Uncovering Cross-Context Inconsistent Access Control Enforcement in Android

Hao Zhou[1], Haoyu Wang[2], Xiapu Luo[1*], Ting Chen[3*], Yajin Zhou[4] and Ting Wang[5]
[1]The Hong Kong Polytechnic University, [2]Beijing University of Posts and Telecommunications
[3]University of Electronic Science and Technology of China, [4]Zhejiang University, [5]Pennsylvania State University

***Abstract*—Due to the complexity resulted from the huge code base and the multi-context nature of Android, inconsistent access control enforcement exists in Android, which can be exploited by malware to bypass the access control and perform unauthorized security-sensitive operations. Unfortunately, existing studies only focus on the inconsistent access control enforcement in the Java context of Android. In this paper, we conduct the *first* systematic investigation on the inconsistent access control enforcement across the Java context and native context of Android. In particular, to automatically discover cross-context inconsistencies, we design and implement `IAceFinder`, a new tool that extracts and contrasts the access control enforced in the Java context and native context of Android. Applying `IAceFinder` to 14 open-source Android ROMs, we find that it can effectively uncover their cross-context inconsistent access control enforcement. Specifically, `IAceFinder` discovers 23 inconsistencies that can be abused by attackers to compromise the device and violate user privacy.**

I. Introduction

Modern operating systems (e.g., Windows, Linux, Android, iOS) commonly employ access control mechanisms to prevent unauthorized applications or users from accessing the sensitive system functions and the users' private data [73]. For example, Linux only allows the processes associated with specific user identifiers (UIDs) to access certain files [64]. As a Linux based operating system, Android not only adopts the UID based security model but also employs the permission based access control mechanism [48]. Precisely, it only permits the Android applications (including Android apps and native programs that are written by C or C++ code) that have specific UIDs and permissions to invoke its sensitive functions.

Unfortunately, due to the complexity and the huge code base of modern operating systems, it is not uncommon that developers fail to enforce consistent access control on sensitive system functions [77]. Such an inconsistency can be exploited by the unauthorized applications to perform security-sensitive operations to compromise privacy of user data or cause damage to the system. For instance, owing to the inconsistent UID check, malware can launch DoS attacks on Android's critical system services [43, 75]. In addition, due to the inconsistent permission check, malicious Android applications can steal users' private data [15, 52, 61, 75, 85].

*The corresponding authors.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23166
www.ndss-symposium.org

Android system services provide both Java and native interfaces for applications to invoke sensitive system functions, which can be divided into two types depending on the programming languages used to implement their core functionality, namely Java system services and native system services that are usually implemented in C++ code [69]. Android is expected to enforce consistent access control to restrict the invocations to these system services [46]. First, since Android applications can call the sensitive functions of system services through multiple ways in the same context [75], consistent access control should be enforced on all possible invocation interfaces to the same function. Second, since the sensitive functions of Java system services may rely on native system services to implement their functionality, the access control enforced in the Java context should be consistent with that in the native context.

Unfortunately, recent studies showed that inconsistent access control enforcement exist in Android's Java system services (i.e., the first scenario) [43, 61, 75]. However, to the best of our knowledge, none of existing work examines the inconsistent access control enforcement across the Java context and native context of Android (i.e., the second scenario).

In order to fill in the gap, in this paper, we conduct the *first* systematic investigation on the cross-context inconsistent access control enforcement in Android framework, and develop the Inconsistent Access control enforcement Finder (`IAceFinder`) to automatically uncover such inconsistencies. More precisely, `IAceFinder` first extracts the access control enforced in the Java context and that in the native context, and then contrasts them. The whole process consists of three steps. *First*, to identify the access control enforced in different contexts, `IAceFinder` performs static analysis on Java libraries (i.e., `.jar` files) and native libraries (i.e., `.so` files) of Android framework to construct the callgraphs for Java system services and native system services, respectively. *Second*, `IAceFinder` analyzes the callgraphs to find the access control enforced in Java system services and native system services. Since the sensitive functions of Java system services may employ the JNI interface [23, 72] to interact with native system services, the JNI interface bridges these two contexts of Android. A JNI interface contains a pair of *JNI method* and *JNI function*. The former is declared in the Java context and the latter is implemented in the native context [22, 80]. Accordingly, the access control enforced to restrict the invocation of the JNI method in Java system services should be consistent with the access control enforced to restrict the execution of the JNI function in native system services. Therefore, `IAceFinder` identifies the access control enforced on the JNI methods and that enforced on the JNI functions, respectively. *Third*, given a JNI interface, `IAceFinder` contrasts

the access control associated with its JNI method and the access control enforced on its JNI function to uncover the potential cross-context inconsistent access control enforcement.

It is challenging to design and develop `IAceFinder` due to the following technical issues. First, it is non-trivial to build the complete callgraph of native system services by analyzing its native libraries individually, because the functions in one native library may rely on the functions exported by the others. To approach this issue, when building the callgraph of a native library, we also analyze the dependent shared libraries (detailed in §VI-A). Second, since native system services are mainly implemented in C++, the polymorphism of C++ language makes it hard to accurately distinguish the objects of the classes that are inherited from the same parent class (especially for the interface class), thus degrading the quality of the callgraph built for native system services. To mitigate this problem, we carefully perform points-to analysis on native code when constructing the callgraph (detailed §VI-A). Third, since the interfaces accessible to Android applications use `Binder` to interact with native system services, there is no explicit function calls from the interfaces to the functions of native system services, and thus the calling relationships between them are missed in the callgraph. To address this issue, we conduct static analysis on `Binder` of native system services to restore the edges that connect the interfaces to the corresponding functions of native system services in the callgraph (detailed in §VI-A).

We use `IAceFinder` to discover the cross-context inconsistent access control enforcement in 14 open-source Android distributions, including the recently released official Android systems and other third-party Android ROMs (e.g., LineageOS [24]). `IAceFinder` uncovers 23 inconsistencies which can be exploited by adversaries to compromise the device or invade user privacy. We have reported the discovered inconsistencies to Google and other ROMs' maintainers and got rewards from Google vulnerability reward program.

In summary, we make the following contributions:

- To the best of our knowledge, we are the *first* to investigate the inconsistent access control enforcement across the Java context and native context of Android framework.
- We develop `IAceFinder`, a new tool to automatically uncover the cross-context inconsistent access control enforcement after tackling several technical challenges. Our tool will be released at https://github.com/moonZHH/IAceFinder.
- We extensively evaluate the performance of `IAceFinder` by applying it to 14 open-source Android ROMs. `IAceFinder` discovers 23 cross-context inconsistent access control enforcement that can be abused to compromise the device and invade user privacy.

II. Background

This section introduces two types of access control enforced by Android in §II-A and Android system services in §II-B. To explain the causes of cross-context inconsistent access control enforcement (in §III), we describe how Android applications interact with Android system services and how different system services interact with each other in §II-C.

A. Access Control

Android mainly employs permission checks and UID checks to protect their sensitive functions [44, 46, 47, 83]. Therefore, in this paper, we mainly focus on analyzing the permission based and UID based access control.

• **Permission**: Android employs the permission based access control model [48, 74, 75], which requires Android applications to gain the necessary permissions to access private user data (e.g., contacts), retrieve sensitive device information (e.g., microphone's states), or use critical system features (e.g., camera) [34]. Depending on the protection level that characterizes the potential risk implied in the permission [8], Android divides permissions [32] into four categories. The permissions with the protection levels `privileged` and `signature` can only be gained by the privileged system applications (e.g., system services and ADB shell [1, 2]), and thus they are associated with high privilege [43]. Since the permissions with the protection levels `dangerous` and `normal` can be obtained by normal applications, they are associated with low privilege [43].

TABLE I: Partial UIDs in Android framework.

UID	Alias	Value	Description
ROOT_UID	AID_ROOT	0	For root user.
SYSTEM_UID	AID_SYSTEM	1,000	For system user.
SHELL_UID	AID_SHELL	2,000	For shell user.
FIRST_APPLICATION_UID	AID_APP_START	10,000	For normal apps.
LAST_APPLICATION_UID	AID_APP_END	19,999	For normal apps.

• **User Identifier (UID)**: Android also employs the UID based access control model [75], which only allows the applications with the required UIDs to call the restricted system functions. Table I lists partial of the critical UIDs and their aliases used by Android. Specifically, for *normal application*s, their UIDs are in the range of $[10,000, 19,999]$, and the UIDs of privileged *system application*s (e.g., system services and ADB shell) are smaller than $10,000$. Generally, the privilege associated with the UIDs of privileged system applications is higher than that associated with the UIDs of normal applications [43]. It is worth to note that, ADB shell shares its UID (i.e., SHELL_UID) and permissions to the native programs executed by it.

B. Android System Services and Binder

Being the essential components of Android, Android system services can be accessed by Android applications through their interfaces [36, 84]. Since system services usually provide sensitive functions, Android enforces access control on their interfaces. Depending on the programming languages used to implement their functionality, system services are categorized into Java system services and native system services [69].

Since Android applications and system services run in separate processes, Android provides `Binder` [10], an inter-process communication (IPC) mechanism, for applications to interact with system services. Android applications use the Binder proxy to communicate with the corresponding Binder stub, which is a special object held by system services. Precisely, the applications invoke the methods (*local interface*s) defined in the classes of the Binder proxy to call the methods (*remote interface*s) defined in the classes of the Binder stub. Note that, the classes of a pair of Binder proxy and Binder stub inherit the

same interface class, where the interface methods are declared. Accordingly, a pair of local interface and remote interface share the same method name, parameter types, and return type.

C. Interacting with Android System Services

We use an example (in Fig.1) to explain the interactions between Android applications and system services and those between Java and native system services because they are essential to the cross-context access control enforcement.

Fig.1 shows the workflow for Android applications to create a secure virtual display. It involves Display Manager service [18] and SurfaceFlinger service [41]. The former is a Java system service that manages the virtual displays in Android framework, and the latter is a native system service that takes the charge of creating virtual displays.

• **Interaction between Android applications and Java system services**: To interact with Java system services (e.g., Display Manager service), applications use the Java Binder proxy (e.g., the `IDisplayManager$Stub$Proxy` object) to send the request to the Java Binder stub (e.g., the `IDisplayManager$Stub` object). This process (*Interact-J* in Fig.1) has three steps.

First, Android applications invoke the method `getService` defined in the class `ServiceManager` to get the Binder proxy. Then, they invoke the method `createVirtualDisplay` implemented in the class `IDisplayManager$Stub$Proxy` to request Display Manager service to create a virtual display (i.e., S1). Second, `createVirtualDisplay` (i.e., the local interface $I1_L$) invokes the method `transact` of the class `BinderProxy` to send the request to the Binder stub (i.e., S2). Third, to create the virtual display (i.e., S3), the method `onTransact` of the Binder stub processes the request, and then invokes `createVirtualDisplay` (i.e., the remote interface $I1_R$) of Display Manager service.

Since `createVirtualDisplay` provides sensitive functionality (i.e., creating a secure virtual display), Display Manager service enforces permission checks and UID checks (i.e., E1) on the applications, which use IPC to call this method. More about the enforced access control is introduced in §III-D.

• **Interaction between Android applications and native system services**: To interact with native system services (e.g., SurfaceFlinger service), applications use the native Binder proxy (e.g., the `BpSurfaceComposer` object) to send the request to the native Binder stub (e.g., the `BnSurfaceComposer` object). This process (*Interact-N* in Fig.1) consists of three steps.

First, Android applications call the function `getService` of the class `IServiceManager` to get the Binder proxy. Then, they call the function `createDisplay` of class `BpSurfaceComposer` to request SurfaceFlinger service to create a virtual display (i.e., S5). Second, `createDisplay` (i.e., the local interface $I2_L$) calls the function `transact` of the class `BpBinder` to send the request to the Binder stub (i.e., S6). Third, to create the virtual display (i.e., S7), the function `onTransact` of the Binder stub handles the request, and then calls the function `createDisplay` (i.e., the remote interface $I2_R$) of SurfaceFlinger service.

Similarly, since the execution of `createDisplay` results in the sensitive operation (i.e., creating a secure virtual display), SurfaceFlinger service enforces access control (i.e., E2) on the applications that use IPC to call this function. More about the access control enforcement is introduced in §III-D.

• **Interaction between two types of system services**: Since native system services usually provide more powerful functionality (e.g., accessing hardware [36]) than Java system services, the latter usually relies on the former to complete certain tasks. For example, to create a secure virtual display, Display Manager service uses Java Native Interface (JNI) to access the native context to request SurfaceFlinger service to do so. This process (*Interact-JN* in Fig. 1) includes two parts.

First, `createVirtualDisplay` of Display Manager service calls the JNI method `SurfaceControl.nativeCreateDisplay` to get access to the native context (i.e, S4-J). Since different contexts have distinct naming conventions for JNI interfaces [21], there is a one-on-one mapping between *JNI methods* and *JNI functions*. For instance, the corresponding JNI function of `nativeCreateDisplay` is `android::nativeCreateDisplay`. Android invokes `AndroidRuntime::registerNativeMethods`, or `RegisterMethodsOrDie`, or `jniRegisterNativeMethods` to record such the JNI method-function mapping. Second, since invoking the JNI method leads to the execution of the corresponding JNI function, `android::nativeCreateDisplay` will be executed (i.e., S4-N). Then, the `BpSurfaceComposer` object will be retrieved for requesting SurfaceFlinger service to create the secure virtual display (i.e., S5→S6→S7).

