1,066,362
4	ColorOS	Android 12	Oneplus	✗	01/2022	148,325	1,147,707
5	MIUI	Android 11	Xiaomi	✓	12/2021	81,233	646,995
6	MIUI	Android 12	Xiaomi	✓	01/2022	90,996	718,184
7	OneUI	Android 11	Samsung	✓	12/2021	164,768	1,155,804
8	OneUI	Android 12	Samsung	✓	01/2022	167,790	1,197,433
9	OriginOS	Android 11	Vivo	✗	12/2021	102,551	798,347
10	OriginOS	Android 12	Vivo	✗	02/2022	123,392	932,481

Data Set. To answer the research questions, we use `LeakDetector` to analyze 10 commercial Android systems. Table 4 lists the details about the systems under evaluation, where `Name`, `Version`, `Vendor`, and `ROM` provide the information about system name, Android version, mobile vendor that deploys the corresponding system on its mobile devices, and whether we found the complete stock ROM of the system. In addition, `#Classes` and `#Methods` provide the number of Java classes and methods defined and used in Android framework of the system, respectively. In detail, we choose the latest versions of Android systems (i.e., Android 11 and 12) deployed on smartphones from popular mobile vendors [17] as our targets, including the official Android system AOSP [10] deployed on Google Pixel, and the commercial Android systems ColorOS [11] deployed on Oneplus smartphones, MIUI [16] deployed on Xiaomi smartphones, OneUI [18] deployed on Samsung Galaxy smartphones, and OriginOS [19] deployed on Vivo smartphones. We extracted JAR files and APK files of Android framework of the systems under evaluation from the downloaded stock ROMs of Google Pixel 4, Xiaomi 11, and Samsung Galaxy S21, and dumped the corresponding files from ColorOS and OriginOS running on smartphones Oneplus 9 and Vivo iQOO 8 between October, 2021 and February, 2022.

Evaluation Setup The evaluation was conducted on a PC equipped with Intel i7-6700k CPU, 64GB RAM, and 2TB SSD.

5.1 Identifying Sensitive Fields and Sensitive Classes in Android Framework (RQ1)

Table 5 lists the results about the sensitive fields and sensitive classes identified by `LeakDetector` in Android framework of AOSP, where `#Field` and `#Class` provide the number of identified sensitive fields and sensitive classes, respectively, and `#Total` provides the total number of sensitive entities. Additionally, Table 6 presents the results about the sensitive fields and sensitive classes identified in Android framework of the commercial systems other than AOSP, where `#Add` provides the increased number of sensitive entities compared to AOSP. Specifically, comparing at the same Android version, the numbers of sensitive fields and sensitive classes identified in Android framework of ColorOS, MIUI, OneUI, and OriginOS

are all greater than those in AOSP. Meanwhile, Android framework of OneUI Android 12 has the largest number of sensitive entities among the systems under evaluation.

Table 5: Statistics about the sensitive fields and sensitive classes in Android framework of AOSP.

	Name	Version	#Field	#FP$_{Field}$	#Class	#FP$_{Class}$	#Total	#FP$_{Total}$
1	AOSP	Android 11	159	8	87	4	246	12
2	AOSP	Android 12	163	6	97	4	260	10

Table 6: Statistics about the sensitive fields and sensitive classes in Android framework of other commercial systems.

	Name	Version	#Field	#Add$_{Field}$	#Class	#Add$_{Class}$	#Total	#Add$_{Total}$
1	ColorOS	Android 11	224	65	114	27	338	92
2	ColorOS	Android 12	186	23	111	14	297	37
3	MIUI	Android 11	195	36	96	9	291	45
4	MIUI	Android 12	178	15	100	3	278	18
5	OneUI	Android 11	232	73	115	28	347	101
6	OneUI	Android 12	211	48	132	35	343	83
7	OriginOS	Android 11	184	25	91	4	275	29
8	OriginOS	Android 12	186	23	101	4	287	27

```
// class "com.android.providers.settings.SettingsProvider
01 private Setting getConfigSetting(*) {                    permission check
02     checkCallingOrSelfPermission(READ_DEVICE_CONFIG);  ◀—
03     if ( /* condition is satisfied*/ ) { // the new instance contains sensitive data
04  ✓    return new Setting(*); // the class Setting is identified as a sensitive class
05     } else { // the field SettingsState.mNullSetting does not contain sensitive data
06  ✗    return SettingsState.mNullSetting; // wrongly identified as a sensitive field
07 } } /* irrelevant code is omitted */
```

Figure 12: A false positive of identified sensitive entities.

