Finding the Missing Piece: Permission Specification Analysis for Android NDK

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

***Abstract*—The Android research community has long focused on building the permission specification for Android framework APIs, which can be referenced by app developers to request the necessary permissions for their apps. However, existing studies just analyze the permission specification for Java framework APIs in Android SDK, whereas the permission specification for native framework APIs in Android NDK remains intact. Since more and more apps implement their functionalities using native framework APIs, and the permission specification for these APIs is poorly documented, the permission specification analysis for Android NDK is in urgent need. To fill in the gap, in this paper, we conduct the first permission specification analysis for Android NDK. In particular, to automatically generate the permission specification for Android NDK, we design and develop `PSGen`, a new tool that statically analyzes the implementation of Android framework and Android kernel to correlate native framework APIs with their required permissions. Applying `PSGen` to 3 Android systems, including Android 9.0, 10.0, and 11.0, we find that `PSGen` can precisely build the permission specification. With the help of `PSGen`, we discover more than 200 native framework APIs that are correlated with at least one permission.**

***Index Terms*—Android, Kernel, NDK, Permission**

I. Introduction

Android SDK [2] and NDK [6] provide Java framework APIs and native framework APIs, respectively, for apps to access the functionalities of Android framework and Android kernel. Java framework APIs are implemented in the framework's `.jar` files, which are used by apps to call the interfaces of Java system services [46]. Native framework APIs are developed in C or C++ code and are implemented in the system's `.so` libraries, which are employed by apps to invoke the interfaces of native system services [42] or access the kernel [29].

To prevent unauthorized apps from accessing the sensitive functionalities in Android framework [28], [32], [33], [46] and its kernel [58], Android adopts a permission based security model. For example, the system services of Android framework that implement sensitive functionalities enforce permission checks to examine whether the calling apps have gained the required permissions. If not, the "permission denied" exception will be thrown [38], which may cause apps to crash [50].

Therefore, in order to properly invoke the framework APIs whose execution leads to permission checks, apps must apply for the required permissions [19]. To do so, developers should know what permissions are required by the framework APIs used by their apps, and request them in the apps. Unfortunately, the correlations between framework APIs and their required permissions are not well documented in the API references published by Google [28], [32]. Therefore, it is in urgent need to build the permission specification for the framework APIs provided by Android SDK and NDK.

Various permission specification analysis has been proposed to build the permission specification for framework APIs [28], [32], [33], [40]. However, they solely focus on Java framework APIs and to our best knowledge, none of the existing work takes native framework APIs into analysis. Almanee et al. recently showed that native code are prevalent in top 200 free apps on Google Play [30]. Moreover, our analysis on more than 266K apps downloaded from a third-party app store reveals that native code is found in around 82% of these apps, indicating that a large number of apps have used native code to implement their functionalities. Since apps usually call native framework APIs in their native code [29], it is essential to build the permission specification for them. Note that the existing approaches for conducting permission specification analysis on framework APIs cannot be applied to Android NDK because these approaches can neither identify the permission checks in native code nor associate the permissions with their protected native framework APIs.

To fill in the gap, in this paper, we conduct the *first* permission specification analysis for Android NDK, and develop a Permission Specification Generator (`PSGen`) to automatically generate the permission specification for native framework APIs. More precisely, `PSGen` first finds the system library functions that enforce permission checks, and then identifies the native framework APIs that call these functions and correlates them with the permissions under check. The whole process consists of three steps. *First*, to find the permission checks enforced in native system services and the kernel, `PSGen` performs static analysis on system libraries of Android framework and the executable file of Android kernel to build callgraphs. *Second*, since native system services provide interfaces and the kernel offers system calls for system library functions to interact with them, `PSGen` analyzes the callgraphs to identify the permission

* The corresponding authors.

restricted interfaces and system calls, whose execution leads to permission checks. *Third*, given a native framework API, PSGen traverses the callgraphs to find the reachable paths between the API and those permission restricted interfaces and system calls, and associates the API with the permissions under check.

We address the following technical challenges in the development of PSGen. First, it is non-trivial to build the complete callgraph of a system library by analyzing it separately, because the functions in the system library may rely on the functions defined in others. To tackle this issue, while building the callgraph of a system library, we also analyze its dependent libraries (detailed in §V-A). Second, the polymorphism feature of C++ code in system libraries makes it hard to differentiate the objects of the classes that are inherited from the same parent class, which will degrade the callgraphs' accuracy. To mitigate this problem, we carefully perform points-to analysis on native code when constructing callgraphs (detailed in §V-A). Third, due to the huge and complex code base of the kernel, it is almost infeasible to directly apply points-to analysis to its executable file (i.e., vmlinux) for building the accurate callgraph. To approach this issue, since the kernel is composed by several independent modules, we perform static analysis on each of them to build their callgraphs, and then merge them to form the complete callgraph of the kernel (detailed in §VI-A).

We use PSGen to build the permission specification for native framework APIs of 3 Android systems, including Android 9.0 with the common kernel 4.4, Android 10.0 with the common kernel 4.4, and Android 11.0 with the common kernel 4.9. The results show that PSGen generates the permission specification for native framework APIs with the precision of over 92.7%. In addition, we discover over 200 native framework APIs that are correlated with at least one permissions.

In summary, we make the following contributions:

- To the best of our knowledge, we are the *first* to investigate the permission specification for Android NDK.
- We develop PSGen, a new tool to automatically build the permission specification for native framework APIs. We will release the tool after the paper gets published.
- We evaluate the performance of PSGen by applying it to 3 Android systems. The results show that PSGen can precisely generate the permission specification for native framework APIs provided by Android NDK. The generated permission specification for Android NDK and the source code of PSGen are available at https://github.com/moonZHH/PSGen.

II. Background

In this section, we provide the necessary knowledge about Android system services (in §II-A), Android kernel (in §II-B), Android NDK (in §II-C), and Android permissions (in §II-D).

A. Android System Services

System services are the essential parts of Android framework and they provide interfaces for apps to call their functions [21]. Depending on the programming languages used to implement their core functions, system services are categorized into the Java system services and the native system services [42].

Since apps and system services run in separate processes, Android provides Binder [7], an inter-process communication (IPC) mechanism, for apps to interact with services. Precisely, for each system service, apps use its Binder proxy to communicate with the Binder stub, which is commonly the service itself. More specifically, apps invoke local interfaces of the system service (i.e., the methods defined in the class of the Binder proxy) to call the service's remote interfaces (i.e., the methods defined in the class of the Binder stub). This process involves two steps. First, the local interface calls the transact function of the Binder proxy to send the request to the Binder stub. Second, the onTransact function of the Binder stub handles the request and calls the correlated remote interface.