III. Cross-Context Inconsistency

This section presents the threat model of cross-context inconsistent access control enforcement in §III-A and introduces how we normalize different access control and contrast them in §III-B. After that, we reveal two types of inconsistencies in §III-C and present their motivating examples in §III-D.

A. Threat Model

According to the example in §II-C, an application has two paths to access `createDisplay` of SurfaceFlinger service. First, it uses Binder to call the interface `createVirtualDisplay` of DisplayManager service, which will internally invoke the JNI method `nativeCreateDisplay` to access `createDisplay`. Second, it uses Binder to call `createDisplay` directly. If the access control enforced in the two paths is different, it is a cross-context inconsistent access control enforcement.

Since existing studies [43, 61, 75] focus on the inconsistent access control enforcement in Java system services, their threat models only assume that attackers will implement the exploitation in Java code of Android apps. In contrast, our threat model extends theirs to include native programs because it is feasible to interact with Java system services through native code [69]. Since C++ code can be executed by either apps or native programs, we assume that adversaries can construct either of them to exploit the inconsistency. For example, adversaries may induce victims to install and launch the attack app or use ADB [2] to push and execute the native program. It is practical for adversaries to use native programs and ADB to exploit the inconsistency because they have been abused by malware to launch attacks as reported by `Trend Micro` [14, 31].

Fig.2 abstracts the example in §II-C and shows entities of the cross-context access control enforcement, including an Android application, a *deputy* (the remote interface of a Java system service), and a *target* (the remote interface of a native system service). The application has two paths to access the

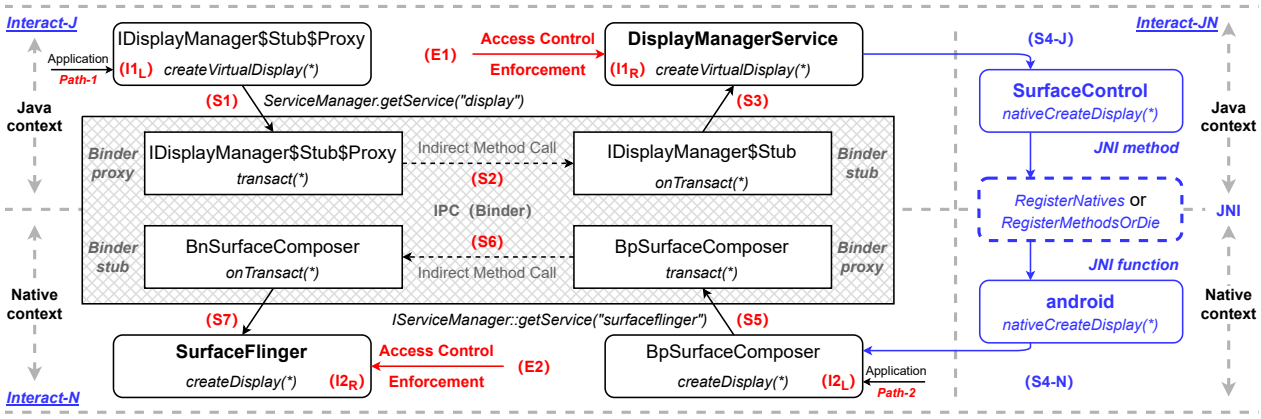


Fig. 1: The workflow of creating a secure virtual display involves three interactions with the Android system services, including an interaction between the application and the Java system service (i.e., *Interact-J*), an interaction between the application and the native system service (i.e., *Interact-N*), and an interaction between the Java system service and the native system service (i.e., *Interact-JN*).

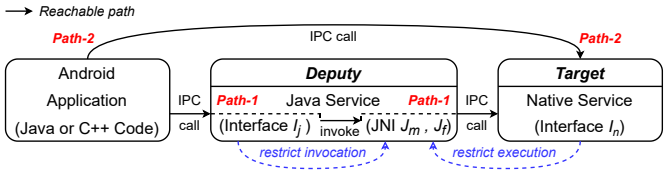


Fig. 2: Entities of cross-context inconsistency.

functionality provided by the native system service. **Path-1**: the application uses IPC to call the remote interface I_j of the Java system service to access the remote interface I_n of the native system service, which implies that I_j is the *deputy* of the *target* I_n. In detail, I_j internally invokes the JNI method J_m, whose corresponding JNI function J_f uses IPC to call I_n. **Path-2**: the application uses IPC to call I_n.

I_j and I_n both enforce access control on the calling process of IPC. More precisely, for **Path-1**, I_j enforces access control on J_m to restrict it from being invoked by the application. We call J_m an *access restricted JNI method*, whose invocation is restricted by access control. In addition, I_n enforces access control on J_f to restrict itself from being called by the Java system service. We call J_f an *access restricted JNI function*, whose execution leads to enforcement of access control. Since the Java system service is a privileged system application, it can pass the access control enforced in I_n. For **Path-2**, I_n enforces access control to restrict itself from being called by the application. Note that, I_n enforces the same access control on J_f (in *Path-1*) and the application (in *Path-2*).

We formally define the entities as follows:

$I_J :=$ set of all remote interfaces of Java system services,
$I_N :=$ set of all remote interfaces of native system services,
$J_M :=$ set of all JNI methods that will be invoked by remote interfaces of Java system services,
$J_F :=$ set of all JNI functions that will call remote interfaces of native system services, and
$\mathcal{J}(\cdot) := J_M \mapsto J_F$, the corresponding JNI functions of every JNI methods.

Definition 1 (*Reachable Path*)**.** For the remote interface $I_j \in I_J$ and the JNI method $J_m \in J_M$, there is a reachable path between them ($I_j \rightsquigarrow J_m$) if I_j internally invokes J_m. For the JNI function $J_f \in J_F$ and the remote interface $I_n \in I_N$, there is a reachable path between them ($J_f \rightsquigarrow I_n$) if J_f uses IPC to call I_n.

For example, in Fig.1, the execution flow S1→S2→S3 shows a reachable path between `createVirtualDisplay` (I_j) and the JNI method `nativeCreateDisplay` (J_m). The execution flow S5→S6→S7 indicates a reachable path between the JNI function `nativeCreateDisplay` (J_f) and `createDisply` (I_n).

Definition 2 (*Deputy*)**.** A remote interface I_j of Java system services ($I_j \in I_J$) is a deputy if $\exists\ J_m \in J_M$, $\exists\ J_f \in J_F$, and $\exists\ I_n \in I_N$ such that $I_j \rightsquigarrow J_m$, $\mathcal{J}(J_m) = J_f$, and $J_f \rightsquigarrow I_n$.

Definition 3 (*Target*)**.** A remote interface I_n of native system services ($I_n \in I_N$) is a target if $\exists\ I_j \in I_J$, $\exists\ J_m \in J_M$, and $\exists\ J_f \in J_F$ such that $I_j \rightsquigarrow J_m$, $\mathcal{J}(J_m) = J_f$, and $J_f \rightsquigarrow I_n$.

For instance, in Fig.1, `createVirtualDisplay` (I_j) is the *deputy* and `createDisplay` (I_n) is the *target*.

B. Contrasting Access Control

We contrast the access control enforced in the *deputy* I_j and that in the *target* I_n to discover cross-context inconsistencies. However, it is non-trivial to accomplish this task because of two reasons. First, permission checks and UID checks are in different forms, making it hard to compare them. Second, I_j and I_n can enforce multiple permission checks and UID checks, raising the difficulty of comparing the access control. To tackle them, we normalize diverse checks and then determine the necessary privilege of multiple checks.

• **Normalizing Permission Checks and UID Checks**: Since permissions and UIDs have equivalent semantics in terms of the privilege they entail [43], we follow the recent study [43] to normalize permission checks and UID checks to the checks on privileges. Compared with the previous work, we use a special privilege level to denote the permissions and UID held by ADB shell as attackers can run normal applications using ADB shell. As listed in Table II, we categorize the privileges into four levels, namely *normal*, *shell*, *system*, ∅. No permissions and UIDs are associated with ∅, and we use it to denote there is no access control. Note that, we use `IAceFinder` to identify inconsistencies at privilege levels rather than at permission and

(or) UID levels because the latter produces many false positives as pointed out by the previous work [43].

In detail, normal applications can request the permissions in either of the protection levels `normal` and `dangerous` (see §II-A). Therefore, we associate these permissions and the UIDs of normal applications (i.e., those in range $[10,000, 19,999]$) with the *normal* privilege. Additionally, ADB shell is granted with several permissions in the protection levels `signature` or `privileged` for debugging purposes [33]. Hence, we correlate these permissions and the UID of ADB shell (i.e., SHELL_UID) to the *shell* privilege. Note that, normal native programs can pass the check on the *shell* privilege if they are run by ADB shell (see §II-A). For the remaining UIDs (i.e., those for privileged system applications excluding ADB shell) and permissions that cannot be gained by normal applications, we associate them with the *system* privilege.

TABLE II: Privilege levels associated with permissions and UIDs.

Privilege	Access Control	Description
System	Permission UID	Those can only be granted to system applications. For system applications excluding ADB shell.
Shell	Permission UID	Those have been granted to ADB shell. For ADB shell, i.e., SHELL_UID or AID_SHELL.
Normal	Permission UID	Those can be granted to normal applications. For normal applications.
$\varnothing$	N/A	N/A

Among the privilege levels except $\varnothing$, the *system* privilege is the highest one as its corresponding permissions and UIDs are only associated with privileged system applications and they cannot be gained by normal applications. The *shell* privilege is lower. Although its corresponding permissions and UIDs are correlated with ADB shell, a privileged system application, they can be gained by normal applications that are run by ADB shell. The *normal* privilege is the lowest privilege among the three.

In summary, the relation among the privilege levels is: *system* $>$ *shell* $>$ *normal* $>$ $\varnothing$, where $>$ means higher privilege.

• **Determining Necessary Privilege**: Necessary privilege is defined as the least privilege level required to execute a reachable path, and we use it to check the cross-context inconsistent access control. Aafer et al. [44] found that permission checks and UID checks are usually disjoint. That is, the applications holding *any* of the permissions or UIDs being checked can pass the access control enforced in the *deputy* I_j or the *target* I_n. Therefore, the necessary privilege is the least one associated with the permissions or UIDs being checked. We use examples to elaborate more on it in §III-D.

```
01 status_t AudioFlinger::setMicMute(*) { // AudioFlinger.cpp
02   IPCThreadState* ipc = IPCThreadState::self();
03   const int uid = ipc->getCallingUid(); // retrieve UID
04   if (uid >= AID_APP_START) {⟵ shell privilege
05   // if (uid != AID_SYSTEM) { if (uid != AID_SHELL) { if (uid != ...) { ... } } }
06     return PERMISSION_DENIED; /* deny UIDs with normal privilege */ }
07   if (uid != AID_AUDIOSERVER) {⟵ system privilege
08     if (!checkPermission("MODIFY_AUDIO_SETTINGS"))⟵ normal privilege
09       return PERMISSION_DENIED; // do not have the required permission
10   }  /* ignore irrelevant code */  }
```

Fig. 3: The access control enforced on setMicMute (*target*).

However, we discover a class of special cases (as shown in Fig.3), where the check on the AID_APP_START UID (in Line 4) is conjoined with the other permission checks (in Line 8) and UID checks (in Line 7). For example, to call the remote interface `setMicMute`, applications must not run with the UIDs associated with the *normal* privilege, and they should either hold the MODIFY_AUDIO_SETTINGS permission (*normal* privilege) or run with the AID_AUDIOSERVER UID (*system* privilege). Since the UID check on AID_START_APP in Line 4 is semantically equivalent to a disjunction of checks on all UIDs associated with the *shell* and *system* privilege as shown in Line 5, we normalize it to the check on the *shell* privilege. The *shell* privilege and the privileges associated with other permissions and UIDs being checked are conjoined ($\wedge$) to determine the necessary privilege required to execute `setMicMute`. More specifically, the check on the higher privilege between the *shell* privilege and the lowest privilege among the privileges associated with other permissions and UIDs is the necessary privilege. To make it clear, we abstract the aforementioned process using the formula: $\mathit{shell} \wedge (\underline{\mathit{system} \vee \mathit{normal}}) = \mathit{shell} \wedge \underline{\mathit{normal}} = \mathit{shell}$. The necessary privilege required to execute `setMicMute` is the *shell* privilege.

Hence, to determine the necessary privilege, we consider both the common cases pointed out by Aafer et al. [44] and the special case found by us. We define *less restrictive* to denote the relation between two necessary privileges.

Definition 4 (*Less Restrictive*)**.** For necessary privileges p_1 and p_2, p_1 is less restrictive than p_2 if $p_1 < p_2$.

C. Two Types of Cross-Context Inconsistency

In this section, we introduce two types of cross-context inconsistent access control enforcement discovered by us.

• **Type-1 Inconsistency**: Fig.4 illustrates the *Type-1* cross-context inconsistent access control enforcement. The access control enforced in the target I_n is less restrictive than that in the deputy I_j, i.e., the privilege check on J_f is less restrictive than that on J_m. Therefore, the attack application can exploit the inconsistency to evade the stricter access control enforced in *deputy* by directly calling *target*.