Moreover, to assess whether `LeakDetector` can precisely identify sensitive fields and sensitive classes in Android framework, we manually investigate the results on AOSP by referring to the source code of sensitive framework APIs, where the sensitive entities are identified. The assessment results are shown in Table 5, where #FP provides the number of false positives in the identified sensitive entities. We totally find 10 and 12 false positives in the sensitive fields and sensitive classes identified in the framework of AOSP Android 11 and AOSP Android 12, respectively. That is, the precision of identified sensitive entities are 95.1% and 96.2%, respectively.

We inspect the false positives and observe that, although these false positives are return values of sensitive framework APIs, they are not the targets that the permission check intends to protect. For example, as shown in Figure 12, the method `getConfigSetting` enforces permission check in Line 2, and it has two return values, including an instance of the class `Setting` in Line 4 and the data accessed from the field `SettingsState.mNullSetting` in Line 6. Therefore, `LeakDetector` considers that `Setting` is a sensitive class and `mNullSetting` is a sensitive field. However, we find that, the instance of `Setting` contains sensitive information about device configurations, whereas the value of `mNullSetting` is an empty

object, which does not contain any sensitive information. That is, the permission check is enforced to prevent the instance of `Setting` rather than the data stored in `mNullSetting` from being retrieved by unauthorized apps. Thus, `mNullSetting` should not be treated as a sensitive field and it is a false positive.

Due to the complexity and huge code base of the Android system, it is non-trivial for `LeakDetector` to identify all sensitive fields and sensitive classes and for us to find all false negatives. We randomly chose ten system services (e.g., AdbService, ConnectivityService, TelecomServiceImpl) of AOSP and manually constructed the ground truth of sensitive fields and sensitive classes by code review. Then, we used the ground truth to find false negatives and compute the recall. Since the manually found sensitive fields and sensitive classes are all identified by `LeakDetector`, no false negatives were found.

> **Answer to RQ1:** `LeakDetector` can precisely identify sensitive fields and sensitive classes in Android framework. Specifically, `LeakDetector` identifies the sensitive entities in Android framework of AOSP Android 11 and AOSP Android 12 with the precision 95.1% and 96.2%, respectively.

5.2 Uncovering Leak$_{Frm}$ in Commercial Android Systems (RQ2)

Table 8 lists the results about Leak$_{Frm}$ uncovered by `LeakDetector` in 10 commercial systems, where #Leak, #Type-1, and #Type-2 provide the number of discovered cases of Leak$_{Frm}$ in both types, in Type-1, and in Type-2, respectively. Specifically, `LeakDetector` finds at least 12 cases of Leak$_{Frm}$ in every system under evaluation. Compared with AOSP, more cases of Leak$_{Frm}$ are uncovered in the framework of ColorOS, MIUI, OneUI, and OriginOS. We discover that the wrongly identified sensitive fields and sensitive classes cause one false positive in cases of Leak$_{Frm}$.

Moreover, to inspect whether the uncovered cases of Leak$_{Frm}$ are exploitable, we manually create proof-of-concept (PoC) unauthorized apps and trigger the framework to send the `Intent` objects involved in these cases. For each case, if the sensitive data carried by the `Intent` object can be received by our PoC app, we consider the case is exploitable. In Table 8, #Exploitation provides the numbers of exploitable cases. The details about each `Intent` object involved in the exploitable cases are presented in Table 7 and Table 9, where `Intent Action`, `Intent Sending API`, and `Sensitive Entity` provide the information about the action of the `Intent` object, the API for sending the `Intent` object, and the entity of the sensitive data carried by the `Intent` object. Specifically, we find 11 exploitable cases of Leak$_{Frm}$ in two Android versions of AOSP, and 25 exploitable cases in the remaining 8 commercial Android systems. For example, the two real cases introduced in §3.3 refer to the 2nd and 8th cases in Table 7. We have reported these exploitable cases to the corresponding mobile vendors. At the time of writing, 16 of them have been confirmed, and we received bug bounty rewards from Google, Samsung, and Xiaomi.

By further analyzing the unexploitable cases, we find the main reason: the dead code, which will never be executed by Android

Table 7: Details about the exploitable Intent based leak of sensitive data in Android framework of AOSP.