It is worth noting that the classes of a system service's Binder proxy and Binder stub will inherit the same interface class [46]. In addition, each pair of the service's local interface and remote interface implements the same method declared in the interface class, and thus they share the same namespace, method name, return type, and parameter types. For instance, as shown in Figure 1, BpCameraService (in Line 5-7) and CameraService (in Line 8-10) are the classes for the Binder proxy and the Binder stub of the camera service, and both of them inherit the ICameraService interface class (in Line 1-4). Moreover, the connectDevice local interface (in Line 6) and the connectDevice remote interface (in Line 9) are a pair of interfaces, and both of them implement the connectDevice virtual method (in Line 3) declared in ICameraService.

```
// ICameraService is the interface class of the camera service.
01 class ICameraService { // ICameraService.h
02     virtual void addListener(*); // register listener for changes to camera status
03     virtual void connectDevice(*); // connect to camera device
04 /* ignore the declarations of other interfaces */ }
// BpCameraSerivce is the class for the Binder proxy of the camera service.
05 class BpCameraService : public ICameraService { // directly inherit
06     void android::hardware::<class>::connectDevice(*) { /* local interface */ }
07 /* ignore the implementations of other local interfaces */ }
// CameraSerivce is the class for the Binder stub of the camera service.
08 class CameraService : public ICameraService { // indirectly inherit
09     void android::hardware::<class>::connectDevice(*) { /* remote interface */ }
10 /* ignore the implementations of other remote interfaces */ }
```

Fig. 1: The C++ classes related to the camera service.

B. Android Kernel

Android kernel is a variant of Linux kernel [5]. Thus, it employs Discretionary Access Control [57] to restrict the access to critical resources based on the identity of subjects or the group to which they belong. For example, the kernel will check the process's user identifier (UID) or group identifier (GID) to decide whether it has the authority to access certain files [41].

The kernel provides system calls [24] for apps to access its functions. Each system call is assigned with a unique system call number and will be handled by a corresponding handler in the kernel. The correspondence between each system call and its handler is defined in the unistd.h file [25] of the kernel and we list partial of them in Table I.

Commonly, to invoke a system call, the syscall function [24] will be called with its first parameter specifying the system

call number. Moreover, it is worth noting that the standard C library (i.e., `libc.so`) defines the wrapper functions (e.g., `open`, `close`, `socket`) for several system calls [39], which use assembly code to call `syscall`.

TABLE I: Partial of system calls and their handlers.

System Call (Number)	Handler	Description
__NR_openat (56)	sys_openat	`open` in libc.so calls it to open a file.
__NR_close (57)	sys_close	`close` in libc.so calls it to close a file.
__NR_socket (198)	sys_socket	`socket` calls it to create a network socket.

C. Android NDK

App developers can use Android NDK [6], [44], [51], which consists of the app accessible system libraries (e.g., those listed in Table II), to implement part of an app or a whole app in native code. Commonly, apps can call native framework APIs provided by NDK libraries (i.e., the functions exported by the system libraries that make up Android NDK) to interact with system services or the kernel. For example, apps can call the `socket` function defined in `libc.so` to request the kernel to create the network socket.

TABLE II: Partial of the NDK libraries.

Library	Description
libc.so	Providing the standard C library APIs, such as `socket`.
libcamera2ndk.so	Providing the APIs to interact with the camera service.
libbinder_ndk.so	Providing the APIs to get the Binder proxy of system services.

D. Android Permissions

Android employs a permission based security model [34], [46] to prevent unauthorized apps from performing sensitive operations. For example, apps should apply for and then be granted with the required permissions in order to access private user data (e.g., contacts and SMSs), retrieve sensitive device information (e.g., microphone's states and input devices' states), or use critical system features (e.g., camera and internet) [19].

Depending on whether the granted permissions will give supplementary GIDs (i.e., Effective GIDs [11] which are commonly used for the privilege check) to the running processes of apps, we divide the permissions into two categories, namely, EGID related permissions and the other general permissions. EGID related permissions are declared in the `platform.xml` file [22] of Android system, part of which are listed in Table III. For example, the apps that have gained the `INTERNET` permission will be running with the AID_INET EGID.

III. Motivation

In §III-A, we introduce the native framework APIs, the execution of which is restricted by permissions. Then, in §III-B, we present how Android performs permission checks. Moreover, in §III-C, we show a motivating example to explain the need of building the permission specification for Android NDK.

TABLE III: Partial of EGID related permissions.

Permission	EGID (Value)
BLUETOOTH_ADMIN	AID_NET_BT_ADMIN (3001)
BLUETOOTH	AID_NET_BT (3002)
INTERNET	AID_INET (3003)
NET_ADMIN	AID_NET_ADMIN (3005)
READ_NETWORK_USAGE_HISTORY	AID_NET_BW_STATS (3006)
UPDATE_DEVICE_STATS	AID_NET_BW_ACCT (3007)

A. Types of native framework APIs

Apps can use three types of native framework APIs provided by Android NDK to interact with system services or the kernel.

• **Type-1**: To interact with system services, Apps can invoke the APIs, which internally call local interfaces of services. For example, apps can call the `ACameraManager_openCamera` API exported by the `libcamera2ndk.so` library to request the camera service to open the camera.

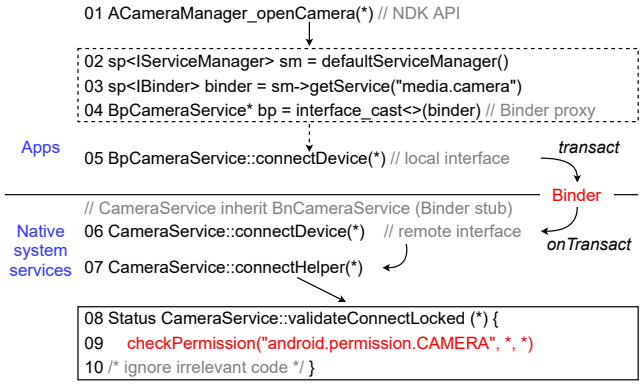


Fig. 2: An example of calling native framework APIs to interact with system services, and a case of the explicit permission check.

The internal of `ACameraManager_openCamera` is illustrated in Figure 2. Specifically, the API first obtains the Binder proxy of the camera service (in Line 2-4). Then, it uses the Binder proxy to invoke the `connectDevice` local interface (in Line 5), which internally calls the `transact` function to send the request to the Binder stub of the camera service. Subsequently, the `onTransact` function of the Binder stub handles the request and calls the `connectDevice` remote interface of the camera service (in Line 6) to open the camera.

TABLE IV: APIs for obtaining the Binder proxy of system services.

Library	API Declaration
libbinder_ndk.so	AServiceManager_getService(const char* instance)
libbinder_ndk.so	AServiceManager_checkService(const char* instance)

• **Type-2**: Since apps can call the APIs listed in Table IV to obtain the Binder proxy of system services, they can directly access the services' remote interfaces using the obtained Binder

proxy. Hence, we include remote interfaces of system services (i.e., Type-2 APIs) into the APIs provided by Android NDK.

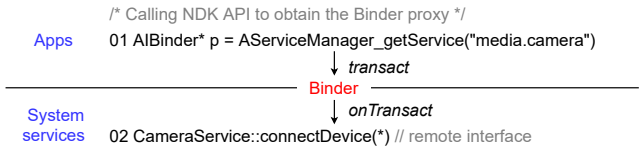


Fig. 3: An example of calling native framework APIs to obtain the Binder proxy to interact with native system services.

The example in Figure 3 shows how apps call the APIs in Table IV to request the camera service to open the camera. In detail, after obtaining the service's Binder proxy (in Line 1), apps directly invoke the `transact` function of the Binder proxy to call the `connectDevice` remote interface (in Line 2).

• **Type-3**: To interact with the kernel, Apps can call the APIs (e.g., `socket`), which internally invoke system calls.

