Securing Applications against Side-Channel Attacks through Resource Access Veto

TOUSIF OSMAN, Concordia University, Canada
MOHAMMAD MANNAN, Concordia University, Canada
URS HENGARTNER, University of Waterloo, Canada
AMR YOUSSEF, Concordia University, Canada

Apps on modern mobile operating systems can access various system resources with, or without, an explicit user permission. Although the OS generally maintains strict separation between apps, an app can still get access to another app's private information, such as the user input, through numerous side-channels For example, keystrokes and swipe gestures from a victim app can be inferred indirectly from the accelerometer or gyroscope output, allowing a zero-permission app to learn sensitive inputs such as passwords from the victim's app. Current mobile OSes allow an app to defend itself in such situations only in some exceptional cases—e.g., by blocking screenshot captures in Android.

In this paper, we propose a general mechanism for apps to defend themselves from any unwanted implicit or explicit interference from other concurrently running apps. Our `AppVeto` solution enables an app developer to easily configure an app's requirements for a *safe* environment; a foreground app can request the OS to *disallow* access—i.e., to enable veto powers—to selected side-channel-prone resources to *all* other running apps for a certain (short) duration, e.g., no access to the accelerometer during password input. In a sense, we enable a finer-grained access control policy than the current runtime permission model. We implement AppVeto on Android using the Xposed framework and PLT hooking techniques, without changing Android APIs. Furthermore, we show that AppVeto imposes negligible overhead, while being effective against several well-known side-channel attacks—implemented via *both* Android Java and/or Native APIs.

CCS Concepts: • **Security and privacy → Mobile platform security**.

Additional Key Words and Phrases: Mobile applications security, mobile operating systems, permission management, side-channel attacks

ACM Reference Format:
Tousif Osman, Mohammad Mannan, Urs Hengartner, and Amr Youssef. 2020. Securing Applications against Side-Channel Attacks through Resource Access Veto. *Digit. Threat. Res. Pract.* 1, 1, Article 1 (August 2020), 30 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Modern smartphones are commonly equipped with various hardware sensors (e.g., microphone, GPS, light sensor, and accelerometer) to interact with the physical world. Smartphones also have access to personal and security-sensitive user information such as contact lists, photos, and passwords. Accessing these sensors/resources and user information by third-party apps is controlled by the operating system (OS), with explicit user approval in some cases (either at the install-time of the app or during its run time, see e.g., [3, 20]).

Contact email: m.mannan@concordia.ca.
Authors' addresses: Tousif Osman, Concordia University, Montreal, Canada; Mohammad Mannan, Concordia University, Montreal, Canada; Urs Hengartner, University of Waterloo, Waterloo, Canada; Amr Youssef, Concordia University, Montreal, Canada.

2576-5337/2020/8-ART1
https://doi.org/10.1145/1122445.1122456

Strict separation of app data is also enforced by the leading OSes. Current permission models enable app developers and OSes to offload many security-critical decisions to users, who usually can barely understand the privacy and security implications of such decisions, see e.g., [10, 49]. Recent access-control changes in Android and iOS are designed to provide more control over these resources; such changes include: background usage permission for location in Android 10 [22], iOS 13 [2], and one-time permission for location, camera, and microphone in Android 11 [23]. Even with these improvements, users still can allow an app to always access these resources (especially for long-running and heavily used apps); i.e., until uninstalled, such apps can keep using the granted permissions.

On the other hand, resources that are considered to pose little or no security/privacy risks, such as the accelerometer or the gyroscope, can be used by any app without the user's knowledge or consent. Many side-channel attacks have been demonstrated using these so-called non-dangerous or normal resources [53, 58, 61], as well as resources that require explicit user consent [5, 47, 54]. Besides compromising PINs and passwords, a recent attack [5] shows that the seemingly benign accelerometer can also be used to eavesdrop on the phone's speaker. Current user-approval based permission models cannot tackle these stealthy but highly-effective attacks.

On the positive side, recent OS versions provide some limited defences against these side-channel attacks. As of Android 9.0, apps by default can no longer access sensors, such as the accelerometer and the gyroscope, the microphone, and the camera, while running in the background [12] without launching a foreground service [14] visible to users as an icon. The AudioPlaybackCapture API in Android 10 which enables an app to capture audio from another app [59], also offers opt-out methods with which an app can prevent other apps from accessing its audio (cf. screenshot blocking). As another defence, Android grants access for the camera and microphone to only one app at each point in time. However, this restriction does not prevent a malicious app from exploiting these resources in a side-channel attack if the victim app does not require access to these resources.