	Name	Version	Type	Sensitive Entity	Intent Action	Intent Sending API
1	AOSP	Android 11	Type-1	ClientModeImpl.mWifiState	android.net.wifi.WIFI_STATE_CHANGED	sendStickyBroadcastAsUser
2	AOSP	Android 11/12	Type-1	TtyManager.mCurrentTtyMode	android.telecom.action.CURRENT_TTY_MODE_CHANGED	sendBroadcastAsUser
3	AOSP	Android 11/12	Type-1	GsmCdmaPhone.mIsPhoneInEcmState	android.intent.action.EMERGENCY_CALLBACK_MODE_CHANGED	sendStickyBroadcastAsUser
4	AOSP	Android 11/12	Type-1	ImsPhone.mIsPhoneInEcmState	android.intent.action.EMERGENCY_CALLBACK_MODE_CHANGED	sendStickyBroadcastAsUser
5	AOSP	Android 11/12	Type-1	AdbConnectionInfo.mPort	com.android.server.adb.WIRELESS_DEBUG_STATUS	sendBroadcastAsUser
6	AOSP	Android 11/12	Type-1	TetheredSoftApTracker.mTetheredSoftApState	android.net.wifi.WIFI_AP_STATE_CHANGED	sendStickyBroadcastAsUser
7	AOSP	Android 12	Type-1	ConcreteClientModeManager.mWifiState	android.net.wifi.WIFI_STATE_CHANGED	sendStickyBroadcastAsUser
8	AOSP	Android 11/12	Type-2	PairDevice	com.android.server.adb.WIRELESS_DEBUG_PAIRING_RESULT	sendBroadcastAsUser
9	AOSP	Android 11/12	Type-2	PairDevice	com.android.server.adb.WIRELESS_DEBUG_PAIRED_DEVICES	sendBroadcastAsUser
10	AOSP	Android 11/12	Type-2	PrintJobInfo	android.print.PRINT_DIALOG	getActivityAsUser
11	AOSP	Android 11/12	Type-2	PhoneAccountHandle	android.intent.action.CALL	getActivity

Table 8: Overall detection results of LeakDetector.

	Name	Version	#Leak	#Type-1	#Type-2	#Exploitation
1	AOSP	Android 11	12	7	5	10
2	AOSP	Android 12	13	7	6	10
3	ColorOS	Android 11	17	8	9	13
4	ColorOS	Android 12	15	7	8	10
5	MIUI	Android 11	21	10	11	14
6	MIUI	Android 12	18	8	10	12
7	OneUI	Android 11	21	10	11	16
8	OneUI	Android 12	17	9	8	14
9	OriginOS	Android 11	18	7	11	15
10	OriginOS	Android 12	14	5	9	11

framework at runtime, wants to send the Intent objects. Specifically, we manually inspect the decompiled code of Android framework and pre-installed apps to determine whether unexploitable cases are dead code by following the approach presented in the existing work [40]. If a case is neither called by pre-installed apps nor framework APIs that can be called by apps, we treat it as an unexploitable case.

> **Answer to RQ2:** LeakDetector effectively uncovers $Leak_{Frm}$ in commercial Android systems under evaluation. More specifically, applying LeakDetector to 10 commercial Android systems, we discover a total of 36 exploitable cases of $Leak_{Frm}$.

5.3 Measuring Efficiency of LeakDetector (RQ3)

We run LeakDetector 10 times on each commercial Android system under evaluation to measure LeakDetector's efficiency, and the results are summarized in Table 10. Precisely, we measure the average time (in seconds) taken by LeakDetector to uncover $Leak_{Frm}$ in each system, and calculate the standard deviation. Additionally, we compute the proportion of analysis time for each main task of LeakDetector, including building callgraph, analyzing sensitive framework APIs, and analyzing Intent objects in the framework.

More specifically, for every system under analysis, LeakDetector takes less than 50 minutes to uncover $Leak_{Frm}$. We notice that the task of building callgraph takes the majority (> 98.0%) of the time in the analysis. In particular, LeakDetector spends more than 47

minutes in building the callgraph of Android framework of OneUI Android 12, and takes around 99.1% of the analysis time to construct the callgraph of the framework of ColorOS Android 12. The reason behind this is that, when building callgraph, LeakDetector needs to load and analyze the bytecode of millions or even billions of Java methods in Android framework (see Table 4). Towards the tasks of analyzing framework APIs and inspecting Intent objects, since LeakDetector just performs data flow analysis on return values of sensitive framework APIs and Intent objects rather than every variables in the framework, these two tasks only take a very small portion (< 2.0%) of the time within the analysis.

> **Answer to RQ3:** LeakDetector can efficiently uncover $Leak_{Frm}$. For every Android system under evaluation, LeakDetector takes less than 50 minutes to uncover $Leak_{Frm}$.

5.4 Case Study

Table 11 details the sensitive data that will be leaked to unauthorized apps in the uncovered 36 exploitable cases of $Leak_{Frm}$. Moreover, it includes the permissions checked by the sensitive framework APIs to restrict the retrieval of the sensitive data. In detail, the leaked data mainly contains the sensitive information about phone state, Wi-Fi state, network connections, and connected Bluetooth devices.