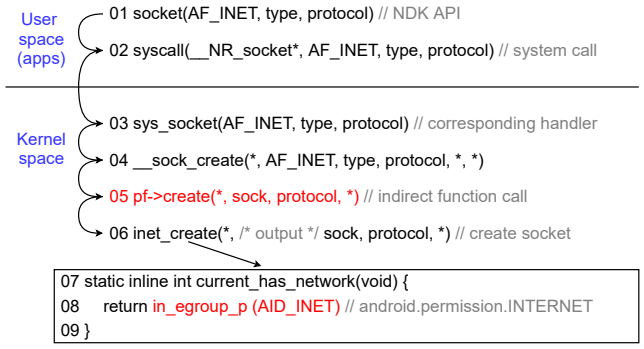


Fig. 4: An example of calling the native framework API to interact with the kernel, and a case of the implicit permission check.

Figure 4 shows the implementation details about `socket`. More specifically, `socket` calls the `syscall` function to invoke the `__NR_socket` system call (in Line 2), which will be handled by the `sys_socket` handler in the kernel (in Line 3).

B. Types of Permission Checks

Depending on the categories of Android permissions introduced in §II-D, we divide permission checks into two types, including the explicit checks for general permissions and the implicit checks for EGID related permissions.

• **Explicit Permission Checks in System Services**: Remote interfaces of system services can be called to perform sensitive operations, and thus they will call permission check functions to examine whether the apps have the required permissions. Since the string constant, representing the permission under check, will passed as a parameter to permission check functions [46], we treat such checks as explicit permission checks.

For instance, in Line 6-10 of Figure 2, the `connectDevice` interface calls the `checkPermission` function (i.e., the permission check function in Line 9) to examine whether the apps, requesting to open the camera, have been granted with the CAMERA permission. Since "android.permission.CAMERA" (i.e., the permission string in Line 9) is passed as a parameter to `checkPermission`, this case is an explicit permission check.

Moreover, since `ACameraManager_openCamera` internally calls `connectDevice` to complete its task, only the apps, having been granted with the CAMERA permission, are allowed to call this API. Accordingly, it is a permission restricted API (i.e., invoking this API requires specific permission).

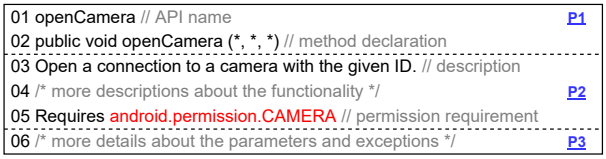
• **Implicit Permission Checks in Kernel**: Apps can invoke system calls to perform sensitive operations [29], and thus the kernel will perform authority checks on UIDs and GIDs of the apps' running processes. Commonly, it will call the `in_egroup_p` function (i.e., the EGID check function) to examine whether the processes are running with the required EGIDs. Since some of the EGIDs are correlated with permissions (e.g., those listed in Table III), the checks on those EGIDs can be seen as the checks on their corresponding permissions. Thus, we treat such EGID checks as implicit permission checks.

For example, in Line 3-9 of Figure 4, `sys_socket` internally calls `in_egroup_p` to check whether the processes, requesting to create the network socket, are running with AID_INET (in Line 8). Since this EGID is associated with the INTERNET permission, this case is an implicit permission check.

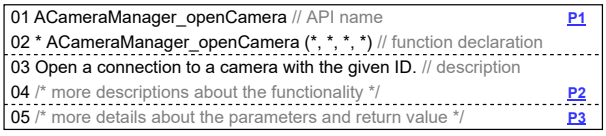
Moreover, since `socket` internally calls `sys_socket`, only the apps, having been granted with the INTERNET permission, are allowed to call this native framework API. Accordingly, it is a permission restricted API as well.

C. A Motivating Example

To help developers properly use Android framework APIs, Google provides the official API references [2], [6], which contain the description and permission specification for each API. However, the documentation of native framework APIs included in Android NDK lacks the permission specification for those permission restricted APIs.



(a) The reference for the SDK API, CameraManager.openCamera.



(b) The reference for the NDK API, ACameraManager_openCamera.

Fig. 5: The motivating example.

Figure 5 shows the official documentation of two framework APIs related to the camera service, both of which can be called to open the camera device. Figure 5a is summarized from the reference of the Java API `CameraManager.openCamera` [10]. Meanwhile, Figure 5b is summarized from the reference of the NDK API `ACameraManager_openCamera` [1]. Both of the API

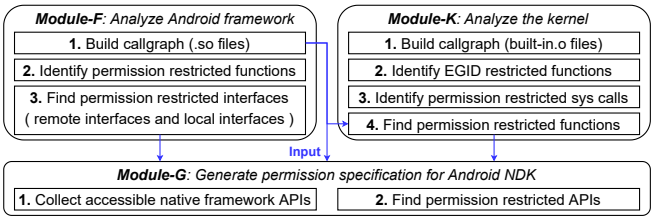


Fig. 6: The overview and workflow of PSGen.

references consist of three parts: `P1` presents the name and method declaration of the API; `P2` describes the functionality of the API, and this part also includes the permission specification that declares the required permissions for calling the API; `P3` provides additional information about the API, such as the parameters, the return value, and the thrown exceptions.

Since using camera requires the CAMERA permission [9], the references of the two APIs should contain the same permission specification. However, we find that the P2 part in Figure 5b does not contain the permission specification, whereas the P2 part (i.e., Line 5) in Figure 5a lists the permission information. That is, the official documentation lacks the necessary permission specification for `ACameraManager_openCamera`. Without including such the information in the API reference, app developers may fail to properly invoke this API in their apps [3]. More specifically, since app developers are not informed to request the CAMERA permission in their apps, the apps that call `ACameraManager_openCamera` to open the camera will fail to request the camera service.

Since existing studies on permission specification analysis for Android framework APIs cannot generate the permission specification for native framework APIs included in Android NDK, we design and implement a new tool named `PSGen`, which analyzes the native code of Android framework and the kernel to build the permission specification for native framework APIs. Specifically, towards this example (referring to Figure 2), `PSGen` first identifies the permission check enforced in `validateConnectLocked` (in Line 9). Then, it will find the permission restricted interfaces of the camera service (i.e., the `connectDevice` interfaces in Line 5,6), and associate them with the CAMERA permission under check. Subsequently, `PSGen` discovers that there exists reachable function call paths from the `ACameraManager_openCamera` API to the `connectDevice` local interface, and thus `PSGen` further correlates the CAMERA permission to `ACameraManager_openCamera`.

IV. PSGen

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

A. Overview

Figure 6 presents the architecture of `PSGen`, which consists of three modules, namely *Module-F* (detailed in §V), *Module-K* (detailed in §VI), and *Module-G* (detailed in §VII). `PSGen` analyzes the implementation of Android framework and the kernel to build the permission specification for Android NDK.

Module-F analyzes native system services and correlates the services' interfaces to the explicit permission checks enforced in them. Built upon SVF [48], an LLVM bitcode based static analysis tool for native code, *Module-F* analyzes LLVM bitcode of system libraries' `.so` files, which are generated by compiling C or C++ code of Android framework via LLVM [4].

Module-K analyzes the kernel of Android and associates the system library functions, which invoke system calls, to implicit permission checks enforced in system call handlers. *Module-K* is also built upon SVF and it analyzes LLVM bitcode of kernel modules' `built-in.o` files, which is generated by compiling the source code of the kernel via LLVM.