Several academic proposals [38, 44, 46, 51, 52, 56] rely on introducing noise, or reducing the sampling rate of information that might be exploited in a side-channel attack. However, it is non-trivial to determine the right volume of noise, or the appropriate sampling rate to guarantee that a side-channel attack will fail while the sensor output remains useful for legitimate apps in the background, e.g., a step counter. Other defences, such as randomizing the layout of a keyboard to make it difficult for a malicious app to figure out the key corresponding to the position where a key press has occurred [44, 54, 56, 66], have poor usability, and defend only against a particular type of side-channel attacks. Blocking access to resources that could be exploited in a side-channel attack while the victim app is displaying a keyboard, or is asking for a PIN/unlock pattern [4, 6, 38, 52, 54, 66], is similarly limited. Temporarily blocking other apps while the victim app is running [65] can severely limit the usefulness of legitimate background apps for long-running victim apps. Introducing permissions that protect access to sensor resources [38, 46, 62] is unlikely to work given the inadequacies of current explicit permission-granting approaches.

In this paper, we propose `AppVeto`, a generic approach that augments the current permission models and empowers apps against side-channel attacks on mobile platforms. AppVeto promotes applications self-defence, assuming app developers are aware when their app is handling security-critical information, and hence can communicate their *veto* needs to the OS so that other concurrent apps are disallowed from accessing selected resources that may leak private information. In particular, AppVeto enables a foreground app to override resource access rights of background apps at certain times, e.g., during password input. Through an app's meta-data such as the Android manifest file, it can inform AppVeto about its veto requirements while the app is visible on the display.

In particular, we enable the following veto powers to block any concurrently running apps from accessing: (i) resources that are well-known to be exploited for certain side-channel attacks as defined in AppVeto, (ii) resources selected by the app developer that may interfere with the app's specific security needs, and (iii) resources that are being used by the requesting app, i.e., allowing exclusive access rights.

We implemented a prototype for our framework on Android. In particular, we use the Xposed framework [60] and PLT hooking techniques [7], so that our prototype can be easily distributed, and security enthusiasts and researchers can install and test it on major Android distributions. Android allows access to resources from both its application and native frameworks (see Sec. 2). Compared to existing Android native hooking techniques, our design allows us to hook virtually any native framework APIs of Android; existing approaches cannot hook several native APIs, including, e.g., OpenSL ES (see Sec. 6.4). This is a major step forward compared to other Android security solutions (e.g., [30]). Resources that we currently enable vetoing include: all motion and environmental sensors [19], camera, and microphone—which have been exploited in real-world and proof-of-concept attacks, as we found in our survey of such attacks (see Sec. 2.3). To control resource access dynamically, we hook the Android application framework APIs and the native framework APIs. We currently do not modify the Android source. However, these hooks can be easily incorporated into the Android source for production distribution. Our code is available on GitHub.[1]

Contributions. Our contributions can be summarized as follows:

(1) We design and implement AppVeto, a new paradigm for mobile application *self-defence* to enable finer-grained resource allocation compared to current models. Apps can communicate their special security and privacy needs, if any, to the OS, which will then be enforced by AppVeto in a fair manner. Both permissioned and permission-less resources can be blocked, or exclusively accessed by a requesting app, while the app is in the foreground.

(2) We empower app developers to control resource access by other simultaneously running apps without explicit user decisions, making our approach developer-centric and *user-agnostic*. Developers can block all commonly exploited resources for side-channel information leakage during, e.g., sensitive user input or output, or they can selectively block a specific resource according to their needs. Enabling an app to benefit from AppVeto requires very minor modifications—only updating its Android manifest file, i.e., no source code needs to be modified. Similarly, other apps installed on the system can remain unchanged.

(3) We provide an open-source implementation of AppVeto, which can be easily distributed, deployed, tested and extended by the community, without replacing stock Android distributions. Our prototype can handle both Android application framework and the native framework.

(4) We evaluate the performance and efficacy of AppVeto by testing AppVeto-enabled apps against relevant known side-channel attacks. Based on our experimental results, AppVeto can indeed effectively prevent such attacks originating from sensor devices, camera and microphone; other resources can be easily incorporated. As AppVeto runs all the time along with other OS components, we also measured its overhead on the system itself and other apps. The measured CPU and memory overheads are low (e.g., 0.64% CPU overhead on a Pixel 3) and should not deter real-world deployment.