In the following, we present two case studies to show that attackers can abuse the leaked sensitive data to control the device by gaining the Shell privilege (case-1) and locate the user (case-2).

• **Case-1**. Referring to the 5th case presented in Table 7, when the wireless debugging state on device changes, Android framework will send an Intent object, carrying the value of the sensitive field AdbConnectionInfo.mPort. Meanwhile, according to the 5th Row in Table 11, since this sensitive field stores the sensitive information about the port on which the wireless debugging is opened, such sensitive information will be leaked to the unauthorized app.

▷ **Exploitation**. To receive the Intent object, the attacker can create a malicious app to register a broadcast receiver, which declares that it can perform the action WIRELESS_DEBUG_STATUS. Consequently, when the wireless debugging state on the device turns on, the malicious app will receive the sensitive information about

Table 9: Details about the exploitable Intent based leak of sensitive data in Android framework of other commercial systems.

	Name	Version	Type	Sensitive Entity	Intent Action	Intent Sending API
1	ColorOS/MIUI /OriginOS	Android 11	Type-1	WifiCountryCode.mDefaultCountryCode	android.net.wifi.COUNTRY_CODE_CHANGED	sendStickyBroadcastAsUser
2	ColorOS/MIUI /OriginOS	Android 11	Type-1	WifiCountryCode.mTelephonyCountryCode	android.net.wifi.COUNTRY_CODE_CHANGED	sendStickyBroadcastAsUser
3	ColorOS/MIUI /OriginOS	Android 11	Type-2	LinkProperties	android.net.wifi.LINK_CONFIGURATION_CHANGED	sendBroadcastAsUser
4	ColorOS	Android 11/12	Type-2	WifiInfo	android.net.wifi2.STATE_CHANGE	sendStickyBroadcastAsUser
5	ColorOS	Android 11	Type-2	WifiInfo	android.net.wifi.STATE_CHANGE	sendStickyBroadcastAsUser
6	ColorOS	Android 12	Type-2	WifiConfiguration	android.net.wifi.DISABLE_ALERT_NETWORKS	sendStickyBroadcastAsUser
7	MIUI	Android 11/12	Type-2	BluetoothDevice	com.android.bluetooth.headset.cancel.antilost	getBroadcast
8	MIUI	Android 11/12	Type-2	WifiConfiguration	miui.intent.action.WIFI_CONNECTION_FAILURE	sendBroadcastAsUser
9	MIUI	Android 12	Type-2	BluetoothDevice	com.android.bluetooth.headset.click.antilost_notification	getBroadcast
10	MIUI	Android 12	Type-2	BluetoothDevice	com.android.bluetooth.headset.click.detail_notification	getBroadcast
11	OneUI	Android 11	Type-1	BluetoothDevice.mAddress	com.samsung.android.input.REMOTE_INPUT_DEVICE_STATE_CHANGED	sendStickyBroadcastAsUser
12	OneUI	Android 11/12	Type-1	BluetoothDevice.mAddress	com.samsung.android.input.REMOTE_INPUT_READY_TO_CONNECT	sendBroadcast
13	OneUI	Android 11/12	Type-1	HeadsetStateMachine.mDevice	android.intent.action.VOICE_COMMAND	startActivityAsUser
14	OneUI	Android 11/12	Type-1	AidGroup.aids	org.mona.CardEmulation.action.PPSE_UPDATED	sendBroadcastAsUser
15	OneUI	Android 11	Type-2	BluetoothDevice	com.samsung.bluetooth.pan.panu_auth.auth_confirm	sendBroadcast
16	OneUI	Android 11/12	Type-2	BluetoothDevice	com.samsung.btopp.intent.action.MSG_SESSION_COMPLETE	sendBroadcast
17	OneUI	Android 11/12	Type-2	BluetoothDevice	com.samsung.btopp.intent.action.MSG_SESSION_ERROR	sendBroadcast
18	OneUI	Android 11	Type-2	BluetoothDevice	com.samsung.bluetooth.pan.inactivenap.ASK_CONFIRM	sendBroadcast
19	OneUI	Android 11/12	Type-2	BluetoothDevice	com.android.nfc.handover.action.DENY_CONNECT	sendBroadcast
20	OneUI	Android 11	Type-2	PhoneAccountHandle	android.intent.action.CALL_PRIVILEGED	startActivity
21	OriginOS	Android 11	Type-2	LinkProperties	android.net.extwifi.LINK_CONFIGURATION_CHANGED	sendBroadcastAsUser
22	OriginOS	Android 11/12	Type-2	BluetoothDevice	android.bluetooth.multA2dp.action.STATE_CHANGED	sendBroadcast
23	OriginOS	Android 11/12	Type-2	WifiConfiguration	android.net.extwifi.VIVO_CONNECTION_ALERT	sendStickyBroadcastAsUser
24	OriginOS	Android 11/12	Type-2	WifiConfiguration	android.net.wifi.VIVO_CONNECTION_ALERT	sendStickyBroadcastAsUser
25	MIUI	Android 11	Type-2	LinkProperties	android.net.wifi.SLAVE_LINK_CONFIGURATION_CHANGED	sendBroadcastAsUser