According to the three types of native framework APIs introduced in §III-A, *Module-G* builds the permission specification for Android NDK by identifying the permission restricted APIs, whose execution leads to permission checks. Specifically, it takes in the analysis results of *Module-F* and *Module-K* and the callgraph of Android framework to correlate permission restricted APIs to their corresponding permission checks.

B. Workflow

The workflow of each module in `PSGen` is illustrated in Figure 6 and we elaborate more on them as follows.

Module-F takes three steps to associate the permission checks enforced in native system services with their interfaces. First, to find permission checks enforced in native system services, this module builds the callgraph for the native code of Android framework, where native system services are implemented (see §V-A). Second, to identify the permission restricted interfaces of system services, whose execution leads to permission checks, this module analyzes each function in the callgraph to find the permission restricted functions that call permission check functions (see §V-B). Third, this module traverses the callgraph from each remote interface of system services to find the permission restricted interfaces and then correlates them to the corresponding permissions under check (see §V-C).

Module-K takes four steps to correlate the implicit permission checks (i.e., EGID checks) enforced in the kernel to their corresponding system library functions. First, to find EGID checks enforced in the kernel, this module builds the callgraph of the kernel (see §VI-A). Second, to identify the permission restricted system calls, the execution of their corresponding handlers will lead to EGID checks, this module analyzes each function in the callgraph to find the EGID restricted functions that call `in_egroup_p` to enforce EGID checks (see §VI-B). Third, to recognize the system library functions that invoke permission restricted system calls, this module traverses the callgraph from each system call handlers to find the permission restricted ones, and then maps them to their corresponding system calls (see §VI-C). Fourth, this module analyzes system library functions to find those that invoke permission restricted system calls and then associates them with the EGID related permissions under check (see §VI-D).

Module-G takes in the permissions associated with interfaces of system services and system library functions, as well as the callgraph of native code of Android framework to build the

permission specification for Android NDK through two steps. First, to collect the native framework APIs that are accessible to apps, this module parses NDK libraries to get their exported functions. Second, to find the permission restricted APIs, which internally call permission restricted interfaces or functions, this module traverses the callgraph from each app accessible API. *Module-G* will correlate each permission restricted API with its corresponding permissions under check. All these correlations form the permission specification for Android NDK.

V. Analyzing Native Code of Android Framework

This section describes the details of *Module-F*. Specifically, we introduce how this module builds the callgraph for native code of Android framework (in §V-A), identifies the permission restricted functions (in §V-B), and find the permission restricted interfaces of native system services (in §V-C).

A. Building Callgraph

Since the native code of Android framework, especially the implementations of native system services, are mainly dispersed in system libraries (e.g., `libcameraservice.so` contains the code for the camera service), we build the callgraph of each system library and then merge them together to form the complete callgraph. Specifically, we leverage SVF [48] to build the callgraph of each system library based on the LLVM bitcode of the library's `.so` file. During this process, we encounter two challenging issues, which will make the callgraph incomplete and inaccurate. In the following, we present the details about these issues and our approaches to addressing them.

• **Indistinguishable C++ Objects**: Different C++ classes can share the same LLVM bitcode representation. It makes SVF unable to distinguish their objects.

▷ **Details**: Each C++ class is represented by the types of its non-static fields in LLVM bitcode [15]. Accordingly, the LLVM bitcode representations of the child classes, inheriting the same parent class without adding additional non-static fields, cannot be distinguished from each other.

We find that the interface classes, which will be inherited by the classes of the Binder proxy or the classes of the Binder stub of native system services, are such kind of child classes, and thus processing them needs to address this issue. For example, as shown in Figure 7a, since both of the `IServiceManager` class (in Line 1-3) and the `ICameraService` class (in Line 6-8) inherit the `IInterface` class, and neither of them has non-static fields, these interface classes will share the same LLVM bitcode representation.

This issue makes SVF unable to correctly determine the types of C++ objects. As shown in Figure 7b, Line 6,8-9 (`LLVM` bitcode) show that the variables `sm` and `cs` are both the `IServiceManager` object. However, it is incorrect according to the corresponding source code in Line 2-3, which indicate that the types of `cs` should be `ICameraService`.

Since SVF requires accurate type information to perform precise points-to analysis to build callgraph [48], the incorrect type information incurred by this issue will negatively affect the analysis and make the callgraph incomplete and inaccurate.

```
// Definitions of IServiceManager and BpServiceManager in libbinder.so
01 class IServiceManager : public IInterface { // IServiceManager.h
02   /* IServiceManager is the interface class of the service manager service */
03   /* declare virtual functions and define static fields */ }      insert
04   int stub[10]; // differentiate IServiceManager from ICameraService
05 class BpServiceManager : public IServiceManager { /* ignore the code */ }
// Definitions of ICameraSerivce and BpCameraService in libcamera_client.so
06 class ICameraService : public IInterface { // ICameraService.h
07   /* ICameraService is the interface class of the camera service */
08   /* declare virtual functions and define static fields */ }      insert
09   int stub[20]; // differentiate ICameraService from IServiceManager
10 class BpCameraService : public ICameraService { /* ignore the code */ }
```

(a) Class declarations of IServiceManager and ICameraService.

```
// C++ Source code of getCameraService in libcamera2ndk.so
01 sp<ICameraService> CameraManagerGlobal::getCameraService(*) {
02   sp<IServiceManager> sm = defaultServiceManager();
03   sp<ICameraService> cs = sm->getService("media.camera");
04   cs->addListener(*); // register the camera status listener
05   /* ignore the irrelevant C++ code */ }
// LLVM-bitcode of getCameraService
06 %"class.android::sp" = type { %"class.android::IServiceManager"* }
07 define void @CameraManagerGlobal16getCameraServiceEv(*) {
08   %sm = alloca %"class.android::sp" // an IServiceManager object
09   %cs = alloca %"class.android::sp" // an IServiceManager object ( X )
10   %vtable = * // get vtable for class of %sm (vtable for IServiceManager)
11   %vfn = *, i32 4 // get the 5th function pointer in vtable
12   %call = call * // call the virtual function reference by %vfn ( ? )
13   %vtable2 = * // get vtable for class of %cs (vtable for IServiceManager)
14   %vfn2 = *, i32 9 // get the 10th function pointer in vtable2
15   %call2 = call * // call the virtual function referenced by %vfn2 ( ? )
16   /* ignore the irrelevant LLVM-bitcode */ }
```

(b) The C++ code and LLVM bitcode related to getCameraService.

```
01 %"class.android::sp" = type { %"class.android::IServiceManager"* }
02 %"class.android::sp.1" = type { %"class.android::ICameraService"* }
03 @_ZTVN*BpServiceManagerE = * // vtable for BpServiceManager
04 @_ZTVN*BpCameraServiceE= * // vtable for BpCameraService
05 define void @CameraManagerGlobal16getCameraServiceEv(*) {
06   %sm = alloca %"class.android::sp"  // an IServiceManager object
07   %cs = alloca %"class.android::sp.1" // an ICameraService object ( √ )
08   /* omit Line 11,14 in Figure.(b) */
09   %vtable = * // refer to vtable of BpServiceManager
10   %call = * // call BpServiceManager::getService ( √ )
11   %vtable2 = * // refer to vtable of BpCameraService
12   %call2 = * // call BpCameraService::addListener ( √ )
13   /* ignore the irrelevant LLVM-bitcode */ }
```

(c) The adjusted LLVM bitcode related to getCameraService.