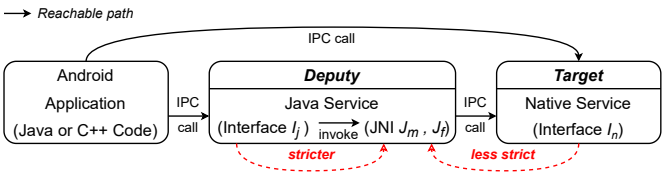


Fig. 4: Type-1 cross-context inconsistency.

In the following, we formally define the Type-1 inconsistency. For the definitions, let:

$\mathcal{P}(\cdot) \coloneqq (J_M \cup J_F) \mapsto \{system, shell, normal, \varnothing\}$, privilege checks (i.e., the necessary access control) on the JNI methods invoked by the remote interfaces of Java system services and the JNI functions that call remote interfaces of native system services.

Definition 5 (*Type-1 Inconsistency*)**.** If $\exists\, I_j \in I_J$, $\exists\, J_m \in J_M$, $\exists\, J_f \in J_F$, and $\exists\, I_n \in I_N$ such that $I_j \rightsquigarrow J_m$, $\mathcal{J}(J_m) = J_f$, $J_f \rightsquigarrow I_n$ and $\mathcal{P}(J_f) < \mathcal{P}(J_m)$, Type-1 inconsistency occurs.

• **Type-2 Inconsistency**: Fig.5 illustrates the *Type-2* cross-context inconsistent access control enforcement. The access control enforced in the deputy I_j is less restrictive than that in the target I_n, i.e., the privilege check on J_m is less restrictive than that on J_f. Thus, the attack application can exploit the inconsistency to pass the stricter access control enforced in *target* as if it was the Java system service by calling *deputy*.

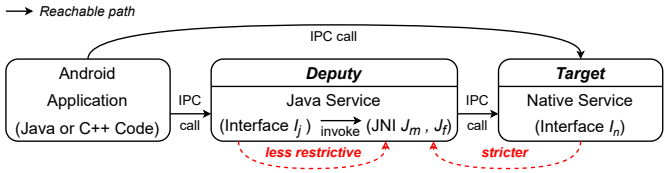


Fig. 5: Type-2 cross-context inconsistency.

We formally define the Type-2 inconsistency as follows:

Definition 6 (*Type-2 Inconsistency*)**.** If $\exists\ I_j \in I_J$, $\exists\ J_m \in J_M$, $\exists\ J_f \in J_F$, and $\exists\ I_n \in I_N$ such that $I_j \rightsquigarrow J_m$, $\mathcal{J}(J_m) = J_f$, $J_f \rightsquigarrow I_n$, and $\mathcal{P}(J_m) < \mathcal{P}(J_f)$, Type-2 inconsistency occurs.

Type-2 inconsistency is a new instance of confused deputy problems on Android. Although confused deputy problems on apps have been widely studied [51, 54, 60, 63, 66, 78, 79], Type-2 inconsistency is different from them and is a new instance, where the deputy is the system service rather than the app.

D. Motivating Examples of Cross-Context Inconsistency

• **Example of Type-1 Inconsistency**: The code snippet in Fig. 6a shows the access control (E1 in Fig.1) enforced in Display Manager service's `createVirtualDisplay` (*deputy*), restricting the JNI method `SurfaceContral.nativeCreateDisplay` to be invoked for creating the secure virtual display. The code snippet in Fig.6b shows the access control (E2 in Fig.1) enforced in SurfaceFlinger service's `createDisplay` (*target*), which restricts the execution of the `android::nativeCreateDisplay` JNI function and restricts itself from being called by applications.

More specifically, in Line 3-6 of Fig.6a, Display Manager service checks whether the application, requesting to create a secure virtual display, is running with the UID SYSTEM_UID (*system* privilege) or has been granted with the permission CAPTURE_SECURE_VIDEO_OUTPUT (*system* privilege). Thus, the access control on the JNI method is the check on the *system* privilege, which is derived by $system \vee system = \underline{system}$. In Line 4-7 of Fig.6b, SurfaceFlinger service examines whether the application, applying for the creation of a secure virtual display, is running with the UID AID_SYSTEM (*system* privilege) or has been granted with the permission ACCESS_SURFACE_FLINGER (*shell* privilege). The access control on the JNI function is the check on the *shell* privilege derived by $system \vee shell = \underline{shell}$.

Since the access control enforced on the JNI function (i.e., the check on the *shell* privilege) is less restrictive than that on the JNI method (i.e., the check on the *system* privilege), a Type-1 inconsistency is found. Accordingly, the attack application can call the *target* to create a secure virtual display, evading the stricter access control enforced in the *deputy*. The created secure virtual display can be misused to steal the user's sensitive data, such as passwords for logging in online payment apps. We present a use case of this inconsistency in §VIII-D.

```
01 public int createVirtualDisplay(*) { // DisplayServiceManager.java
02   int callingUid = Binder.getCallingUid(); // retrieve UID
03   if (callingUid != Process.SYSTEM_UID && ...) { ← system privilege
04     if (!checkCallingPermission(CAPTURE_SECURE_VIDEO_OUTPUT, *)) {
05       throw new SecurityException(*);          ↑
06   } } /* ignore irrelevant code */   system privilege
07   SurfaceControl.createDisplay(*); // call SurfaceControl.nativeCreateDisplay }
```

(a) The access control enforced in createVirtualDisplay (*deputy*).

```
01 bool callingThreadHasUnscopedSurfaceFlingerAccess() { // SurfaceFlinger.cpp
02   IPCThreadState* ipc = IPCThreadState::self();
03   const int uid = ipc->getCallingUid(); // retrieve UID
04   if (uid != AID_SYSTEM) { ←system privilege
05     if (!PermissionCache::checkPermission("ACCESS_SURFACE_FLINGER"))
06       return false; // do not have the required permission      ↑
07   } /* ignore irrelevant code */                          shell privilege
08   return true; // pass the UID check and the permission check   }
```

(b) The access control enforced in createDisplay (*target*).

Fig. 6: An example of the Type-1 inconsistency.

• **Example of Type-2 Inconsistency**: Fig.7a shows the code snippet of Audio service's interface `setRingerModeExternal` (*deputy*), where no access control is enforced to restrict the JNI method `AudioSystem.setForceUse` from being invoked to set the audio routing for the vibrate ringtone (in Line 3). The code snippet in Fig.7b shows the access control enforced in AudioPolicy service's interface `setForceUse` (*target*), which restricts itself from being called by Android applications and the JNI function `AudioSystem::setForceUse`.

More precisely, since no permission checks and UID checks are enforced on the JNI method, its access control is the check on $\varnothing$. In Line 4-7 of Fig.7b, AudioPolicy service examines whether the application, requesting to set the audio routing for a specific usage, is running with the UID AID_AUDIOSERVER (*system* privilege) or has been granted with the permission MODIFY_AUDIO_ROUTING (*system* privilege). Therefore, the access control on the JNI function is the check on the *system* privilege, which is derived from $system \vee system = \underline{system}$.

```
01 public void setRingerModeExternal(*) { // AudioService.java
02   // no access control to restrict the invocation of the JNI method setForceUse
03   AudioSystem.setForceUse(FOR_VIBRATE_RINGING, FORCE_NONE);
04   /* ignore irrelevant code */   }
```

(a) The access control enforced in setRingerModeExternal (*deputy*).

```
01 status_t AudioPolicyService::setForceUse(*) { // AudioPolicyService.cpp
02   IPCThreadState* ipc = IPCThreadState::self();
03   const int uid = ipc->getCallingUid(); // retrieve UID
04   if (uid != AID_AUDIOSERVER) {← system privilege
05     if (!checkPermission("MODIFY_AUDIO_ROUTING"))← system privilege
06       return PERMISSION_DENIED; // do not have the required permission
07   }   /* ignore irrelevant code */   }
```

(b) The access control enforced in setForceUse (*target*).

Fig. 7: An example of the Type-2 inconsistency.

Since the access control enforced on the JNI method (i.e., no privilege check) is less restrictive than that enforced on the JNI function (i.e., the check on the *system* privilege), a Type-2 inconsistency is found. Accordingly, the attack application can call the *deputy* to access the remote interface `setForceUse` of AudioPolicy service as if it is the privileged system service,

passing the stricter access control enforced in the *target*. As a result, attackers can silence the vibrate ringtone, which should have been output by an audio device (e.g., Bluetooth headset) when the Android smartphone is on the vibrate mode.

Although various work has been proposed to find vulnerabilities in Android system services [43, 61, 62, 69, 70, 75], to our best knowledge, *none of them* can discover the cross-context inconsistent access control enforcement, because they neither studied the access control enforced in native system services nor investigated the interactions between Java system services and native system services.

To fill in the gap, we design and develop `IAceFinder`, a new tool for discovering the cross-context inconsistent access control enforcement. Specifically, for the example in Fig.6, `IAceFinder` identifies the access control enforced in `createVirtualDisplay` (*deputy*) and `createDisplay` (*target*) at first. Then, the tool associates the identified access control enforcement to the JNI interface `nativeCreateDisplay` and analyzes them to determine the necessary privilege. Since the privilege check on the JNI function `android::nativeCreateDisplay` (i.e., the access control enforced in the *target*) is less restrictive than that on the JNI method `SurfaceControl.nativeCreateDisplay` (i.e., the access control enforced in the *deputy*), `IAceFinder` uncovers a Type-1 cross-context inconsistency.

IV. IAceFinder

In this section, we introduce the overview and the workflow of `IAceFinder` in §IV-A and §IV-B, respectively.

A. Overview

Fig.8 shows the architecture of `IAceFinder`, which has three modules, including *Module-J* (detailed in §V), *Module-N* (detailed in §VI), and *Module-D* (detailed in §VII). `IAceFinder` analyzes both Java system services and native system services to discover cross-context inconsistencies.

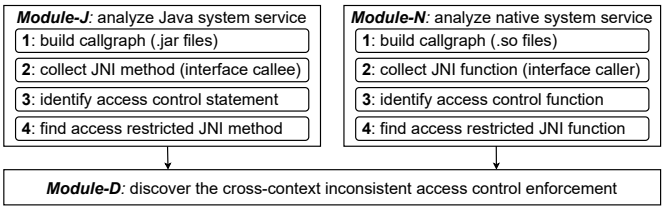


Fig. 8: The overview of IAceFinder.

Module-J analyzes Java system services to associate the access control enforced in their remote interfaces to JNI methods. Built upon `Soot` [40], a static Java bytecode analysis framework, *Module-J* inspects the `.jar` files compiled from the Java code of Android framework.

Module-N analyzes native system services to correlate the access control enforced in their remote interfaces to JNI functions. Built upon SVF [76], a static LLVM bitcode analysis framework, *Module-N* inspects the LLVM bitcode of the `.so` files compiled from the C/C++ code of Android framework.

Module-D contrasts the access control on a pair of JNI method and JNI function to uncover the cross-context inconsistent access control enforcement. Specifically, this module takes in the analysis results of *Module-J* and *Module-N* and the mapping between JNI methods and JNI functions.

B. Workflow

Fig.8 also presents the brief workflow of each module in `IAceFinder`, and we elaborate more on them as follows.

Module-J takes four steps to associate the access control enforced in Java system services to JNI methods. First, it builds the callgraph of the services to find the access control enforced in Java system services. Second, it traverses the callgraph to collect the JNI methods that can be invoked by the remote interfaces of Java system services for the purpose of identifying the access restricted JNI methods, whose invocation is restricted by access control enforcement. Third, it analyzes each method in the callgraph to find the statements that enforce access control (e.g., the invocation of `checkCallingPermission` in Line 4 of Fig.6a) for the sake of determining the access control enforced on the access restricted JNI methods. Fourth, it performs control dependence analysis [45] on the JNI methods and the statements found in the previous steps to identify the access restricted JNI methods and the access control enforced on them.

Module-N takes four steps to correlate the access control enforced in native system services to JNI functions. First, it builds the callgraph of the services to find the access control enforced in native system services. Second, to find the JNI functions that correspond to the JNI methods collected by *Module-J*, it analyzes the registration processes of JNI interfaces and maps each JNI method to its corresponding JNI function for the sake of uncovering cross-context inconsistencies. Moreover, it traverses the callgraph to collect the JNI functions that call remote interfaces of native system services for the purpose of identifying access restricted JNI functions, whose execution leads to enforcement of access control. Third, it analyzes each function in the callgraph to find the access control functions that enforce access control (e.g., the caller of `checkPermission` in Line 5 of Fig.6b) for the sake of determining the access control enforced on the access restricted JNI functions. Fourth, it traverses the callgraph from each JNI function to the access control functions to identify the access restricted JNI functions and the access control enforced on them.

Module-D takes in the access control enforced on each JNI method and JNI function. It contrasts the access control (i.e., privilege checks) on each pair of JNI method and JNI function to discover cross-context inconsistencies.

V. *Module-J*: Analyzing Java System Service

This section elaborates on *Module-J*, including how it builds the callgraph for Java system services (in §V-A), collects the JNI methods that are invoked by remote interfaces of Java system services (in §V-B), identifies the access control statements (in §V-C), and finds the access restricted JNI methods (in §V-D).