Table 10: Time (in seconds) consumed for each main task of LeakDetector.

	Name	Version	Total		Build Callgraph			Analyze Sensitive APIs			Analyze Intent Objects		
			Mean	Deviation	Mean	Deviation	Proportion	Mean	Deviation	Proportion	Mean	Deviation	Proportion
1	AOSP	Android 11	476.9	7.7	470.0	7.7	98.5%	3.1	0.2	0.7%	3.8	0.1	0.8%
2	AOSP	Android 12	646.7	4.0	637.8	3.8	98.6%	4.3	0.3	0.7%	4.6	0.2	0.7%
3	ColorOS	Android 11	1505.4	8.2	1482.6	8.1	98.5%	5.5	0.3	0.4%	17.3	0.4	1.1%
4	ColorOS	Android 12	1745.7	10.8	1730.6	10.8	99.1%	5.1	0.3	0.3%	10.0	0.3	0.6%
5	MIUI	Android 11	914.3	5.8	903.2	5.9	98.8%	4.5	0.2	0.5%	6.6	0.2	0.7%
6	MIUI	Android 12	1034.2	7.7	1022.4	7.7	98.9%	4.7	0.2	0.4%	7.1	0.2	0.7%
7	OneUI	Android 11	2180.3	12.4	2147.2	12.0	98.5%	7.0	0.3	0.3%	26.1	0.6	1.2%
8	OneUI	Android 12	2881.4	33.2	2831.1	33.7	98.2%	21.9	0.8	0.8%	28.4	0.3	1.0%
9	OriginOS	Android 11	1181.7	10.8	1158.3	10.5	98.0%	4.9	0.2	0.4%	18.5	0.6	1.6%
10	OriginOS	Android 12	1547.6	20.0	1528.1	19.8	98.7%	5.8	0.3	0.4%	13.7	0.3	0.9%

the port on which the wireless debugging is opened. Note that, the app does not need to gain the permission MANAGE_DEBUGGING, which is enforced on the sensitive framework API to prevent unauthorized apps from getting such sensitive information.

▷ **Security Impact**. The malicious app can abuse the leaked port information to set up wireless ADB debugging, run malicious code in Shell privilege, and further control the device. More specifically, to locally set up ADB debugging in the malicious app, the app can bundle an ADB server within it and leverage the wireless ADB debugging feature on device [15]. In normal cases, setting up wireless ADB debugging requires the user to turn on the wireless debugging mode and then input the information about the port on which the wireless debugging is opened [2]. However, since the port information will be leaked to the malicious app, after inducing the victim to enable the wireless debugging mode, the malicious app can set up wireless debugging without the need of user input. Then, the app can launch ADB shell to run malicious code in Shell

Table 11: Details about the sensitive data leaked from Android framework.