Fig. 7: Adjusting LLVM bitcode of system libraries.

▷ **Solution**: We insert extra non-static fields to the interface classes (e.g., `IServiceManager` and `ICameraService`) in order to make their `LLVM` bitcode representations different.

To accomplish this task, we insert arrays with different length to the interface classes. For instance, as illustrated in Figure 7a, we add an integer array with 10 elements (in Line 4) to `IServiceManager` and an integer array with 20 elements (in Line 9) to `ICameraService`, respectively. As a result, the two interface classes have different non-static fields. Hence, their LLVM bitcode representations become different as presented in Line 1-2 of Figure 7c. Accordingly, `SVF` can correctly figure out that the `cs` variable is an `ICameraService` object.

• **Unknown Virtual Function Calls**: The LLVM bitcode of a

library does not include its dependent libraries' LLVM bitcode. Thus, it lacks the information about the virtual functions defined in the classes of other libraries. Accordingly, by analyzing each system library individually, SVF cannot completely resolve the virtual function calls from the library's LLVM bitcode.

▷ **Details**: When compiling the source code of the C++ class that implements virtual functions, LLVM will save the pointers of these functions to an array called virtual function table (a.k.a `vtable`) [15]. More specifically, LLVM adds a `vtable` variable, representing the virtual function table, to the class's LLVM bitcode. Since this variable contains the information (e.g., function names) about the class's virtual functions, SVF relies on it to resolve virtual function calls.

Since the source code of a system library does not contain the source code of its dependent libraries, so does the library's LLVM bitcode. Therefore, a library's LLVM bitcode will not include the variables, storing the information about the virtual functions defined in the C++ classes of other libraries.

This issue makes SVF unable to completely resolve the virtual function calls. For example, referring to Figure 7a and 7b, since the `getCameraService` function is implemented in `libcamera2ndk.so` while the `BpCameraService` class is defined in `libcamera_client.so`, the `LLVM` bitcode of `getCameraService` will not contain the `vtable` variable of `BpCameraService`. Thus, when analyzing the LLVM bitcode in Line 13-15 of Figure 7b, `SVF` cannot recognize the corresponding virtual function call to `addListener` in Line 4.

Since numerous virtual function calls in system libraries cannot be resolved, this issue makes callgraph incomplete.

▷ **Solution**: We link the `LLVM` bitcode of each system library with the LLVM bitcode of its dependent libraries. Then, the `LLVM` bitcode of the system library will contain the demanded `vtable` variables of the classes defined in other libraries.

Specifically, we take two steps to finish this task. First, we use `llvm-objdump` [14] to retrieve each system library's dependent libraries. Second, we use `llvm-link` [13], a `LLVM` bitcode linker, to merge the `LLVM` bitcode of the dependent libraries to that of the system library. For example, we link the `LLVM` bitcode of `libcamera2ndk.so` with those of `libbinder.so` and `libcamera_client.so`, the adjusted `LLVM` bitcode (shown in Figure 7c) includes the vtable variables (in Line 3-4) for resolving the virtual function calls. Then, `SVF` can successfully resolve the virtual function calls to `addListener`.

B. Identifying Permission Restricted Functions

To find the permission restricted interfaces of native system services, we identify the permission restricted functions, where permission checks are enforced. Precisely, these permission restricted functions will call permission check functions (e.g., the `checkPermission` function in Line 9 of Figure 2), which take the permission string as the parameter, to enforce permission checks (see §III-B). Hence, we treat the callers of permission check functions as permission restricted functions (e.g., the `validateConnectLocked` function in Line 8 of Figure 2).

More specifically, we take three steps to find permission restricted functions from the LLVM bitcode of each system library. First, we find the permission strings in LLVM bitcode. Second, for each permission string, we conduct data flow analysis on its def-use chain constructed by SVF to find the permission check function that consumes the string. Third, we get the callers of each permission check function, which are the permission restricted functions.

For each identified permission restricted function (F_p), we also record the permission under check (P) and store them in a map $M_{fp} : \{F_p \rightarrow P\}$, which will then be used to determine the permission restricted remote interfaces of systems services.

C. Finding Permission Restricted Interfaces

Module-F traverses the callgraph to find permission restricted remote interfaces and local interfaces of native system services. During this process, we record services' remote interfaces in a set S_{ri}, which will then be used to determine the app-accessible native framework APIs (in §VII-A).

• **Finding Permission Restricted Remote Interfaces**: Since the `onTransact` function of the Binder stub will call remote interfaces of system services (see §II-A), we record the callees of `onTransact` (e.g., the `connectDevice` remote interface in Line 6 of Figure 2) to S_{ri} and analyze them to identify the permission restricted remote interfaces. Specifically, we traverse the callgraph from each callee of `onTransact` to find whether there are reachable paths from it to the permission restricted functions that are stored in M_{fp}. If so, the callee is a permission restricted remote interface.

Since a permission restricted remote interface (R_p) may access multiple permission restricted functions, we correlate it with a set of permissions (S_{rp}). For each reachable permission restricted function, we query M_{fp} to retrieve the permission under check. For example, S_{rp} of the `connectDevice` remote interface is $\{CAMERA\}$. We store such the correlation to the map $M_{rp} : \{R_p \rightarrow S_{rp}\}$, which will then be used in the analysis of permission restricted local interfaces.

• **Finding Permission Restricted Local Interfaces**: A permission restricted local interface is always correlated with a permission restricted remote interface, because the latter is invoked by the former. Based on this observation, we identify the permission restricted local interfaces through two steps. First, since local interfaces will call the `transact` function (see §II-A), we get the callers of the `transact` function (e.g., the `connectDevice` local interface in Line 5 of Figure 2) from the callgraph in order to collect the local interfaces. Second, since a pair of local interfaces and remote interfaces share the same method declaration (see §II-A), for each caller of `transact`, we search M_{rp} to check whether it has a corresponding permission restricted remote interface. If so, we find a permission restricted local interface. Note that, during this process, we remove the functions from S_{ri}, which are not services' remote interfaces because they do not share the same method declaration with any of the local interfaces.

A pair of permission restricted local interfaces (L_p) and remote interfaces (R_p) should be protected by the same set of permissions. Therefore, we query M_{rp} to get the permission set S_{lp} for L_p. For example, S_{lp} of the `connectDevice` local interface is $\{CAMERA\}$, which is the same as S_{rp} of the `connectDevice` remote interface. We store such the correlation to the map $M_{lp} : \{L_p \rightarrow S_{lp}\}$, which will be used to build the permission specification for native framework APIs (in §VII).

VI. Analyzing Android Kernel

This section describes the details of *Module-K*. Specifically, we present how this module builds the callgraph of Android kernel (in §VI-A), identifies the EGID restricted kernel functions (in §VI-B), and find the system calls and system library functions (in §VI-C and §VI-D), whose executions are restricted by EGID related permissions.

A. Building Callgraph

When compiling source code of the kernel, the compiler (e.g., LLVM) will generate an object file named `built-in.o` for each module of the kernel. These object files are then statically linked to generate a single executable file (i.e., `vmlinux`) for the kernel [12]. Since `vmlinux` contains all necessary code of the kernel, we can directly perform static code analysis on it to build the callgraph. Specifically, we apply SVF and KMI [20], a tool specially designed to resolve the indirect function calls in the kernel, to analyzing the LLVM bitcode of `vmlinux`. However, it is non-trivial to build the entire callgraph of the kernel due to its huge code base. In the following, we present the details about the issue and our approach to solving it.