A. Building Callgraph of Java System Services

Since we follow the same steps as the previous work [75] to build the callgraph for each Java system service, the details are left in Appendix-A. During this process, we also collect the remote interfaces of Java system services, which will be used to identify the JNI methods invoked by them, and save each found remote interface I_j to the set I_J.

B. Collecting JNI Methods

Module-J analyzes the callgraph to collect the JNI methods that will be invoked by the remote interfaces of Java system services. Precisely, it traverses the callgraph from each remote interface $I_j \in I_J$ to find the target JNI method J_m. If there is a reachable path between them ($I_j \rightsquigarrow J_m$), a target JNI method is found. We save each found J_m to the set J_M, which will be used to discover cross-context inconsistencies (see §VII).

C. Identifying Access Control Statements

Since we inspect the permission based and the UID based access control (as described in §II-A), we employ the approaches proposed in [43, 62, 75] to identify the statements that enforce permission checks and UID checks in remote interfaces of Java system services. The details of this process are in Appendix-B.

For each identified permission check statement S_p, we record the permission p being checked and store them in the statement-permission map $M_{sp} := \{S_p \mapsto p\}$. For each identified UID check statement S_u, we record the value of the UID u being checked and store them in the statement-UID map $M_{su} := \{S_u \mapsto u\}$. M_{sp} and M_{su} will be used to determine the access control enforced on JNI methods (see §V-D).

D. Finding Access Restricted JNI Methods

Module-J finds the access restricted Java methods, whose invocation is control dependent on the access control statements, to identify the access restricted JNI methods, which are called by the access restricted Java methods. It performs intra-procedure control flow analysis to find the access restricted Java methods. For instance, in Fig.6a, since the UID check statement in Line 3 and the permission check statement in Line 4 decide whether the method invocation in Line 7 will be executed or not, the invocation of the `SurfaceControl.createDisplay` method is control dependent on those access control statements, and thus `createDisplay` is identified as an access restricted Java method.

Once an access restricted Java method is found, *Module-J* traverses the callgraph to find the JNI methods invoked by the Java method. These JNI methods are the access restricted ones, because invoking them is also control dependent on the statements that restrict the invocation of the Java method. For example, since `SurfaceControl.createDisplay` invokes the JNI method `nativeCreateDisplay`, this JNI method is an access restricted one, because its execution is restricted by the permission check statement and the UID check statement presented in Line 3-4 of Fig.6a.

It is noteworthy that there may be a few false positives in the identified access restricted JNI methods because of two reasons. First, we treat all JNI methods that are reachable from the access restricted Java methods as the access restricted JNI methods. For example, since the JNI method `println_native` defined in the class `android.util.Log` is frequently invoked by the access restricted Java methods to perform the logging operation, we wrongly treat such a security-insensitive utility method as an access restricted JNI method. Second, we treat all JNI methods that are control dependent on the access control statements as the access restricted JNI methods. For example, in Fig.9, since the invocations to the JNI methods `getMasterMute` (in Line 6) and `setMasterMute` (in Line 7) are control dependent on the permission check statement in Line 3, both of them are considered as the access restricted methods. However, since `getMasterMute` is a normal method rather than a sensitive one as `setMasterMute`, we wrongly treat `getMasterMute`, which is not the target the access control statement intend to protect, as an access restricted JNI method.

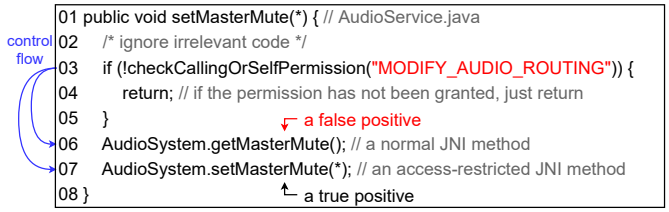


Fig. 9: A false-positive case of access restricted JNI methods.

To reduce the false positives caused by the first reason, we ignore the identified access restricted JNI methods defined in the packages (e.g., *android.content.res* [6], *android.graphics* [7], *android.util* [9]) that just provide security-insensitive utility methods. For the false positives caused by the second reason, we observe that there are usually reachable paths from remote interfaces of Java system services to them without access control. For example, `getMasterMute` can be invoked by `isMasterMute` of Audio service with no access control. Based on this observation, given an access restricted JNI method, we first check whether such paths exist (i.e., whether it is reachable by a remote interface of a Java system service with no access control). If so, it is a false positive and will be removed.

For each access restricted JNI method J'_m ($J'_m \in J_M$), we correlate J'_m with a set of access control S_m enforced on it. Specifically, we query M_{sp} and M_{su} to retrieve the permissions and UIDs examined in the access control statements that J'_m control dependents on. For example, S_m for `nativeCreateDisplay` is (`CAPTURE_SECURE_VIDEO_OUTPUT`, SYSTEM_UID). We store such the correlation to the map $M_m := \{J'_m \mapsto S_m\}$, which will be used to identify cross-context inconsistencies (see §VII).

VI. *Module-N*: Analyzing Native System Service

This section elaborates on *Module-N*, including how it builds the callgraph for native system services (in §VI-A), collects the JNI functions that call remote interfaces of native system services (in §VI-B), identifies the access control functions (in §VI-C), and finds the access restricted JNI functions (in §VI-D).

A. Building Callgraph of Native System Services

Since the implementations of native system services are dispersed in native system libraries (e.g., `libgui.so` contains the code for SurfaceFlinger service and `libinput.so` contains the code for InputFlinger service), we build the callgraph for each native library and then merge them together to form the complete callgraph. This process consists of two steps.

In the first step, we use SVF [76] to build the callgraph of each library based on its LLVM bitcode. During this process, we tackle the problems of differentiating interface classes and resolving virtual function calls in analyzing LLVM bitcode to make the callgraph more complete. We will elaborate how we address these two problems in the following. Since local

interfaces of native system services use Binder to access their corresponding remote interfaces, there are no explicit function invocations from local interfaces to remote interfaces, making SVF unable to identify the reachable paths among them correctly. Then, in the second step, we add callgraph edges to connect local interfaces to the corresponding remote interfaces.

• **LLVM Bitcode of Native Libraries**: We address two issues when analyzing the LLVM bitcode of native libraries to make the callgraph more accurate and complete.

```
/* A summary of nativeCreateDisplay in libandroid_runtime.so */
01 static jobject nativeCreateDisplay(env, *) { // android_view_SurfaceControl.cpp
02     sp<IServiceManager> manager = defaultServiceManager();
03     const string16 name("SurfaceFlinger"); // name of SurfaceFlinger service
04     sp<ISurfaceComposer> composer = service->getService(name);
05     return javaObjectForIBinder(env, composer->createDisplay(*));    }
/* Class definitions for IServiceManager and BpServiceManager in libbinder.so */
06 class IServiceManager : public IInterface {  // IServiceManager.h
07     int placeholder[100]; // distinguish IServiceManager from ISurfaceComposer
08     /* declarations of pure virtual functions */
09     /* definitions of static fields */    }
10 class BpServiceManager : public IServiceManager { /* ignore the code */ }
/* Class definition for ISurfaceComposer and BpSurfaceComposer in libgui.so */
11 class ISurfaceComposer : public IInterface {  // ISurfaceComposer.h
12     int placeholder[200]; // distinguish ISurfaceComposer from IServiceManager
13     /* declarations of pure virtual functions */    }
14 class BpSurfaceComposer : public ISurfaceComposer { /* ignore the code */ }
```

(a) The relevant C++ code of nativeCreateDisplay.

```
01 %"class.android::sp" = type { %"class.android::IServiceManager"* }
02 /* no vtable is included in the LLVM-bitcode */
03 define void @nativeCreateDisplay(*) { // implementation of nativeCreateDisplay
04     %manager = alloca %"class.android::sp" // define variable "manager"
05     %composer = alloca %"class.android::sp" // define variable "composer"
06     %vtable = * // get vtable for the type of %manager
07     %vfn = *, i32 4 // get the 5th element in vtable (currently unknown)
08     %vtable2 = * // get vtable for the type of %composer
09     %vfn3 = *, i32 6 // get the 7th element in vtable2 (currently unknown)
10     /* ignore the irrelevant LLVM-bitcode */    }
```

(b) The original LLVM bitcode of nativeCreateDisplay.

```
01 %"class.android::sp" = type { %"class.android::IServiceManager"* }
02 %"class.android::sp.1" = type { %"class.android::ISurfaceComposer"* }
03 @_ZTVN7android16BpServiceManager = * // vtable for BpServiceManager
04 @_ZTVN7android17BpSurfaceComposer = * // vtable for BpSurfaceComposer
05 define void @nativeCreateDisplay(*) { // implementation of nativeCreateDisplay
06     %manager = alloca %"class.android::sp" // define variable "manager"
07     %composer = alloca %"class.android::sp.1" // define variable "composer"
08     %vtable = * // get vtable for the type of %manager
09     %vfn = *, i32 4 // get the 5th element in vtable (i.e., getService)
10     %vtable2 = * // get vtable for the type of %composer
11     %vfn3 = *, i32 6 // get the 7th element in vtable2 (i.e., createDisplay)
12     /* ignore the irrelevant LLVM-bitcode */    }
```

(c) The updated LLVM bitcode of nativeCreateDisplay.

Fig. 10: The LLVM bitcode of libandroid_runtime.so.

▷ **Distinguishing Interface Classes**: LLVM bitcode represents a C++ class using the types of its non-static fields [27]. In some cases, such representation cannot distinguish the classes that inherit the same parent class. For example, in Fig.10a, since both the class `IServiceManager` (in Line 6-9) and the class `ISurfaceComposer` (in Line 11-13) inherit the class `IInterface` and none of them has additional non-static fields, these two classes have the same LLVM bitcode representation, which makes SVF unable to correctly determine the types of their objects. For instance, the LLVM bitcode in Fig.10b Line 4-5 indicates that the variables `manager` and `composer` have the same type. However, it is incorrect according to the corresponding C++ code in Line 2,4 of Fig.10a, which show that their types are different. Since SVF relies on the type information to conduct points-to analysis [76], this problem negatively influences the analysis and makes the callgraph inaccurate.

To tackle this issue, we make the LLVM bitcode representations of interface classes differ from each other by adding extra non-static fields to such classes. Precisely, we insert integer arrays with various length to different interface classes. For example, in Fig.10a, we add an array with 100 elements (in Line 7) to `IServiceManager` and an array with 200 elements (in Line 12) to `ISurfaceComposer`, respectively. As a result, Line 1-2, 6-7 of Fig.10c show that the LLVM bitcode representations for the types of variables `manager` and `composer` become different, and thus these two objects can be distinguished by SVF.

▷ **Linking Shared Libraries**: When compiling C++ code to LLVM bitcode, an additional variable, representing the virtual function table (a.k.a vtable), will be added to the LLVM bitcode of the class that implements virtual functions [27]. Since the variable contains the information (e.g., function names) about the virtual functions, this variable is important for resolving virtual function calls. For example, SVF identifies the function call to `getService` (in Line 4 of Fig.10a) by parsing the value of the variable `@_ZTVN7android16BpServiceManagerE` (in Line 3,9 of Fig.10c), representing the vtable of `BpServiceManager`.

However, since shared libraries do not have the implementations of their dependent libraries, the LLVM bitcode of shared libraries does not contain the LLVM bitcode of their dependent libraries. For instance, since the function `nativeCreateDisplay` is implemented in `libandroid_runtime.so` whereas the class `BpServiceManager` is defined in `libbinder.so`, the LLVM bitcode of `nativeCreateDisplay` does not include the variable `@_ZTVN7android16BpServiceManagerE` as presented in Fig.10b. Therefore, `SVF` cannot resolve the function call to `getService`. This issue makes the callgraph incomplete because numerous virtual function calls remain unresolved.

To address this issue, we link the LLVM bitcode of shared libraries with those of their dependent libraries. As a result, the LLVM bitcode of shared libraries contains the LLVM bitcode of their dependent libraries. More specifically, we first use `llvm-objdump` [26] to get the dependent libraries of each shared library. Then, we use `llvm-link` [25], a LLVM bitcode linker, to merge the LLVM bitcode of the dependent libraries to the LLVM bitcode of the shared library. For example, once we link the LLVM bitcode of `libandroid_runtime.so` with those of `libbinder.so` and `libgui.so`, the updated LLVM bitcode (Fig.10c) has the variables (Line 3-4), containing the information for resolving the virtual function calls. Consequently, SVF can build a more complete callgraph.

• **Callgraph**: We add the missing callgraph edges associated with Binder IPC by collecting local and remote interfaces of native system services and then connect them in the callgraph.

▷ **Collecting Local Interfaces**: Since local interfaces will call the `transact` function (see §II-C), we treat the callers of this function (e.g., $I2_L$: `BpSurfaceComposer::createDisplay` in Fig.1) as local interfaces and find them from the callgraph.

▷ **Collecting Remote Interfaces**: Remote interfaces share the same function names, parameter types, and return types as their corresponding local interfaces (see §II-B). Meanwhile, they are invoked by the function `onTransact` (see §II-C). Based on these, we first find the callees of `onTransact` (e.g., $I2_R$: `SurfaceFlinger::createDisplay` in Fig.1). Then, we compare each callee's function name, parameter types, and return type with each of the collected local interfaces to find whether the callee is a remote interface and correlate it to its corresponding local interface. We save each remote interface I_n to the set I_N.