	Type	Sensitive Entity	Permission	Sensitive Data
1	Field	ClientModeImpl.mWifiState	ACCESS_WIFI_STATE	Wi-Fi enabled state of the device.
2	Field	TtyManager.mCurrentTtyMode	READ_PRIVILEGED_PHONE_STATE	Teletypewriter mode of the device.
3	Field	GsmCdmaPhone.mIsPhoneInEcmState	READ_PRIVILEGED_PHONE_STATE	Emergency callback mode of GSM.
4	Field	ImsPhone.mIsPhoneInEcmState	READ_PRIVILEGED_PHONE_STATE	Emergency callback mode of IMS.
5	Field	AdbConnectionInfo.mPort	MANAGE_DEBUGGING	Port on which the wireless ADB debugging is opened.
6	Field	TetheredSoftApTracker.mTetheredSoftApState	ACCESS_WIFI_STATE	Tethered Wi-Fi hotspot enabled state.
7	Field	ConcreteClientModeManager.mWifiState	ACCESS_WIFI_STATE	Wi-Fi enabled state of the device.
8	Field	WifiCountryCode.mDefaultCountryCode	NETWORK_SETTINGS	Default country code set by the Wi-Fi framework.
9	Field	WifiCountryCode.mTelephonyCountryCode	NETWORK_SETTINGS	Country code supplied by the telephony module.
10	Field	BluetoothDevice.mAddress	BLUETOOTH	Hardware address of a remote bluetooth device.
11	Field	HeadsetStateMachine.mDevice	BLUETOOTH	Information about the connected Bluetooth headset, such as MAC address.
12	Field	AidGroup.aids	NFC_PREFERRED_PAYMENT_INFO	Registered application identifiers for the NFC card emulation service (e.g., payment service).
13	Class	PairDevice	MANAGE_DEBUGGING	Information about a client in the ADB connection, such as client name, GUID.
14	Class	PrintJobInfo	ACCESS_ALL_PRINT_JOBS	Properties of a print job, such as task id, print attributes.
15	Class	PhoneAccountHandle	MODIFY_PHONE_STATE	The unique identifier for a method to make a phone call, such as ICCID for SIM card.
16	Class	LinkProperties	ACCESS_NETWORK_STATE	Properties of a network link, such as gateway's MAC address.
17	Class	WifiInfo	ACCESS_WIFI_STATE	Information about an active Wi-Fi connection, such as RSSI.
18	Class	WifiConfiguration	ACCESS_WIFI_STATE	Configurations about a Wi-Fi network, such as SSID, BSSID, password.
19	Class	BluetoothDevice	BLUETOOTH	Information about a remote Bluetooth device, such as MAC address.

privilege [62]. Since ADB shell provides powerful functionality [2], such as installing apps and modifying phone states, the malicious app can abuse it to further control the device.

• **Case-2**. Referring to the 24th case listed in Table 9, when an alert of Wi-Fi connection occurs (e.g., failure of connecting to the Wi-Fi network due to invalid password), Android framework of OriginOS will send an `Intent` object, carrying an instance of the sensitive class `WifiConfiguration`. Meanwhile, according to the 18th Row in Table 11, since this sensitive class encapsulates the sensitive information about the configuration of the Wi-Fi network, such as the name of Wi-Fi network (i.e., SSID), such sensitive information will be leaked to the unauthorized app.

▷ **Exploitation**. To receive the `Intent` object, the adversary can create a malicious app to register a broadcast receiver, which declares that it can perform the action `VIVO_CONNECTION_ALERT`. Consequently, when an alert of Wi-Fi connection occurs, the malicious app will receive the sensitive information about the configuration of this Wi-Fi network. Note that, the app does not need to gain the permission `ACCESS_WIFI_STATE`, which is enforced on the sensitive framework API to prevent unauthorized apps from getting such sensitive information.

▷ **Security Impact**. We find that there are two ways for the malicious app to locate the user by abusing the leaked Wi-Fi configuration information. First, abusing the semantic information in SSID, such as names of business entities, the attacker can infer the user's fine-granular location [35, 56]. Second, we discover that OriginOS customizes `WifiConfiguration` to make it encapsulate more sensitive information related to the Wi-Fi network. Specifically, a new field `vivoWifiConfiguration` is added to `WifiConfiguration`, which stores the instance of `VivoWifiConfiguration`, a new class introduced by OriginOS. The new class has two fields `vivoLatitude`

and `vivoLongitude`, which store the latitude and longitude of the user's position when connecting the Wi-Fi network. Since the sensitive geolocation information is leaked to the malicious app associated with the instance of the sensitive class `WifiConfiguration`, the attacker can abuse the geolocation information to locate user.

6 LIMITATIONS AND FUTURE WORK

`LeakDetector` has three main limitations.

First, `LeakDetector` mainly considers the permission based access control in Android framework, because the framework employs permission check to protect `Intent` [43]. In future work, we will take other types of access control enforced on sensitive framework APIs into consideration, and then extend `LeakDetector` to uncover more cases of Leak$_{Frm}$.

Second, we currently require manual efforts to evaluate whether the cases of Leak$_{Frm}$ uncovered by `LeakDetector` are exploitable. In future work, we intend to reducing the manual work. For example, we can leverage dynamic testing [38] to trigger Android framework to send `Intent` objects.

Third, `LeakDetector` just analyzes Dex files of Android framework (i.e., Java code of Android framework), and thus `LeakDetector` may miss the cases of Leak$_{Frm}$ in native code of Android framework. We will adopt a hybrid approach to find such in future work. For example, we could conduct binary analysis to identify the sensitive data in native code and inspect the `Intent` objects sent by native code of Android framework. We could also adopt dynamic instrumentation techniques [61] to track the sensitive data in native code to find whether it will be sent by the `Intent` objects, which do not set the permission requirements for their recipients.