• **High Resource Consuming Points-to Analysis**: Due to the huge and complex code base of the kernel, it is both memory and time consuming for SVF to conduct points-to analysis on `vmlinux` to build the entire callgraph of the kernel [57].

▷ **Details**: SVF will track all the objects (e.g., functions and variables) included in the target's LLVM bitcode to perform precise points-to analysis for building the accurate callgraph [48]. Therefore, it is unscalable to the complex Android kernel with over 2M LOC [5]. We have deployed SVF on a machine with 192 GB memory and applied it to analyzing `vmlinux`. Unfortunately, SVF has exhausted all the memory in around 2 hours without finishing the points-to analysis.

Without the results of the precise points-to analysis, SVF cannot build the complete and accurate callgraph for the kernel.

▷ **Solution**: Since `vmlinux` is composed of `built-in.o` files, we leverage SVF and KMI to build the callgraph of each `built-in.o` file instead, and then merge them together to form the entire callgraph of the kernel. It is noteworthy that, in our experiment, SVF can build the callgraph for each object file with the need of no more than 64 GB memory.

B. Identifying EGID Restricted Kernel Functions

To find the permission restricted system call handlers, we identify the EGID restricted kernel functions, where EGID checks are enforced. Specifically, these EGID restricted functions call the `in_egroup_p` function, which takes the value of EGID as the parameter (e.g., those listed in Table III), to conduct EGID checks (see §III-B). Accordingly, we treat the callers of the `in_egroup_p` function as EGID restricted functions (e.g., `current_has_network` in Line 7 of Figure 4).

In detail, we identify EGID restricted functions from the LLVM bitcode of each `built-in.o` file through two steps. First, we locate the function call to `in_egroup_p` in LLVM bitcode and then perform data flow analysis on its parameter to determine the value of EGID under examination. Since there is a gap between the EGID related permission and the value of EGID under examination, we build a map (M_{egid}) between them. Specifically, we parse the `platform.xml` file of Android system [22], where the correspondence between each EGID related permission and its EGID is defined. Second, for each function call to `in_egroup_p`, we get its callers, which are the EGID restricted functions.

For each identified EGID restricted function (F_e), we query M_{egid} to get the EGID related permission (E) according to the value of EGID under examination. We store such the correlation in a map $M_{fe} : \{F_e \rightarrow E\}$, which will be used to determine the permission restricted system call handlers.

C. Identifying Permission Restricted System Calls

To find the system library functions, whose executions are restricted by EGID related permissions, we identify the permission restricted system calls (e.g., `__NR_socket` in Line 3 of Figure 4), whose handler will enforce EGID checks.

In detail, we traverse the callgraph of the kernel from each system call handler, whose name commonly starts with "sys_" [39], to find whether there are reachable paths from it to the EGID restricted functions stored in M_{fe}. If so, a permission restricted system call handler is found. To further get the system call from the handler, we build a map ($M_{handler}$) between them. Specifically, we parse the `unist.h` file of the kernel [25], where the correspondence between each system call and its corresponding handler is defined. Then, we can query $M_{handler}$ to get the permission restricted system call (S_p) of each found permission restricted system call handler.

Since a permission restricted system call handler may access multiple EGID restricted kernel functions, we associate S_p to a set of EGID related permissions (S_{sp}). For each reachable EGID restricted function, we query M_{fe} to get the permission under check. For example, S_{sp} of `__NR_socket` is $\{INTERNET\}$. We store such the correlation to the map $M_{sp} : \{S_p \rightarrow S_{sp}\}$, and will use it to find the system library functions, whose executions are restricted by EGID related permissions.

D. Finding Permission Restricted System Library Functions

Since the system library functions, whose executions are restricted by EGID related permissions, should call the `syscall` function to invoke the permission restricted system calls in M_{sp}, we analyze the callers of `syscall` to identify such permission restricted system library functions through two steps. First, we locate each function call to `syscall` in the

LLVM bitcode of each system library and then get its caller. Second, we perform data flow analysis on the first parameter of `syscall` to determine whether the system call number refers to a permission restricted system call in M_{sp}. If so, the caller of `syscall` is a permission restricted system library function.

However, we find that LLVM will not generate the bitcode for the wrapper functions of system calls written by assembly code (introduced in §II-B). Thus, we need to correlate each wrapper function to the corresponding system call. Specially, we parse the assembly files of wrapper functions (e.g., the `socket.S` file [23] for `socket`) to find the system call number. Similarly, if the system call number refers to a permission restricted system call in M_{sp}, the wrapper function is a permission restricted system library function.

For each identified system library function (L_e) whose executions are restricted by EGID related permissions, we query M_{sp} to retrieve the set of permissions under check (S_{le}) according to the system call number. For example, S_{le} of `socket` is $\{INTERNET\}$. We store such the correlation in the map $M_{le} : \{L_e \rightarrow S_{le}\}$, which will be used to build the permission specification for native framework APIs (in §VII).

VII. Generating Permission Specification

This section describes the details of *Module-G*. Specifically, we present how this module collects native framework APIs that are accessible to apps (in §VII-A), and identifies the permission restricted native framework APIs (in §VII-B).

A. Collecting App-Accessible Native Framework APIs

The main purpose of the permission specification for Android NDK is to guide app developers to properly use the native framework APIs in their apps. Therefore, before we generate the permission specification for each native framework APIs, we collect the APIs that can be called by native code of apps. According to the types of native framework APIs presented in §II-C, we collect these app-accessible APIs from two aspects. (1) For Type-1 and Type-3 APIs, which are defined in the system libraries of Android NDK, we collect them from the exported symbol table of NDK libraries [17] (e.g., those listed in Table II). Specifically, we use `objdump` [18], a utility for dumping information from object files, to retrieve the exported functions of NDK libraries. (2) For Type-2 APIs (i.e., remote interfaces of system services), we collect them from S_{ri} (see §V-C), which stores the services' remote interfaces. All these app-accessible APIs are recorded in the set S_{api}, from which we further identify the permission restricted APIs.

B. Finding Permission Restricted Native Framework APIs

We identify the native framework APIs, which will call permission restricted functions, to build the permission specification for Android NDK. Specifically, we traverse the callgraph of Android framework (built in §V-A) from each app-accessible API stored in S_{api} to determine whether there are reachable paths from it to the permission restricted interfaces of system services stored in M_{rp} and M_{lp} (see §V-C) or other permission restricted system library functions stored in M_{le} (see §VI-D). If so, we find a permission restricted native framework API.

A permission restricted API (R_{api}) may require apps to gain multiple permissions, and thus we correlate it with a set of permissions (P_{api}). Specifically, for each reachable permission restricted function, we query M_{rp}, M_{lp}, or M_{le} to retrieve the permissions under check. All the correlations between R_{api} and P_{api} (as those shown in Table V) form the permission specification for Android NDK.

TABLE V: A part of permission specification for Android NDK.

Permission Restricted API (R_{api})	Required Permissions (P_{api})
ACameraManager_openCamera (Type-1)	android.permission.CAMERA
CameraService::connectDevice (Type-2)	android.permission.CAMERA
socket (Type-3)	android.permission.INTERNET

VIII. Evaluation

We evaluate the performance of `PSGen` by answering the following three research questions (RQs).