▷ **Adding Edges**: Since the execution of local interfaces will result in the execution of their corresponding remote interfaces (see §II-C), there is a reachable path from each local interface to its corresponding remote interface. Hence, we add the callgraph edges that connect local interfaces to remote interfaces.

B. Collecting JNI Functions

We collect the target JNI functions that call remote interfaces of native system services by finding all JNI functions in native context and then identifying those having reachable paths to the remote interfaces from the callgraph.

• **Finding All JNI Functions**: Android framework provides the `JNINativeMethod` structure for developers to declare the mapping between JNI methods and their corresponding JNI functions [23]. More specifically, `JNINativeMethod` is composed of three elements storing the name, the subsignature (including parameter types and return type) of the JNI method, and the function pointer (i.e., function name) of the JNI function, respectively. Meanwhile, as introduced in §II-C, to register the mapping to Android runtime, Android framework calls the functions `jniRegisterNativeMethods`, or `RegisterMethodsOrDie`, or `registerNativeMethods`. Note that the second parameter of these functions offers the package name of JNI method and the third parameter is an array of `JNINativeMethod` structures.

Based on the aforementioned observations, we find all JNI functions and correlate them to their corresponding JNI methods by two steps. First, from LLVM bitcode, we analyze each `JNINativeMethod` array to find the JNI functions and get the basic information (e.g., names and parameter types) about the corresponding JNI methods. Second, we retrieve the package names of JNI methods for uniquely identifying them by their package names, method names, and subsignatures. Precisely, for each `JNINativeMethod` array, we conduct data flow analysis to find the function (e.g., `RegisterMethodsOrDie`) that consumes it and analyze the second parameter to get the package name.

Once we find a mapping between a JNI method J_m and a JNI function J_f, we store it to the map $\mathcal{J} := \{J_m \mapsto J_f\}$. The mapping will be used to discover the cross-context inconsistent access control enforcement (see §VII).

• **Identifying Target JNI Functions**: We analyze the callgraph to collect the target JNI functions that call remote interfaces of native system services. They will be used for finding the access restricted JNI functions, whose execution leads to enforcement of access control (see §VI-D). Specifically, we traverse the callgraph from each JNI function J_f to find whether there is a reachable path to $I_n \in I_N$ ($J_f \rightsquigarrow I_n$). If so, J_f is the target JNI function and we save it to the set J_F, which is then used to discover cross-context inconsistencies (see §VII).

C. Identifying Access Control Functions

To determine the permission checks and UID checks on the access restricted JNI functions, we first identify the access control functions, where the permission checks or UID checks are enforced (detailed in the following), and then correlate them to the JNI functions (see §VI-D).

• **Finding Permission Check Based Functions**: Android offers several permission check functions (e.g., `checkPermission` in Line 5 of Fig.6b) for native system services to examine whether the calling process of IPC has been granted with the required permission. The permission under check is passed as a parameter to permission check functions, and each permission is represented as a string constant in Android framework [75].

Based on this insight, we treat the callers of these permission check functions as permission check based access control functions (e.g., `callingThreadHasUnscopedSurfaceFlingerAccess` in Fig.6b), which conduct permission checks to enforce access control. We identify these access control functions via two steps. First, we find the permission strings in LLVM bitcode. Second, for each permission string, we apply data flow analysis to get the permission check function that consumes it, and then get the function's callers from the callgraph.

For each identified permission check based access control function F_p, we record the permission p being checked and store them in a function-permission map $M_{fp} := \{F_p \rightarrow p\}$, which will be used to determine the access control enforced on JNI functions (see §VI-D).

• **Finding UID Check Based Functions**: Android takes two steps to enforce the UID check. First, it retrieves the UID of the calling process of IPC. For instance, in Line 3 of Fig.6b, native system services call `IPCThreadState::getCallingUid` to retrieve the UID. Second, the retrieved UID is compared with a constant representing a specific UID (referring to Table I). For example, the `if` statement in Line 4 of Fig.6b checks whether the UID of the calling process of IPC equals to AID_SYSTEM.

Based on this observation, we regard the functions that enforce UID checks as the UID check based access control functions (e.g., `callingThreadHasUnscopedSurfaceFlingerAccess` in Fig.6b). We identify these access control functions through two steps. First, we retrieve the callers of `getCallingUid` from the callgraph. Second, in the LLVM bitcode of each caller, we perform intra-procedure data flow analysis on the return value of `getCallingUid` (i.e., the retrieved UID) to find whether it is used by an `if` statement. If so, the caller is a UID check based access control function.

For each identified UID check based access control function F_u, we record the UID u being checked and store them in a function-UID map $M_{fu} := \{F_u \mapsto u\}$ that is used to determine the access control enforced on JNI functions (see §VI-D).

D. Finding Access Restricted JNI Functions

To identify the access restricted JNI functions, we traverse the callgraph from each JNI function J_f ($J_f \in J_F$) to find whether there are reachable paths from it to the access control functions in M_{fp} or M_{fu}. If so, the JNI function under analysis is an access restricted JNI function. For example, since the execution of `nativeCreateDisplay` results in the invocation of

TABLE III: The open-source Android distributions under analysis.

	Name	Version	Description	#Binder$_j$	#J$'_m$	Precis$'_m$	#J$_m$	Precis$_m$	Δ#J$_m$	ΔPrecis$_m$	#Binder$_n$	#J$_f$	Precis$_f$	#Inconsistency
1	AOSP	Android 10	Official Android system	519	70	72.9%	57	89.5%	-13	+16.6%	108	55	94.5%	18
2	AOSP	Android 11	Official Android system	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
3	LineageOS	Android 10	AOSP based Android distribution	533	71	73.2%	58	89.7%	-13	+16.5%	108	56	94.6%	19
4	OmniROM	Android 10	AOSP based Android distribution	519	70	72.9%	57	89.5%	-13	+16.6%	108	55	94.5%	18
5	CAF	Android 11	AOSP based Android distribution	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
6	BlissROM	Android 11	AOSP based Android distribution	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
7	PixelExperience	Android 11	AOSP based Android distribution	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
8	GrapheneOS	Android 11	AOSP based Android distribution	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
9	DirtyUnicorns	Android 11	AOSP based Android distribution	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
10	NitrogenOS	Android 11	AOSP based Android distribution	566	79	74.7%	66	89.4%	-13	+14.7%	137	63	93.7%	22
11	Replicant	Android 10	LineageOS based Android distribution	533	71	73.2%	58	89.7%	-13	+16.5%	108	56	94.6%	19
12	crDroid	Android 11	LineageOS based Android distribution	579	80	75.0%	67	89.6%	-13	+14.6%	137	64	93.8%	23
13	EvolutionX	Android 11	LineageOS based Android distribution	579	80	75.0%	67	89.6%	-13	+14.6%	137	64	93.8%	23
14	MoKee	Android 10	CAF based Android distribution	519	70	72.9%	57	89.5%	-13	+16.6%	108	55	94.5%	18

`callingThreadHasUnscopedSurfaceFlingerAccess` (an access control function), it is an access restricted JNI function.

For each access restricted JNI function J'_f ($J'_f \in J_F$), we correlate J'_f with a set of access control enforced on it. More specifically, we query M_{fp} and M_{fu} to collect the permissions and UIDs examined in the access control functions that are reachable from J'_f. For example, S_f for `nativeCreateDisplay` is (`ACCESS_SURFACE_FLINGER`, AID_SYSTEM). We store this correlation to the map $M_f := \{J'_f \mapsto S_f\}$, which will be used to identify cross-context inconsistencies (see §VII).

VII. *Module-D*: Detecting Inconsistency

Module-D takes two steps to uncover inconsistencies. First, it computes the privilege checks on the access restricted JNI methods and JNI functions. Second, it contrasts the privilege checks on JNI methods and their corresponding JNI functions to discover cross-context inconsistencies.

In the first step, *Module-D* gets the set of access control enforced on each access restricted JNI method J'_m ($J'_m \in J_M$) and JNI function J'_f ($J'_f \in J_F$) from M_m and M_f, respectively. Then, it follows the criteria defined in §III-B to compute the privilege check P on J'_m or J'_f. For each of the remaining JNI method J''_m ($J''_m \neq J'_m \wedge J''_m \in J_M$) and JNI function J''_f ($J''_f \neq J'_f \wedge J''_f \in J_F$), since there is no access control enforced on it, its privilege check is $\varnothing$. We store such correlations to the map $\mathcal{P} := \{J'_m \mapsto P\} \cup \{J'_f \mapsto P\} \cup \{J''_m \mapsto \varnothing\} \cup \{J''_f \mapsto \varnothing\}$.

For example, for the JNI method `nativeCreateDisplay` of `SurfaceControl`, *Module-D* queries M_m to get the set of access control enforced on it, i.e., (`CAPTURE_SECURE_VIDEO_OUTPUT`, SYSTEM_UID). Then, it computes the privilege check (i.e., the check on the *system* privilege) as detailed in §III-D and saves the correlation `SurfaceControl.nativeCreateDisplay` $\mapsto$ *system* to $\mathcal{P}$. For the JNI function `android::nativeCreateDisplay`, *Module-D* queries M_f to retrieve the set of access control enforced on it, i.e., (`ACCESS_SURFACE_FLINGER`, AID_SYSTEM). Then, it computes the privilege check (i.e., the check on the *shell* privilege) as presented in §III-D and stores the correlation `android::nativeCreateDisplay` $\mapsto$ *shell* to $\mathcal{P}$.

In the second step, *Module-D* inspects $\mathcal{J}$ to get each pair of JNI method J_m and JNI function J_f ($J_m \in J_M$, $J_f \in J_F$, $\mathcal{J}(J_f) = J_m$), especially for the access restricted JNI methods and JNI functions. Then, *Module-D* queries $\mathcal{P}$ to retrieve the privilege checks on the pair of JNI method ($\mathcal{P}(J_m)$) and JNI function ($\mathcal{P}(J_f)$) and contrasts their strictness to discover inconsistencies. If $\mathcal{P}(J_m) > \mathcal{P}(J_f)$ (i.e., the access control enforced on J_f is less restrictive than that on J_m), a Type-1 inconsistency is discovered. If $\mathcal{P}(J_m) < \mathcal{P}(J_f)$ (i.e., the access control enforced on J_m is less restrictive than that on J_f), a Type-2 inconsistency is uncovered.

For instance, since the check on the *shell* privilege associated with the JNI function `android::nativeCreateDisplay` is less restrictive than the check on the *system* privilege correlated to the JNI method `SurfaceControl.nativeCreateDisplay`, a Type-1 cross-context inconsistency is found.

VIII. Evaluation

We implement `IAceFinder` in around $3k$ SLOC Java (for *Module-J*) and $2k$ SLOC Python (for *Module-N* and *Module-D*). We evaluate it by answering three research questions (RQs).

RQ1: Can `IAceFinder` precisely associate JNI methods and JNI functions with their access control enforcement?

RQ2: Can `IAceFinder` discover the inconsistent access control enforcement in the official Android systems?

RQ3: Can `IAceFinder` discover the inconsistent access control enforcement in open-source third-party Android ROMs?

Data Set: To answer the research questions, we use `IAceFinder` to analyze 14 open-source Android distributions. Table III lists the details about the Android distributions under evaluation, where `Name`, `Version`, and `Description` provide the name, Android version, and additional information about each Android distribution. In addition, `#Binder`$_j$ and `#Binder`$_n$ provide the number of Binder proxy (or Binder stub) in the Java context and native context of Android, respectively. In detail, AOSP [5] is the official Android system provided by Google. LineageOS [24], OmniROM [30], CAF [3], GrapheneOS [20], BlissROM [11], PixelExperience [35], DirtyUnicorns [17], and NitrogenOS [29] are variants of AOSP with customization. Since LineageOS (previously known as CyanogenMod [16]) and CAF are two of popular open-source third-party Android distributions, several Android ROMs are implemented based on them, including Replicant [37], crDroid [13], EvolutionX [19], and MoKee [28]. We downloaded and compiled these ROMs between October, 2020 and January, 2021. During compilation, we use `WLLVM` [42] to link the LLVM bitcode of each object

(.obj) file of a native library to a single bitcode file so that IAceFinder can get the LLVM bitcode of each native library.

A. Precision of Correlating Access Control Enforcement to JNI Methods and JNI Functions (RQ1)

To assess the precision of IAceFinder in correlating access control to their corresponding JNI methods and JNI functions, we manually examine the results of IAceFinder on analyzing the system services of 14 open-source Android ROMs.

Results: Table III lists the details about the access restricted JNI methods and JNI functions recognized by IAceFinder, where $\#\mathtt{J}_m$ and $\#\mathtt{J}_f$ provide the number of identified access restricted JNI methods and JNI functions, and $\mathtt{Precis}_m$ and $\mathtt{Precis}_f$ provide the precision of the identified access restricted JNI methods and JNI functions, respectively. Table III also lists the results showing the effectiveness of our approaches to filter out false positives in identifying access restricted JNI methods (see §V-D). $\#\mathtt{J}'_m$ and $\Delta\#\mathtt{J}_m$ denote the number of identified access restricted JNI methods without filtering and the number of false positives reduced by our approaches, respectively. $\mathtt{Precis}'_m$ and $\Delta\mathtt{Precis}_f$ denote the precision of the identified access restricted JNI methods without filtering and the improvement in precision resulted from our approaches, respectively.