7 RELATED WORK

We present the related work on analyzing `Intent` in Android apps in §7.1 and analyzing Android framework in §7.2.

7.1 Analyzing Intent in Android Apps

Numerous studies have been proposed to resolve the `Intent` objects and analyze their security problems in Android apps. EPICC [53] and IC3 [52] use interprocedural data-flow and points-to analysis to resolve `Intent` objects for uncovering inter-component communication connection points in apps. PRIMO [54] applies probabilistic models to resolve Intent and Intent filter matching to reduce false positives in the analysis results of EPICC and IC3.

ComDroid [36], CHEX [50], COVERT [30], and SEALANT [47] study the security challenges of `Intent` (e.g., `Intent` hijacking and `Intent` spoofing) and statically detect these vulnerabilities in apps. Apposcopy [44], Amandroid [60], IccTA [49], DroidSafe [45], DI-ALDroid [34], Karim et al. [41], and Flair [31] rely on predefined sensitive framework API lists generated by the existing work (e.g., Susi [27] and PScout [28]) to identify the sensitive data retrieved by apps. Then, these studies statically inspect the `Intent` objects sent by apps to identify $Leak_{App}$.

However, the existing approaches cannot be applied to uncover $Leak_{Frm}$, because Android framework does not rely on sensitive framework APIs to retrieve the sensitive data, resulting in that these approaches cannot determine whether the `Intent` objects sent by Android framework carry the sensitive data.

7.2 Analyzing Android Framework

Most of the existing work, targeting at analyzing Android framework, focuses on building the permission specification for framework APIs. Precisely, they identify the access control enforced on framework APIs. PScout [28] and ARCADE [24] construct a context-insensitive and a path-sensitive callgraph of Android framework, respectively, to correlate framework APIs to their required permissions. Alexandre et al. [33] assessed the performance of two callgraph building algorithms (i.e., CHA and Spark) on constructing the mapping between framework APIs and permissions. AXPLORER [29] and HEAPHELPER [51] build more precise callgraphs of Android framework to make the generated permission specification more accurate. PSGen [63] analyzes native code of Android framework to build the permission specification for native framework APIs. Besides the tools built upon static analysis, DYNAMO [38] dynamically tests Android framework APIs and records the permissions being checked to generate the permission specification.

An application of permission specification analysis is to identify the inconsistent permission check enforced in Android framework. Kratos [57], AceDroid [23], ACMiner [46], and IAceDroid [62] statically build the permission specification for framework APIs, and then they examine whether the two framework APIs, implementing the same functionality, are protected by the same access control to detect inconsistent access control enforcement.

However, since neither of these tools analyze sensitive framework APIs to find the sensitive entities in Android framework nor inspect the `Intent` objects sent by Android framework, they cannot be used to uncover $Leak_{Frm}$.

8 CONCLUSION

We are the *first* to reveal and systematically investigate the `Intent` based leak of sensitive data in Android framework. To automatically uncover such kind of vulnerability, we design and develop `LeakDetector`, a novel tool that performs static analysis on Android framework, especially the sensitive framework APIs and the `Intent` objects sent by the framework. Applying `LeakDetector` to 10 commercial Android systems, we find that `LeakDetector` can effectively and efficiently discover the `Intent` based data leak in Android framework. In particular, we uncover 36 exploitable cases, which can be abused by unauthorized apps to retrieve the sensitive data, violating the access control in Android framework.

REFERENCES

[1] 2022. 7-Zip. https://www.7-zip.org/.
[2] 2022. Android Debug Bridge (ADB). https://developer.android.com/studio/command-line/adb.
[3] 2022. Android IPC Security Considerations. https://chromium.googlesource.com/chromium/src.git/+/refs/heads/main/docs/security/android-ipc.md.
[4] 2022. android-simg2img. https://github.com/anestisb/android-simg2img.
[5] 2022. Android System Partition. https://source.android.com/devices/bootloader/partitions.
[6] 2022. android.content.ComponentName. https://developer.android.com/reference/android/content/ComponentName.
[7] 2022. android.debug.PairDevice. https://cs.android.com/android/platform/superproject/+/master:out/soong/.intermediates/frameworks/base/core/java/android.debug_aidl-java-source/gen/android/debug/PairDevice.java.
[8] 2022. android.net.NetworkInfo. https://developer.android.com/reference/android/net/NetworkInfo.
[9] 2022. android.telephony.TelephonyManager. https://developer.android.com/reference/android/telephony/TelephonyManager.
[10] 2022. AOSP. https://developers.google.com/android/images.
[11] 2022. ColorOS. https://www.coloros.com.
[12] 2022. Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters.
[13] 2022. Java Collections Framework Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.
[14] 2022. java.lang.Class. https://developer.android.com/reference/java/lang/Class.
[15] 2022. LADB, A local ADB shell for Android. https://github.com/tytydraco/LADB.
[16] 2022. MIUI. https://home.miui.com.
[17] 2022. Mobile Vendor Market Share Worldwide. https://gs.statcounter.com/vendor-market-share/mobile.
[18] 2022. OneUI. https://developer.samsung.com/one-ui.
[19] 2022. OriginOS. https://www.vivo.com/originos.
[20] 2022. Permissions overview. https://developer.android.com/guide/topics/permissions/overview.
[21] 2022. Soot. https://github.com/soot-oss/soot.
[22] 2022. System Broadcast Intents. https://developer.android.com/about/versions/12/reference/broadcast-intents-31.
[23] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proc. NDSS*.
[24] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API Protection Mapping Derivation and Reasoning. In *Proc. CCS*.
[25] Anshul Arora, Sateesh K Peddoju, and Mauro Conti. 2019. Permpair: Android malware detection using permission pairs. *IEEE Transactions on Information Forensics and Security* 15 (2019), 1968–1982.
[26] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*.