RQ1: Can `PSGen` generate the permission specification for the Type-1 native framework APIs that require permissions?

RQ2: Can `PSGen` identify the permission restricted Type-2 APIs and associate them with the required permissions?

RQ3: Can `PSGen` derive the correlations between the Type-3 native framework APIs and their corresponding permissions?

TABLE VI: Overview of the framework and the kernel under analysis.

ID	Framework Version	#Function	Kernel Version	#Function
S1	android-9.0.0_r46	359,750	common-android-4.4	63,438
S2	android-10.0.0_r41	409,605	common-android-4.4	63,438
S3	android-11.0.0_r21	496,649	common-android-4.9	57,781

• **Data Set**: To answer the research questions, we use `PSGen` to analyze 3 Android systems (i.e., *S1*, *S2*, and *S3*), each of which is composed by a pair of Android framework and Android kernel. Table VI lists the details about the framework and kernel under evaluation, including their versions and the number of functions (#Function) included in their LLVM bitcode. In detail, *S1* consists of the framework of Android 9.0 and the common Android kernel 4.4. *S2* is composed by the framework of Android 10.0 and the common Android kernel 4.4. *S3* is made up of the framework of Android 11.0 and the common Android kernel 4.9. It is worth noting that these pairs of Android framework and Android kernel are close to those deployed on Pixel [8]. Meanwhile, the versions of the framework under analysis are the three most popular ones and they took about 70% of the market share worldwide on March 2021 [16].

When compiling the source code of each framework and kernel under analysis, we use WLLVM [26] to link the LLVM bitcode for each object file (i.e., `.o` or `.obj` file) of a system library or a kernel's module to a single LLVM bitcode file so that `PSGen` can get the complete LLVM bitcode of the target.

TABLE VII: Overview of the permission specification for native framework APIs.

Android System	Type-1 APIs			Type-2 APIs			Type-3 APIs			Total		
	#API	#FP	Precision	#API	#FP	Precision	#API	#FP	Precision	#API	#FP	Precision
Android 9.0 + Kernel 4.4 (*S1*)	2	0	100%	106	4	96.2%	25	2	92.0%	133	6	95.5%
Android 10.0 + Kernel 4.4 (*S2*)	7	0	100%	188	3	98.4%	18	2	88.9%	213	5	97.7%
Android 11.0 + Kernel 4.9 (*S3*)	7	0	100%	192	14	92.7%	19	2	89.5%	218	16	92.7%

• **Overall Results**: We list the overall results of our permission specification analysis for Android NDK in Tables VII and VIII. More specifically, Table VII presents the number of native framework APIs (#API) that require permissions to invoke them. Meanwhile, we also include the number of false positives (#FP) and the precision of PSGen in correlating the APIs to their required permissions in Table VII. In addition, Table VIII presents the distribution of the number of permissions (#Permission) required by the permission restricted APIs.

TABLE VIII: Overview of number of permissions required by APIs.

#Permission	1	2	3	4
Android 9.0 + Kernel 4.4 (*S1*)	118 (92.9%)	8 (6.3%)	0 (0.0%)	1 (0.8%)
Android 10.0 + Kernel 4.4 (*S2*)	97 (46.6%)	104 (50.0%)	5 (2.4%)	2 (1.0%)
Android 11.0 + Kernel 4.9 (*S3*)	94 (46.5%)	97 (48.0%)	9 (4.5%)	2 (1.0%)

A. Permission Specification Analysis for Type-1 APIs

As presented in Table VII, for the system *S1*, PSGen identifies 2 Type-1 native framework APIs that require permissions to call them. For *S2* and *S3*, PSGen discovers 7 such the kind of Type-1 APIs. We find that each of these APIs is correlated with only one permission, but the official API references do not provide the permission specifications for them.

To evaluate the precision of PSGen, we examine the generated permission specification by manually inspecting the source code of corresponding APIs. Specifically, no false positives is found, and thus the precision of PSGen in generating the permission specification for the Type-1 APIs is 100%.

Answer to RQ1: PSGen can precisely generate the permission specification for the Type-1 native framework APIs with the precision of 100%.

B. Permission Specification Analysis for Type-2 APIs

For each of the systems under evaluation, PSGen identifies 106, 188, and 192 permission restricted Type-2 native framework APIs, respectively, whose executions lead to permission checks. From their permission specifications, we observe that a majority of (about 95%) these APIs are correlated with one or two permissions, and few of them are associated with more than two permissions (as shown in Table VIII).

Similarly, we manually inspect the results to assess the precision of PSGen. Specifically, PSGen identifies the permission-restricted Type-2 APIs with the precision of 96.2%, 98.4%, and 92.7%, respectively. We find that the main reason for causing the false positives is the imprecise points-to analysis (i.e., Andersen's points-to analysis [31]) adopted to build callgraphs.

Answer to RQ2: PSGen can precisely identify the permission restricted Type-2 APIs and correlate them with their required permissions with the precision of over 92.7%.

C. Permission Specification Analysis for Type-3 APIs

PSGen derives 25, 18, 19 correlations between the permission restricted Type-3 native framework APIs and their corresponding permissions for the systems under evaluation, respectively. Although some of these Type-3 APIs (e.g., `socket`) are commonly used by apps [29], all of their permission specification is not available in the official API references from Google.

We further find that all these Type-3 APIs are correlated with the same permission `android.permission.INTERNET`, an EGID related permission. After manually analyzing the relevant source code of Android kernel, we notice that, in higher versions of Android kernel, they remove almost all the other EGID checks and only allow the root user to perform the sensitive operations, which are previously restricted by those EGID related permissions (e.g., those listed in Table III).

We also manually check the permission specification derived by PSGen and find two false positives. In detail, the precision of PSGen in analyzing the permission specification for Type-3 APIs is 92.0%, 88.9%, and 89.5%, respectively. Moreover, we analyze the false positives to find the root cause. Specifically, since the algorithm adopted by PSGen to build callgraphs is path-insensitive, the condition that makes the callgraph edge feasible is ignored, resulting in false positives.

Answer to RQ3: PSGen can precisely derive the correlations between the Type-3 native framework APIs and their corresponding permissions with the precision of over 88.9%.

IX. Threat to Validity

The threat to the external validity is mainly caused by the imprecise static analysis employed by PSGen. Due to the huge code base of Android framework and Android kernel, PSGen chooses the scalable but imprecise path-insensitive points-to analysis to build their callgraphs. However, the imprecise static code analysis causes false positives and degrades the precision of PSGen in generating the permission specification for native framework APIs as presented in §VIII. To mitigate the problem, in future work, we will try to use the more precise path-sensitive points-to analysis [47] to build the callgraphs.

X. Related Work

Although researchers have proposed various work on analyzing Android SDK, to the best of our knowledge, none of the

existing work targets at analyzing Android NDK. Meanwhile, most of the existing studies (e.g., [37], [45], [49], [52]–[56], [59]) focus on analyzing the Android SDK APIs called by apps, and only few of them [28], [32], [33], [35], [36], [43] conduct the permission specification analysis for Android SDK. PScout [32] statically identifies the permission check methods and builds the context-insensitive callgraph of Android framework. Then, it performs backward reachability analysis to construct the mapping between Java framework APIs and their required permissions. Alexandre et al. [35] evaluated the performance of two static analysis methods (CHA and Spark) on building the permission specification for Java framework APIs. AXPLORER [33] models the runtime behaviors of several complicated Java classes of Android framework to promote the accuracy of permission specification analysis. ARCADE [28] constructs the path-sensitive callgraph of Android framework to make the Android API protection mapping more precise. HEAPHELPER [43] leverages the dynamic information stored in the heap of Android framework to assist the construction of a more precise callgraph, which makes the generated Android permission specification more accurate. Besides the tools built upon static analysis, DYNAMO [36] generates the permission specification by dynamically executing Java framework APIs and recording the permissions being checked. Since none of them analyze native code of Android framework, we cannot use them to build the permission specification for native framework APIs.