More specifically, our approaches effectively filter out 13 false-positive cases in identifying the access restricted JNI methods and increase the precise from around 74.1% to 89.5%. After manually inspecting the remaining wrongly identified access restricted JNI methods, we find that, although they are control dependent on access control statements, they are not the targets the access control intends to protect. The remaining false positives that are not filtered out by our approaches are because all reachable paths from remote interfaces of Java system services to them enforce access control. For instance, we fail to filter out the wrongly identified access restricted JNI method nativeReloadCalibration because there is only one reachable path to it from the Input Manager service's remote interface setTouchCalibrationForInputDevice [39], which enforces the access control on the Java method setTouchCalibration rather than the JNI method.

The average precision of the identified access-restricted JNI functions is around 94.0%. After manually analyzing the four false positives, we find that the enforced access control is not to restrict the execution of JNI functions but to make the functions execute properly. For example, the wrongly identified access restricted JNI function MediaPlayer_set_audio_session_id [38] enforces UID check in the interface releaseAudioSessionId of the AudioFlinger service to adjust the value of the interface's local variable to a correct one.

Answer to RQ1: IAceFinder can precisely correlate the access control enforcement to their corresponding JNI methods and JNI functions with the precision of 89.5% and 94.0%.

B. Cross-Context Inconsistent Access Control Enforcement in Recently Released Official Android Systems (RQ2)

To uncover inconsistencies in the official Android systems, we apply IAceFinder to two recently forked branches of AOSP [5], including branches 10.0.0_r41 and 11.0.0_r21.

Results: Table III also lists the statistics about the discovered inconsistencies, where #Inconsistency provides the number of uncovered inconsistent access control enforcement.

IAceFinder discovers 18 and 22 cross-context inconsistent access control enforcement in Android 10 and Android 11 of AOSP, respectively. Table IV presents details about them, where Type indicates the type of each inconsistency, Interface shows the remote interfaces of Java system services (*deputy*) and native system services (*target*) involved in the inconsistencies, Access Control lists the permission checks and UID checks enforced on the interfaces, and Privilege provides the necessary privilege to execute the interfaces. For example, the 3^{rd} case in Table IV lists the details about the example of Type-1 inconsistency that has been introduced in §III-D.

In total, IAceFinder uncovers 22 cross-context inconsistencies in the two official Android systems (i.e., the top 22 cases presented in Table IV), 18 of them (i.e., the top 18 cases) are found in both Android systems, and the remaining 4 cases are uniquely found in Android 11. Meanwhile, among the 22 cases, 18 of them are Type-1 inconsistencies, and 4 of them are Type-2 inconsistencies. We observe that, since numerous interfaces of Audio service (a Java system service for managing audio devices, audio streams, audio policies, and audio settings) rely on native system services (e.g., AudioPolicy service and AudioFlinger service) to accomplish their tasks, Audio service is involved in many of the discovered inconsistencies. We have reported the discovered inconsistencies to Google and got 9,500 USD rewards from Google vulnerability reward program.

Adversaries can exploit the cross-context inconsistent access control enforcement to compromise the device and violate user privacy. In Table V, we present the potential exploitability of each discovered inconsistencies listed in Table IV. For example, according to the 3^{rd} case in Table V, the attack application can abuse the inconsistency to create a secure virtual display to steal the user's password, violating user privacy. Meanwhile, according to the 9^{th} case in Table V, the attack application can exploit the inconsistency to increase or decrease the volume of audio steams, compromising the usability of the device.

Answer to RQ2: IAceFinder discovers 18 and 22 inconsistent cross-context access control enforcement in 2 recently released official Android systems, respectively.

C. Cross-Context Inconsistent Access Control Enforcement in Open-Source Third-Party Android ROMs (RQ3)

To find inconsistencies in third-party Android distributions, we apply IAceFinder to 12 open-source Android ROMs, whose source code can be successfully compiled by us.

Results: Table III, Table IV, and Table V also list the statistics, details, and exploitability of the cross-context inconsistencies discovered in the third-party Android ROMs.

Specifically, comparing $\#\mathtt{Binder}_j$, $\#\mathtt{Binder}_n$, $\#\mathtt{J}_m$ and $\#\mathtt{J}_f$ of the open-source third-party Android ROMs with those of the official Android systems, we notice that the open-source Android ROMs rarely customize system services. Hence, the numbers of the identified inconsistencies are almost the same as those found in the official Android systems. Nonetheless, we discover an extra inconsistency in LineageOS based Android

TABLE IV: Details about the discovered cross-context inconsistent access control enforcement.

	Type	Interface[1](*Deputy*)	Access Control[2]	Privilege	Interface[1](*Target*)	Access Control[1]	Privilege
1	Type-1	AS.isAudioServerRunning	MODIFY_PHONE_STATE	*shell*	SM::checkService	N/A	∅
2	Type-1	CDS.getTransformCapabilities	CONTROL_DISPLAY_COLOR_TRANSFORMS	*system*	SF::getProtectedContentSupport	N/A	∅
3	Type-1	DMS.createVirtualDisplay	CAPTURE_SECURE_VIEDO_OUTPUT	*system*	SF::createDisplay	ACCESS_SURFACE_FLINGER	*shell*
4	Type-1	UAC.getWindowAnimationFrameStats	SYSTEM_UID	*system*	SF::getAnimationFrameStats	ACCESS_SURFACE_FLINGER	*shell*
5	Type-1	UAC.clearWindowAnimationFrameStats	SYSTEM_UID	*system*	SF::clearAnimationFrameStats	ACCESS_SURFACE_FLINGER	*shell*
6	Type-1	AS.getVolumeIndexForAttributes	MODIFY_AUDIO_ROUTING	*system*	APS::listAudioProductStrategies	N/A	∅
7	Type-1	AS.getMaxVolumeIndexForAttributes	MODIFY_AUDIO_ROUTING	*system*	APS::getMaxVolumeIndexForAttributes	N/A	∅
8	Type-1	AS.getMinVolumeIndexForAttributes	MODIFY_AUDIO_ROUTING	*system*	APS::getMinVolumeIndexForAttributes	N/A	∅
9	Type-1	AS.setVolumeIndexForAttributes	MODIFY_AUDIO_ROUTING	*system*	APS::setVolumeIndexForAttributes	MODIFY_AUDIO_SETTINGS	*normal*
10	Type-1	AS.getVolumeIndexForAttributes	MODIFY_AUDIO_ROUTING	*system*	APS::getVolumeIndexForAttributes	N/A	∅
11	Type-1	AS.getAudioVolumeGroups	MODIFY_AUDIO_ROUTING	*system*	APS::listAudioVolumeGroups	N/A	∅
12	Type-1	AS.setMasterMute	MODIFY_AUDIO_ROUTING	*system*	AF::setMasterMute	AID_APP_START	*shell*
13	Type-1	DMS.requestColorMode	CONFIGURE_DISPLAY_COLOR_MODE	*system*	SF::setActiveColorMode	ACCESS_SURFACE_FLINGER	*shell*
14	Type-1	CS.notifyCameraState	CAMERASERVER_UID	*system*	AF::setParameters	MODIFY_AUDIO_SETTINGS	*normal*
15	Type-2	DMS.releaseVirtualDisplay	N/A	∅	SF::destroyDisplay	ACCESS_SURFACE_FLINGER	*shell*
16	Type-2	AS.setRingerMode	N/A	∅	APS::setForceUse	MODIFY_AUDIO_ROUTING	*system*
17	Type-2	AS.setMode	MODIFY_AUDIO_SETTINGS	*normal*	APS::setPhoneState	AID_APP_START	*shell*
18	Type-2	AS.setMicrophoneMute	MODIFY_AUDIO_SETTINGS	*normal*	AF::setMicMute	AID_APP_START	*shell*
19	Type-1	AS.getDevicesForAttributes	MODIFY_AUDIO_ROUTING	*system*	APS::getDevicesForAttributes	AID_APP_START	*shell*
20	Type-1	AS.setPreferredDeviceForStrategy	MODIFY_AUDIO_ROUTING	*system*	APS::setPreferredDeviceForStrategy	AID_APP_START	*shell*
21	Type-1	AS.getPreferredDeviceForStrategy	MODIFY_AUDIO_ROUTING	*system*	APS::getPreferredDeviceForStrategy	AID_APP_START	*shell*
22	Type-1	AS.removePreferredDeviceForStrategy	MODIFY_AUDIO_ROUTING	*system*	APS::removePreferredDeviceForStrategy	AID_APP_START	*shell*
23	Type-1	LAS.listAudioSessions	DUMP	*shell*	APS::listAudioSessions	N/A	∅

[1] APS:AudioPolicyService, AS:AudioService, CDS:ColorDisplayService, CS:CameraServiceProxy, DMS:DisplayManagerService, LAS:LineageAudioService, SF:SurfaceFlinger, SM:ServiceManager, UAC:UiAutomationConnection.
[2] Due to space limitation, we just list the access control that can represent the privilege check enforced on remote interfaces of system services.

TABLE V: Potential exploitability of the discovered cross-context inconsistent access control enforcement.

	Type	Potential Exploitability
1	Type-1	Missing permission check on *deputy* ServiceManager::checkService lets third-party apps obtain the runtime status of AudioFlinger.
2	Type-1	Missing permission check on *deputy* SurfaceFlinger::getProtectedContentSupport lets third-party apps know whether protected content is supported in GPU composition.
3	Type-1	Less restrictive permission check on *deputy* SurfaceFlinger::createDisplay lets applications ran by ADB take screenshots of secured windows.
4	Type-1	Less restrictive UID check on *deputy* SurfaceFlinger::getAnimationFrameStats lets applications ran by ADB get the frame statistics of animations.
5	Type-1	Less restrictive UID check on *deupty* SurfaceFlinger::clearAnimationFrameStats lets applications ran by ADB clear the frame statistics of animations.
6	Type-1	Missing permission check on *deputy* AudioPolicyService::listAudioProductStrategies lets third-party apps retrieve audio product strategies of system.
7	Type-1	Missing permission check on *deputy* AudioPolicyService::getMaxVolumeIndexForAttributes lets third-party apps get the maximum volume index of an audio stream.
8	Type-1	Missing permission check on *deputy* AudioPolicyService::getMinVolumeIndexForAttributes lets third-party apps get the minimum volume index of an audio stream.
9	Type-1	Less restrictive permission check on *deputy* AudioPolicyService::setVolumeIndexForAttributes lets third-party apps adjust the volume index of an audio stream.
10	Type-1	Missing permission check on *deputy* AudioPolicyService::getVolumeIndexForAttributes lets third-party apps get the current volume index of an audio stream.
11	Type-1	Missing permission check on *deputy* AudioPolicyService::listAudioVolumeGroups lets third-party apps retrieve audio volume groups of system.
12	Type-1	Less restrictive access control on *deputy* AudioFlinger::setMasterMute lets applications ran by ADB silence the global sound of system.
13	Type-1	Less restrictive permission check on *deputy* SurfaceFlinger::setActiveColorMode lets applications ran by ADB configure the color mode of a display.
14	Type-1	Less restrictive access control on *deputy* AudioFlinger::setParameters lets third-party apps set the audio parameters related to camera.
15	Type-2	Missing permission check on *target* SurfaceFlinger::destroyDisplay lets third-party apps destroy the virtual display.
16	Type-2	Missing permission check on *target* AudioPolicyService::setForceUse lets third-party apps select the audio device for specific usages.
17	Type-2	Less restrictive access control on *target* AudioPolicyService::setPhoneState lets third-party apps change the phone state of system.
18	Type-2	Less restrictive access control on *target* AudioFlinger::setMicMute lets third-party apps silence the microphone of device.
19	Type-1	Less restrictive access control on *deputy* AudioPolicyService::getDevicesForAttributes lets applications ran by ADB get the audio devices used for an audio attribute.
20	Type-1	Less restrictive access control on *deputy* AudioPolicyService::setPreferredDeviceForStrategy lets applications ran by ADB set preferred audio device of an audio stream.
21	Type-1	Less restrictive access control on *deputy* AudioPolicyService::getPreferredDeviceForStrategy lets applications ran by ADB get preferred audio device of an audio stream.
22	Type-1	Less restrictive access control on *deputy* AudioPolicyService::removePreferredDeviceForStrategy lets applications ran by ADB remove preferred audio device of an audio stream.
23	Type-1	Missing permission check on *deputy* AudioPolicyService::listAudioSessions lets third-party apps retrieve the information about current audio sessions.

ROMs (the 23^{rd} case in Table IV). The inconsistency can result in an information disclosure (see the 23^{rd} case in Table V). We have reported the found inconsistency to the maintainers of LineageOS, who confirmed and patched this inconsistency.

Answer to RQ3: `IAceFinder` identifies an extra case of the cross-context inconsistent access control enforcement from the 12 open-source Android ROMs under analysis.