(5) We document a comprehensive exploration of available Android native framework hooking techniques, and propose a new mechanism that can virtually hook any current native API libraries (including OpenSL ES). Our techniques can be used in other privacy/security projects to effectively hook both the application and native frameworks.

Differences with the ACSAC version [43]. The major addition to AppVeto compared to our ACSAC paper [43] is the capability to hook both the Android application and native frameworks. This capability can enable much more comprehensive protection against side-channel attacks. We utilize new native code hooking techniques to address limitations of the existing state-of-the-art; no past technique could hook all Android native APIs, which we achieve by hooking any native library as soon as it is loaded in an app's memory. Our current design includes native counterparts for all the resources we considered earlier [43], mandating a significant design extension (Sec. 7). Our new techniques can also help improve privacy/security tools that rely on hooking Android APIs. We also enhance our evaluation of AppVeto against side-channel attacks, considering resource abuse from native

[1]https://github.com/tousifosman/app-veto

code, and performance impact due to native API hooking (Sec. 8). Furthermore, we test AppVeto with several popular apps using native code for resource access. We provide comprehensive details on low-level Android resource access mechanisms (Sec. 2, Appendix C, D), and an extensive discussion on Android native hooking techniques (Sec. 2, Appendix C), which may help researchers to understand/choose appropriate techniques for their work. Finally, our experiments on resource access in popular Android apps shed light on recent shifts in Android development and their impact on security analysis of these apps (Sec. 9).

2 BACKGROUND AND CHALLENGES

In this section, we present a few definitions that we use throughout the paper, and provide a brief overview of the Android architecture, resource end-points, and Android hooking techniques (see Appendix C.1 for a detailed overview of Android's architecture).

A *resource* is an end-point where an app can get access to information that is not provided by the app itself. An Android Activity is considered as a *foreground activity* as long as it has focus and is visible on the device's display. As soon as the activity loses focus, leaves the screen, or the screen is turned off, the activity loses its state as a foreground activity. We consider any app as a *foreground app* whenever any of its activities become a foreground activity. In contrast, a *background app* is any app that has no foreground activity on the device's display.

2.1 Android Native Binaries

Android is built on top of a Linux kernel, and therefore it supports both static and shared libraries [32], and uses the Executable and Linkable Format (ELF) for its libraries [15]. Android is supported on various CPU architectures such as ARM, ARM64, x86, and x86_64. A set of rules that specify how a binary executable exchanges information with services, like the kernel or a library, at run time is called an Application Binary Interface (ABI [32]). Combinations of different CPU architectures and instruction sets are defined as different ABIs. A build of Android primarily supports the ABI corresponding to the machine code used in its own system image. Optionally, Android can support other ABIs supported by the system image [15], e.g., ARM64 can execute binary code for the 32 bit ARM ABI. Generally, third-party apps ship their native executable as shared libraries with the app itself. When compiling native libraries, third-party apps generally compile the library for different ABIs to support the apps on different CPU architectures. During the run time, an app needs to choose its native library that is supported by the underlying platform.

2.2 Android Resources

Android is composed of various components starting from local files, audio/video sources to on-board sensors. We consider all these components as the resources under Android OS, which an app can access on demand (cf. Android's definition of resources [19]). Below we discuss resources available in the Android mobile platform that can lead to side-channel attacks.

2.2.1 Android Sensor Framework. Most Android phones come with various built-in sensors. These sensors generally interact with the surrounding physical world, and do not involve any (explicit) personal information. These sensors are categorized as motion sensors, environmental sensors, and position sensors. Apps access these sensors using the *sensor framework* [19]. Fig. 1-a depicts a simplified workflow of sensor access by an app. Apps use the Android SDK to access the Android sensor framework, and create an instance of a service called the *sensor service* [19]. Using this service, apps need to register a callback, called `SensorEventListener` [18], to receive sensor data. On request, the sensor service then registers the callback in the sensor framework, which accesses the Hardware Abstraction Layer (HAL) to link an app with the sensor. Whenever there is some new data available for the given registered sensor, the callback method is invoked, and the the app is notified with the sensor data.