One of the applications of permission specification analysis is to discover the inconsistent permission checks in Android framework. Kratos [46], AceDroid [27], and ACMiner [40] statically build the permission specification for Java framework APIs and then examine whether the two APIs, implementing the same functionality, are protected by the same permission. However, since these tools just focus on discovering the inconsistent permission specification in Java framework APIs, our work can help analysts to uncover the inconsistent permission checks enforced for native framework APIs.

XI. Conclusion

We conduct the first permission specification analysis for Android NDK by developing `PSGen`, a novel automated tool that statically analyzes the implementation of Android framework and Android kernel to correlate native framework APIs with their required permissions. Applying `PSGen` to 3 Android systems spanning from Android 9.0 to 11.0, we find that `PSGen` can precisely build the permission specification for Android NDK. We discover more than 200 native framework APIs that are correlated with at least one permission.

XII. Acknowledgment

We thank the anonymous reviewers for their helpful comments. This research is partially supported by the Hong Kong RGC Project (No. PolyU15223918), Hong Kong ITF Project (No. ITS/197/17FP), and the National Natural Science Foundation of China (No.62072046, 61872057), and National Key R&D Program of China (2018YFB0804100), 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] "ACameraManager_openCamera," https://developer.android.com/ndk/reference/group/camera?#acameramanager_opencamera, 2021.
[2] "Android API reference," https://developer.android.com/reference, 2021.
[3] "Android Camera2 executable failed to get frames," https://stackoverflow.com/questions/52710811/android-camera2-executable-failed-to-get-frames, 2021.
[4] "Android Clang/LLVM Toolchain," https://android.googlesource.com/toolchain/llvm_android/+/master, 2021.
[5] "Android Common Kernels," https://source.android.com/devices/architecture/kernel/android-common, 2021.
[6] "Android NDK API Reference," https://developer.android.com/ndk/reference, 2021.
[7] "Binder," https://developer.android.com/reference/android/os/Binder, 2021.
[8] "Building Kernels," https://source.android.com/setup/build/building-kernels, 2021.
[9] "Camera API," https://developer.android.com/guide/topics/media/camera#manifest, 2021.
[10] "CameraManager.openCamera," https://developer.android.com/reference/android/hardware/camera2/CameraManager?#openCamera(java.lang.String,android.hardware.camera2.CameraDevice.StateCallback,android.os.Handler), 2021.
[11] "Credentials in Linux," https://www.kernel.org/doc/html/v4.15/security/credentials.html, 2021.
[12] "Kernel Size Tuning Guide," https://elinux.org/Kernel_Size_Tuning_Guide, 2021.
[13] "LLVM bitcode linker," http://llvm.org/docs/CommandGuide/llvm-link.html, 2021.
[14] "LLVM's object file dumper," https://llvm.org/docs/CommandGuide/llvm-objdump.html, 2021.
[15] "Mapping High Level Constructs to LLVM IR," https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/README.html, 2021.
[16] "Mobile Android Version Market Share Worldwide," https://gs.statcounter.com/android-version-market-share/mobile/worldwide/, 2021.
[17] "NDK libraries," https://developer.android.com/ndk/guides/libs, 2021.
[18] "Object file dumper," https://man7.org/linux/man-pages/man1/objdump.1.html, 2021.
[19] "Permissions overview," https://developer.android.com/guide/topics/permissions/overview, 2021.
[20] "PeX," https://github.com/lzto/pex#resolve-indirect-call-kmi-or-cvf, 2021.
[21] "Platform Architecture," https://developer.android.com/guide/platform, 2021.
[22] "platform.xml," https://cs.android.com/android/platform/superproject/+/master:frameworks/base/data/etc/platform.xml, 2021.
[23] "socket.S," https://android.googlesource.com/platform/bionic/+/db1ea34/libc/arch-x86/syscalls/socket.S, 2021.
[24] "System Calls," https://www.gnu.org/software/libc/manual/html_node/System-Calls.html, 2021.
[25] "unistd.h," https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/unistd.h, 2021.
[26] "Whole Program LLVM," https://github.com/travitch/whole-program-llvm, 2021.
[27] 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.
[28] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li, "Precise Android API Protection Mapping Derivation and Reasoning," in *Proc. CCS*, 2018.
[29] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proc. NDSS*, 2016.
[30] S. Almanee, A. Unal, and M. Payer, "Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code," in *Proc. ICSE*, 2021.
[31] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[32] K. W. Y. Au, Y. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proc. CCS*, 2012.
[33] 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.
[34] 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.
[35] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.
[36] A. Dawoud and S. Bugiel, "Bringing balance to the force: Dynamic analysis of the android application framework," in *Proc. NDSS*, 2021.
[37] M. Fan, L. Yu, S. Chen, H. Zhou, X. Luo, S. Li, Y. Liu, J. Liu, and T. Liu, "An empirical evaluation of GDPR compliance violations in Android mHealth apps," in *Proc. ISSRE*, 2020.
[38] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. CCS*, 2011.
[39] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proc. RAID*, 2020, pp. 443–458.
[40] 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.
[41] A. Grünbacher, "POSIX Access Control Lists on Linux," in *Proc. USENIX ATC*, 2003.
[42] 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.
[43] L. Luo, "Heap Memory Snapshot Assisted Program Analysis for Android Permission Specification," in *Proc. SANER*, 2020.
[44] C. Qian, X. Luo, Y. Shao, and A. Chan, "On tracking information flows through jni in android applications," in *Proc. DSN*, 2014.
[45] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, 2015.
[46] 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.
[47] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proc. PLDI*, 2018.
[48] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proc. CC*, 2016.
[49] Y. Tang, X. Zhan, H. Zhou, X. Luo, Z. Xu, Y. Zhou, and Q. Yan, "Demystifying application performance management libraries for android," in *Proc. ASE*, 2019.
[50] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang, "Exception beyond Exception: Crashing Android System by Trapping in "Uncaught Exception"," in *Proc. ICSE*, 2017.
[51] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 814–828, 2019.
[52] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "Packergrind: An adaptive unpacking system for android apps," *IEEE Transactions on Software Engineering*, 2020.
[53] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking Android Apps via a Hardware-Assisted Approach," in *Proc. S&P*, 2021.
[54] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. K. Leung, "PPChecker: Towards Accessing the Trustworthiness of Android Apps' Privacy Policies," *IEEE Transactions on Software Engineering*, 2018.
[55] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android apps," in *Proc. ICSE*, 2021.
[56] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *Proc. ASE*, 2020.
[57] 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.
[58] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proc. CCS*, 2013.
[59] H. Zhou, H. Wang, Y. Zhou, X. Luo, Y. Tang, L. Xue, and T. Wang, "Demystifying diehard android apps," in *Proc. ASE*, 2020.