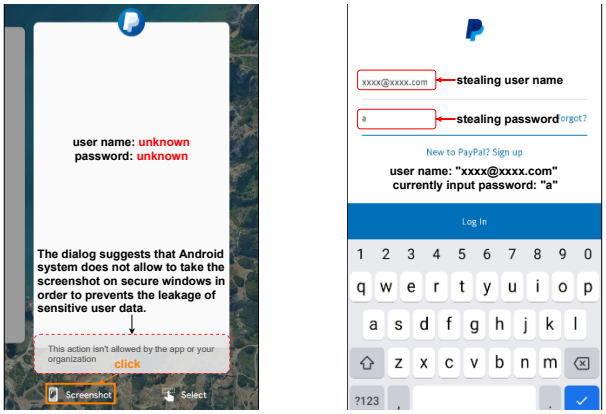
D. Case Study

In this section, we present the case studies of the motivating examples that have been introduced in §III-D.

• **Example of Type-1 Inconsistency**: According to the 3^{rd} case listed in Table IV, the Display Manager service's interface `createVirtualDisplay` (*deputy*) and the SurfaceFlinger service's interface `createDisplay` (*target*) enforce inconsistent access control. More specifically, the *deputy* allows the applications, having gained the `CAPTURE_SECURE_VIDEO_OUTPUT` permission (*system* privilege), to create the secure virtual display, whereas the *target* allows the applications, having gained the `ACCESS_SURFACE_FLINGER` permission (*shell* privilege) to create the secure virtual display.

▷ **Exploitation**: Since the access control enforced in the *target* is less restrictive than that enforced in the *deputy*, the attack application can call the *target* to evade the stricter access

control enforced in the *deputy*. In practice, attackers can create a native program to call `createDisplay` to create the secure virtual display. By using the techniques of malware [14, 31], the native program can be pushed to the victim's device and then be launched by the remotely connected ADB. Since the native program shares the same UID with ADB (i.e., SHELL_UID), it has the `ACCESS_SURFACE_FLINGER` permission. Note that, the native program does not have the `CAPTURE_SECURE_VIDEO_OUTPUT` permission since this permission has not been granted to ADB.



(a) Disallow screenshot. (b) Leak sensitive data.

Fig. 11: Security impact of the example of Type-1 inconsistency.

▷ **Security Impact**: The secure virtual display created by the attack application can be misused to steal the sensitive user information (i.e., user name and password) rendered on the device screen. Specifically, online payment apps or mobile banking apps commonly render sensitive user data in the secure window to prevent the leakage of such sensitive information [53]. By design, the visual content of the secure window should not be included in the screenshot (as shown in Fig.11a). However, by leveraging the secure virtual display, adversaries can take a screenshot (i.e., obtaining the visual content) of the secure window, and thus they can steal the user's sensitive data rendered on the secure window.

For example, Fig.11b demonstrates that, by leveraging the secure virtual display, attackers can obtain the visual content of the secure window. More specifically, the attacker can obtain the user name ("xxxx@xxxx.com") and the currently input password ("a") from the screenshot of PayPal's login window (i.e., the secure window under attack). This case study shows that the cross-context inconsistent access control enforcement can be abused to invade user privacy.

• **Example of Type-2 Inconsistency**: For the 16^{th} case listed in Table IV, the Audio service's interface `setRingerModeExternal` (*deputy*) and the AudioPolicy service's interface `setForceUse` (*target*) enforce inconsistent access control. More specifically, the *deputy* enforces no access control and thus allows any applications to set the audio routing for the vibrate ringtone, whereas the *target* allows the applications, having been granted with the `MODIFY_AUDIO_ROUTING` permission (*system* privilege) to set the audio routing for the vibrate ringtone.

▷ **Exploitation**: Since the access control enforced in the *deputy* is less restrictive than that enforced in the *target*, the attack application can call the *deputy* to access the *target* as if it was the privileged system service. In practice, attackers can create a normal app to call `setRingerMode` in order to control the audio routing for the vibrate ringtone.

▷ **Security Impact**: The attack application can set no audio routing for the vibrate ringtone (i.e., silence the vibrate ringtone) as if the device was on the silence mode. By default, Android system sets the Bluetooth SCO channel (e.g., Bluetooth headset) as the audio routing for the vibrate ringtone. Accordingly, if the user wears a Bluetooth headset, he can hear the vibrate ringtone when there is a phone call. However, the attack application can set no audio routing for the vibrate ringtone to make the user unable to hear the ringtone from the Bluetooth headset, letting him miss the phone call. This case study shows that the cross-context inconsistent access control enforcement can be abused to compromise the usability of the device.

IX. Limitations of IAceFinder

`IAceFinder` has two main limitations.

First, `IAceFinder` just considers the permission based and UID based access control because we have conducted a manual inspection on C++ code of more than 10 native system services and observed that their remote interfaces mainly enforce these two kinds of access control. In future work, we will adapt and employ the text analysis proposed by ACMiner [61] to find all potential access control enforced in native system services, and then extend `IAceFinder` to detect cross-context inconsistencies in the newly discovered access control enforcement.

Second, `IAceFinder` can just identify cross-context inconsistencies in open-source Android ROMs because we modify their source code to distinguish C++ objects in native code (e.g., objects of interface classes introduced in §VI-A). Since `IAceFinder` will be open source after paper publication, the developers of closed source Android ROMs can use it to discover cross-context inconsistencies in their ROMs. To detect cross-context inconsistencies in closed source ROMs, we will adopt a hybrid approach to inspect such ROMs in future work. For example, we could conduct binary analysis to identify the access control enforcement whenever possible. We could also adopt fuzzing techniques to dynamically execute each framework API and instrument the framework to record and compare the access control enforced in different contexts of Android.

X. Related Work

We present the related work on finding inconsistent access control enforcement in Android (§X-A), detecting confused deputy (or capability leak) problems on Android (§X-B), and inspecting access control in traditional operating systems (§X-C).

A. Finding Inconsistent Access Control in Android

There are a number of studies on finding inconsistent access control enforcement in Android. To the best of our knowledge, they all focus on finding inconsistencies in Java system services. Kratos [75] builds the context-insensitive callgraph of Java system services and discovers inconsistent permission checks and UID checks based on the callgraph. AceDroid [43] identifies inconsistencies by comparing Java system services' access control enforcement of diverse vendor customization. It models the access control in a path-sensitive manner and normalizes

diverse authority inspections to a canonical form. ACMiner [61] combines static code analysis and text analysis to generate a set of authority inspections and uses association rule mining to detect inconsistent access control enforcement

`IAceFinder` is different from the existing studies because it uncovers the cross-context inconsistent access control enforcement in Android whereas all of the previous work just finds inconsistencies in the Java context of Android. Existing studies cannot be used to identify cross-context inconsistencies because of three reasons. First, they do not analyze the access control enforced on JNI methods in Java context of Android, and thus they cannot identify the access restricted JNI methods for detecting cross-context inconsistencies. Second, they do not analyze native system services. Therefore, they cannot find the access control enforced in native system services and the access control enforced on JNI functions, which are used to identify cross-context inconsistencies. Third, they do not analyze the JNI interfaces (i.e., pairs of JNI methods and JNI functions) that bridge Java and native context of Android, and thus they cannot find the cross-context inconsistent access control enforcement.

B. Detecting Confused Deputy Problems on Android

Confused deputy (or capability leak) problems on Android have been widely investigated, and most of the existing studies focus on detecting and preventing such problems on Android apps. To detect confused deputy (or capability leak) problems on apps, existing studies [51, 54, 60, 63, 66, 78, 79] use static code analysis to find whether the apps' sensitive functionalities can be accessed by other apps. If so, the apps suffer from the problems. To prevent such problems, previous work [49, 50, 55, 60, 71] modifies Android framework to monitor inter-component communication among apps at runtime. To our best knowledge, ARF [62] is the only work that analyzes confused deputy problems in Android system, and it uses ACMiner [61] to find the problems on Java system services.

It is worth noting that, existing approaches for detecting confused deputy (or capability leak) problems on Android apps can not be used to identify Type-2 inconsistency, because it is a new instance of confused deputy problems where the deputy is a system service rather than an app. Moreover, Type-2 inconsistency is different from the problem uncovered by ARF, because it involves both Java and native system services rather than only Java system services. Therefore, ARF cannot be used to identify Type-2 inconsistency.

C. Inspecting Access Control in Traditional Operating Systems

Besides Android, researchers have conducted many studies on inspecting access control in traditional operating systems, especially Linux. Existing work [56–58, 65, 77, 81, 82] adopts static code analysis or system event monitoring to identify inconsistent Linux Security Modules (LSM) authorization and capability permission check in Linux kernel.

Since the access control enforced in Android system services is different from that in Linux kernel, the previous approaches cannot be used to identify such inconsistencies in Android.

XI. Conclusion

We conduct the *first* systematic investigation on the cross-context inconsistent access control enforcement in Android. To automatically detect such inconsistencies, we design and develop IAceFinder, a novel tool that performs static analysis on both contexts of Android after tackling several technical challenges. Applying IAceFinder to 14 open-source Android distributions, we find that it can effectively discover the cross-context inconsistent access control enforcement. In particular, it uncovers 23 cross-context inconsistencies, which can be abused by attackers to compromise the device and invade user privacy.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This research is partially supported by the Hong Kong RGC Project (No. PolyU15223918), National Natural Science Foundation of China (No. 62072046, No. 61872057, No. 61872438), National Key R&D Program of China (No. 2018YFB0804100), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform ZJUNGICS2021016, K20200019), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005), and the National Science Foundation under Grant (No. 1951729, 1953813, and 1953893).

References

[1] "ADB shell," https://cs.android.com/android/platform/superproject/+/master:frameworks/base/packages/Shell/, 2021.
[2] "Android Debug Bridge (adb)," https://developer.android.com/studio/command-line/adb, 2021.
[3] "Android for MSM," https://www.codeaurora.org/projects/android-for-msm, 2021.
[4] "Android Interface Definition Language (AIDL)," https://developer.android.com/guide/components/aidl, 2021.
[5] "Android Open Source Project," https://source.android.com/, 2021.
[6] "android.content.res," https://developer.android.com/reference/android/content/res/package-summary, 2021.
[7] "android.graphics," https://developer.android.com/reference/android/graphics/package-summary, 2021.
[8] "android:protectionLevel," https://developer.android.com/guide/topics/manifest/permission-element, 2021.
[9] "android.util," https://developer.android.com/reference/android/util/package-summary, 2021.
[10] "Binder," https://developer.android.com/reference/android/os/Binder, 2021.
[11] "Bliss ROMs," https://blissroms.com/, 2021.
[12] "checkSelfPermission," https://developer.android.com/reference/android/content/Context#checkCallingPermission(java.lang.String), 2021.
[13] "crDroid Android," https://crdroid.net/, 2021.
[14] "Cryptocurrency-Mining Botnet Spreads via ADB, SSH," https://www.trendmicro.com/en_us/research/19/f/cryptocurrency-mining-botnet-arrives-through-adb-and-spreads-through-ssh.html, 2021.
[15] "CVE-2020-27057," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-27057, 2021.
[16] "CyanogenMod," https://github.com/CyanogenMod, 2021.
[17] "Dirty Unicorns," https://dirtyunicorns.com/, 2021.
[18] "DisplayManager," https://developer.android.com/reference/android/hardware/display/DisplayManager, 2021.
[19] "Evolution X," https://evolution-x.org/, 2021.
[20] "GrapheneOS," https://grapheneos.org/, 2021.