[27] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *Technical Report TUDCS-2013-0114* (2013).

[28] K. Au, Yifan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proc. CCS*.

[29] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proc. USENIX Security*.

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

[31] Hamid Bagheri, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek. 2021. Flair: efficient analysis of Android inter-component vulnerabilities in response to incremental changes. *Empirical Software Engineering* (2021).

[32] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. 2010. A Methodology for Empirical Analysis of Permission-Based Security Models and Its Application to Android. In *Proc. CCS*.

[33] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2014. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering* 40, 6 (2014), 617–632.

[34] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proc. AsiaCCS*.

[35] Maxim Chernyshev, Craig Valli, and Peter Hannay. 2015. On 802.11 access point locatability and named entity recognition in service set identifiers. *IEEE Transactions on Information Forensics and Security* (2015).

[36] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proc. MobiSys*. 239–252.

[37] Abdallah Dawoud and Sven Bugiel. 2019. DroidCap: OS support for capability-based permissions in android. In *Proc. NDSS*.

[38] Abdallah Dawoud and Sven Bugiel. 2021. Bringing balance to the force: Dynamic analysis of the android application framework. In *Proc. NDSS*.

[39] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP*.

[40] Zeinab El-Rewini and Yousra Aafer. 2021. Dissecting Residual APIs in Custom Android ROMs. In *Proc. CCS*.

[41] Karim O Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G Ryder. 2018. Identifying mobile inter-app communication risks. *IEEE Transactions on Mobile Computing* (2018).

[42] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware. In *Proc. USENIX Security*.

[43] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding android security. *IEEE security & privacy* 7, 1 (2009), 50–57.

[44] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proc. FSE*.

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

[46] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proc. CODASPY*.

[47] Youn Kyu Lee, Jae Young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A sealant for inter-app security holes in android. In *Proc. ICSE*.

[48] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-An, and Heng Ye. 2018. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3216–3225.

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

[50] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. CCS*.

[51] Lannan Luo. 2020. Heap Memory Snapshot Assisted Program Analysis for Android Permission Specification. In *Proc. SANER*.

[52] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proc. ICSE*.

[53] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proc. USENIX Security*.

[54] Octeau, Damien and Jha, Somesh and Dering, Matthew and McDaniel, Patrick and Bartel, Alexandre and Li, Li and Klein, Jacques and Le Traon, Yves. 2016. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proc. POPL*.

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

[56] Suranga Seneviratne, Fangzhou Jiang, Mathieu Cunche, and Aruna Seneviratne. 2015. SSIDs in the wild: Extracting semantic information from WiFi SSIDs. In *Proc. LCN*.

[57] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proc. NDSS*.

[58] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y Ko, and Lukasz Ziarek. 2014. Information flows as a permission mechanism. In *Proc. ASE*.

[59] Wenna Song, Jiang Ming, Lin Jiang, Yi Xiang, Xuanchen Pan, Jianming Fu, and Guojun Peng. 2021. Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization. In *Proc. CCS*.

[60] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. CCS*.

[61] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *Proc. USENIX Security*.

[62] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. 2022. Uncovering Cross-Context Inconsistent Access Control Enforcement in Android. In *Proc. NDSS*.

[63] Hao Zhou, Haoyu Wang, Shuohan Wu, Xiapu Luo, Yajin Zhou, Ting Chen, and Ting Wang. 2021. Finding the Missing Piece: Permission Specification Analysis for Android NDK. In *Proc. ASE*.