[21] "Java Native Interface Programming," http://journals.ecs.soton.ac.uk/java/tutorial/native1.1/implementing/index.html, 2021.
[22] "Java Native Interface Specification," https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html, 2021.
[23] "JNI tips," https://developer.android.com/training/articles/perf-jni, 2021.
[24] "LineageOS Android Distribution," https://lineageos.org/, 2021.
[25] "LLVM bitcode linker," http://llvm.org/docs/CommandGuide/llvm-link.html, 2021.
[26] "LLVM's object file dumper," https://llvm.org/docs/CommandGuide/llvm-objdump.html, 2021.
[27] "Mapping High Level Constructs to LLVM IR," https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/README.html, 2021.
[28] "MoKee ROM," https://www.mokeedev.com/, 2021.
[29] "Nitrogen OS," https://github.com/nitrogen-project, 2021.
[30] "OmniROM," https://omnirom.org/, 2021.
[31] "Open ADB Ports Used to Spread Possible Satori Variant," https://www.trendmicro.com/en_us/research/18/g/open-adb-ports-being-exploited-to-spread-possible-satori-variant-in-android-devices.html, 2021.
[32] "Permissions defined in Android framework," https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/res/AndroidManifest.xml, 2021.
[33] "Permissions for ADB shell," https://cs.android.com/android/platform/superproject/+/master:frameworks/base/packages/Shell/AndroidManifest.xml, 2021.
[34] "Permissions overview," https://developer.android.com/guide/topics/permissions/overview, 2021.
[35] "Pixel Experience," https://download.pixelexperience.org/, 2021.
[36] "Platform Architecture," https://developer.android.com/guide/platform, 2021.
[37] "Replicant," https://replicant.us/, 2021.
[38] "set_audio_session_id," https://cs.android.com/android/platform/superproject/+/master:frameworks/base/media/jni/android_media_MediaPlayer.cpp;drc=master;l=991, 2021.
[39] "setTouchCalibrationForInputDevice," https://cs.android.com/android/platform/superproject/+/master:frameworks/base/services/core/java/com/android/server/input/InputManagerService.java;l=968, 2021.
[40] "Soot," https://github.com/soot-oss/soot, 2021.
[41] "SurfaceFlinger," https://source.android.com/devices/graphics/surfaceflinger-windowmanager, 2021.
[42] "Whole Program LLVM," https://github.com/travitch/whole-program-llvm, 2021.
[43] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection," in *Proc. NDSS*, 2018.
[44] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li, "Precise Android API Protection Mapping Derivation and Reasoning," in *Proc. CCS*, 2018.
[45] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, p. 9, 1986.
[46] K. Au, Y. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proc. CCS*, 2012.
[47] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis," in *Proc. USENIX Security*, 2016.
[48] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A Methodology for Empirical Analysis of Permission-Based Security Models and Its Application to Android," in *Proc. CCS*, 2010.
[49] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Technical Report TR-2011-04*, 2011.
[50] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards Taming Privilege-Escalation Attacks on Android," in *Proc. NDSS*, 2012.
[51] P. P. Chan, L. C. Hui, and S. M. Yiu, "DroidChecker: Analyzing Android Applications for Capability Leak," in *Proc. WISEC*, 2012.
[52] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks," in *Proc. USENIX Security*, 2014.
[53] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *Proc. ICSE*, 2020.
[54] B. F. Demissie, M. Ceccato, and L. K. Shar, "Security analysis of permission re-delegation vulnerabilities in Android apps," *Empir Software Eng*, vol. 25, p. 5084–5136, 2020.
[55] M. Dietz, "QUIRE: Lightweight Provenance for Smart Phone Operating Systems," in *Proc. USENIX Security*, 2011.
[56] A. Edwards, T. Jaeger, and X. Zhang, "Maintaining the correctness of the Linux security modules framework," in *Ottawa Linux Symposium*, 2002.
[57] A. Edwards, T. Jaeger, and X. Zhang, "Runtime verification of authorization hook placement for the Linux security modules framework," in *Proc. CCS*, 2002.
[58] D. Efremov and I. Shchepetkov, "Runtime Verification of Linux Kernel Security Module," in *Proc. FM*, 2019.
[59] A. Einarsson and J. D. Nielsen, "A survivor's guide to Java program analysis with soot," *BRICS, Department of Computer Science, University of Aarhus, Denmark*, vol. 17, 2008.
[60] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *Proc. USENIX Security*, 2011.
[61] S. A. Gorski, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, and A. Bartel, "ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware," in *Proc. CODASPY*, 2019.
[62] S. A. Gorski and W. Enck, "ARF: Identifying Re-Delegation Vulnerabilities in Android System Services," in *Proc. WiSec*, 2019.
[63] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proc. NDSS*, 2012.
[64] A. Grünbacher, "POSIX Access Control Lists on Linux," in *Proc. USENIX ATC*, 2003.
[65] T. Jaeger, A. Edwards, and X. Zhang, "Consistency analysis of authorization hook placement in the Linux security modules framework," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, pp.

175–205, 2004.
[66] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A SEALANT for Inter-App Security Holes in Android," in *Proc. ICSE*, 2017.
[67] O. Lhoták and L. Hendren, "Context-Sensitive Points-to Analysis: Is It Worth It?" in *Proc. CC*, 2006.
[68] O. Lhoták and L. Hendren, "Scaling Java Points-to Analysis Using SPARK," in *Proc. CC*, 2003.
[69] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing Android Native System Services via Automated Interface Analysis," in *Proc. USENIX Security*, 2020.
[70] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "System Service Call-Oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation," in *Proc. MobiSys*, 2017.
[71] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proc. USENIX Security*, 2013.
[72] C. Qian, X. Luo, Y. Shao, and A. Chan, "On Tracking Information Flows through JNI in Android Applications," in *Proc. DSN*, 2014.
[73] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, pp. 40–48, 1994.
[74] Y. Shao, X. Luo, and C. Qian, "RootGuard: Protecting Rooted Android Phones," *IEEE Computer*, vol. 47, 2014.
[75] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao, "Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework," in *Proc. NDSS*, 2016.
[76] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proc. CC*, 2016.
[77] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically Inferring Security Specification and Detecting Violations," in *Proc. USENIX Security*, 2008.
[78] J. Wu, T. Cui, T. Ban, S. Guo, and L. Cui, "PaddyFrog: Systematically Detecting Confused Deputy Vulnerability in Android Applications," *Sec. and Commun. Netw.*, vol. 8, p. 2338–2349, 2015.
[79] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The Impact of Vendor Customizations on Android Security," in *Proc. CCS*, 2013.
[80] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. Chan, "NDroid: Towards Tracking Information Flows Across Multiple Android Contexts," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 14, p. 814–828, 2019.
[81] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "PeX: A Permission Check Analysis Framework for Linux Kernel," in *Proc. USENIX Security*, 2019.
[82] X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for Static Analysis of Authorization Hook Placement," in *Proc. USENIX Security*, 2002.
[83] H. Zhou, H. Wang, S. Wu, X. Luo, Y. Zhou, T. Chen, and T. Wang, "Finding the Missing Piece: Permission Specification Analysis for Android NDK," in *Proc. ASE*, 2021.
[84] H. Zhou, H. Wang, Y. Zhou, X. Luo, Y. Tang, L. Xue, and T. Wang, "Demystifying Diehard Android Apps," in *Proc. ASE*, 2020.
[85] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *Proc. S&P*, 2014.

APPENDIX

A. Building Callgraph for Java System Services

Since it is error-prone to build the callgraph for the entire Java context because of the huge code base [43, 46, 75], we choose to build the callgraph for each Java system service. Our approach will not affect the soundness of the inconsistency analysis as we let the callgraph include all potential interactions between Android applications and each Java system service.

Specifically, we take two steps to build the callgraph. First, to include all possible interactions between applications and the Java system service, we collect the local interfaces, which are provided by the Java Binder proxy for applications to call remote interfaces of the service. In addition, since `Soot` needs a single entrypoint (like the `main` method in Java programs) to build the callgraph [59], we carefully create one, which contains the invocations to all collected local interfaces. Note that, the previous work [67, 75] has pointed out that adopting context-sensitive points-to analysis to build the callgraph is time-consuming, and such a complex approach will not dramatically improve the accuracy of the callgraph. Therefore, we decide to use SPARK [68], a context-insensitive points-to analysis based callgraph generation algorithm supported by `Soot`, to build the callgraph for each Java system service.

Since the adopted imprecise callgraph generation algorithm will introduce incorrect callgraph edges, especially for those associated with IPC, in the second step, we remove them.

```
/* The class that declares the local interfaces */
01 interface IDisplayManager extends android.os.IInterface {
02     public int createVirtualDisplay(*); // local interface    }
/* The class for the Binder-proxy of Display Manager service */
03 class IDisplayManager$Stub$Proxy implements IDisplayManager {
04     private IBinder mRemote; // the real type is BinderProxy
05     public int createVirtualDisplay(*) {
06         boolean * = mRemote.transact(*); // IBinder.transact(*)    }  }
/* The class for the Binder-stub of Display Manager service */
07 class IDisplayManager$Stub extends Binder implements IDisplayManager {
08     public boolean onTransact(*) {
09         this.createVirtualDisplay(*); // "this" refers to DisplayManagerService  } }
/* The adjusted class for the Binder-stub of Display Manager service */
10 class DisplayManagerService$BinderService extends IDisplayManager$Stub {
11     public int createVirtualDisplay(*) { /* ignore the implementation */ }   }
```

Fig. 12: android/hardware/display/IDisplayManager.java.

• **Collecting Local Interfaces**: We find the class of each Java Binder proxy to collect the local interfaces. Java Binder proxy executes the method `transact` to send the request to the Binder stub of Java system services [10]. Accordingly, we treat the classes, implementing the methods that invoke `transact`, as the classes for the Binder proxy. For example, as shown in Fig.12, since `createVirtualDisplay` calls `transact` in Line 6, `IDisplayManager$Stub$Proxy` is class for the Binder proxy.

Once a class of the Java Binder proxy is found, to include the potential interactions between applications and the service in the callgraph, we create a method to invoke all this class's methods (i.e., local interfaces) that call `transact`. This method is treated as the entrypoint for `Soot` to build the callgraph.

It is worth noting that, when collecting local interfaces, we also find the class of each Java Binder stub, which is further used to collect remote interfaces for adjusting the callgraph. The class of the Binder proxy and the class of the corresponding Binder stub implement the same interface class [4], where interfaces of the service are declared. Accordingly, the class, which implements the same interface class as the class of the Binder proxy, is the class of the Binder stub. For example, in Fig.12, since `IDisplayManager$Stub$Proxy` (i.e., the class for the Binder proxy) in Line 3 and `IDisplayManager$Stub` in Line 7 both implement `IDisplayManager` (i.e., the interface class) in Line 1, `IDisplayManager$Stub` is the class of the Binder stub. Moreover, we observe that the class of the Binder stub may be extended by another class to implement the inherited interface methods. In this case, we treat the extending class as the class for the Binder stub. For instance, since `IDisplayManager$Stub` will be extended by `DisplayManagerService$BinderService` in Line 10, we treat the latter as the class of the Binder stub.

Once a class of the Java Binder stub is found, all this class's methods, which are invoked by the class's method `onTransact`, are the remote interfaces. We store them to the set I_J.

```
/* An implementation of the IBinder.transact method */
01 class Binder implements IBinder {
02    public final boolean transact(*) {
03       boolean r = onTransact(*); // implies IBinder.transact may call onTransact
04    } // other irrelevant code is ignored   }
```

Fig. 13: android/os/Binder.java.

• **Adjusting Callgraph**: The callgraph built by `Soot` contains incorrect edges resulted from the imprecise points-to analysis. For example, since SPARK cannot accurately determine the concrete type of the variable `mRemote` in Line 6 of Fig.12, for the consideration of soundness, SPARK will create the edges that connect `transact` to all its potential callees. Specifically, regarding the code snippet in Fig.13, since the method `transact` calls `onTransact` (in Line 3), every `onTransact` method is the potential callee of `transact`. SPARK connects `transact` invoked by the local interfaces to `onTransact` defined in the class of each Binder stub, introducing incorrect edges.

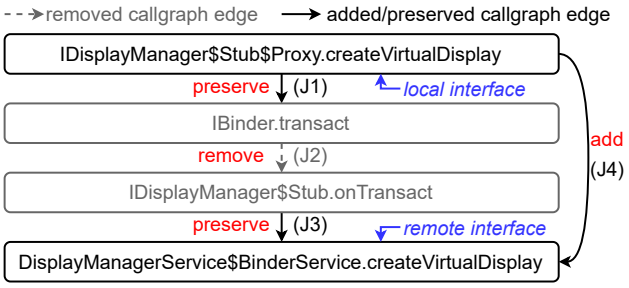


Fig. 14: An example of adjusting callgraph edges.

To remove the incorrect edges, we remove those that connect `transact` to `onTransact`. However, it will break the link between the Binder proxy and Binder stub, causing unreachability from local interfaces to remote interfaces. Fig.14 shows the original reachable path (J1↪J2↪J3) started from a local interface of the Binder proxy (the `IDisplayManager$Stub$Proxy` object) to the corresponding remote interface of the Binder stub (the `DisplayManagerService$BinderService` object). However, since J2 is the edge that connects `transact` to `onTransact`, it will be pruned. Thus, no path in the callgraph links the local interface to its corresponding remote interface. To rebuild the link, we connect local interfaces to their corresponding remote interfaces. For instance, we create the edge J4 to reconnect the two `createVirtualDisplay` methods.

B. Finding Access Control Statements in Java System Services

We mainly identify two types of access control statements, i.e., permission check statements and UID check statements.

• **Identifying Permission Check Statements**: Android provides a series of permission check methods for Java system services to examine whether the calling process of IPC has been granted with the required permission [12]. More specifically, the permission under check will be passed as a parameter to permission check methods and each permission is represented as a string constant in Android framework [75].

Based on this insight, we inspect every parameter value of each method invocation to identify the permission check statements. For example, as the invocation in Line 4 of Fig.6a, since the value of its first parameter is the string constant of the permission `CAPTURE_SECURE_VIDEO_OUTPUT`, the invocation of `checkCallingPermission` is identified as a permission check statement. Each time we identify a permission check statement S_p, we also record the permission p under check and store them in a statement-permission map $M_{sp} := \{S_p \mapsto p\}$, which is then used to determine the access control enforced on the access restricted JNI methods (see §V-D).

• **Identifying UID Check Statements**: Android takes two steps to enforce the UID check. First, it retrieves the UID of the calling process of IPC. To finish this task, as shown in Line 2 of Fig.6a, Java system services commonly call the method `getCallingUid` of `Binder` [75]. Second, the retrieved UID is compared with a constant representing a specific UID (see Table I). For example, as the `if` statement in Line 3 of Fig.6a, it checks whether the UID of calling process of IPC equals to SYSTEM_UID.

Based on this observation, we follow the two steps to find UID check statements. First, we find the methods that call `getCallingUid` from the callgraph. Second, we conduct def-use analysis [45] on the return value of `getCallingUid` to find whether it is used by an `if` statement. If so, a UID check statement is found. Each time we find a UID check statement S_u, we also record the value of UID u under check and store them in a statement-UID map $M_{su} := \{S_u \mapsto u\}$, which is then used to determine the access control enforced on the access restricted JNI methods (see §V-D).