Android's sensor HAL is the single client for accessing a sensor. The sensor framework performs multiplexing to allow multiple apps to access a sensor concurrently. To link the hardware with the Android OS, hardware manufacturers need to provide the actual implementation of HAL's C header file `sensors.h` [25], the device driver, and other intermediate components. However, as the hardware specific components are implemented by the hardware manufacturers, these components are device dependent. Hence, hooking or altering these components at run time or even at compile-time is impractical as the modifications will be device and hardware specific.

Native Sensor API. Android also allows access to sensors using native APIs. The Android NDK includes a header file (`sensor.h`) to access all the motion and environmental sensors from the native framework [16]. An Android app loads the native library `libsensors.so` [17] at run time, which has the native implementation of functions defined in `sensor.h`. Unlike the Java API, where an app receives sensor data by registering a callback, when using native APIs, an app needs to call the `ASensorEventQueue_getEvents` function [16] to retrieve the pending sensor events. The native API `ASensorEventQueue_hasEvents` checks if the sensor queue has any pending events to process, and returns 1 when there is a pending event or 0 otherwise. Fig. 1-b illustrates a simplified sensor access flow through native APIs.

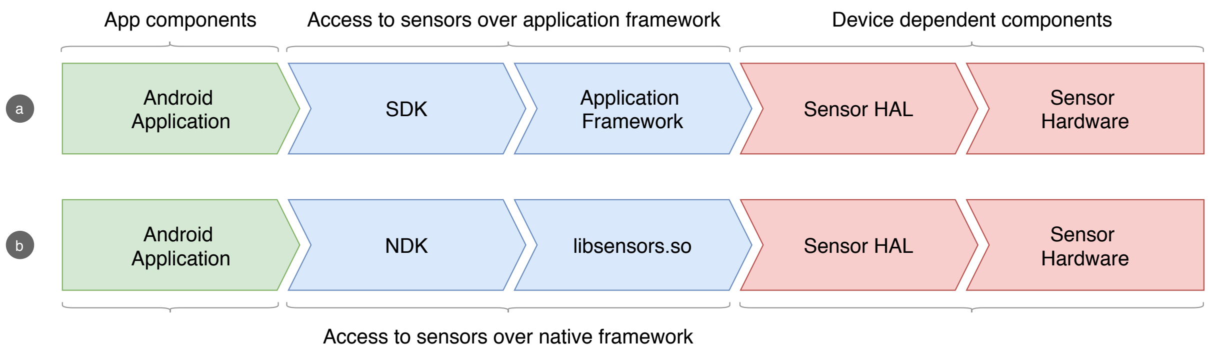


Fig. 1. Simplified access to sensors; (a) access from the application framework, (b) access from the native framework.

2.2.2 Android Camera API. Like the sensor framework, Android uses a similar architecture for its *camera API*. This API also has a HAL that creates an interface between the camera hardware and the Android application framework. The components of this HAL, and the device drivers are also implemented by the hardware manufacturers. However, in Android API level 21 (Android 5.0), a newer API—*camera APIv2*—was introduced, and the older *Camera APIv1* was deprecated. Currently, both of these APIs are available in the latest released distribution of the OS (Android 10). Furthermore, camera APIv1 is still used by many popular apps.

Unlike the sensor framework, multiple apps cannot use the camera hardware at the same time. Apps also cannot directly access the camera. Android runs a native service called the *media server*. A component of this service is called the *camera service*, which acts as an interface to the camera hardware. Fig. 2-a shows a simplified workflow for camera access. When an app uses camera APIv2 to access the camera, it first needs to create a session with the camera service. After having a session, the app can make a request to the camera service, to use the camera, and the service will capture the image for the app. In case of APIv2, apps cannot directly get the captured image. Rather, when making a request, apps must specify an Android `Surface` component, where the captured image will be placed. This surface component can point to a UI or a file. Then the surface is passed to the native camera service, which then writes the captured camera data on it. When an app wants to create a preview for the camera, it needs to bind a UI component with the surface, and when saving the file, the app needs to make a new request with a surface bound to a file.

On the other hand, camera APIv1 has less control and flexibility over the camera. However, with APIv1, apps can get the captured data bytes. Apps also need to make requests to the camera service when using APIv1, and register a callback to receive the captured data; in addition, apps can provide a `Surface` for preview. Nevertheless, APIv1 and APIv2's implementations are independent and reside in their own packages.

Native Camera API. An app needs to include the headers specific to the camera API to access the camera from the native [16], and the functions defined in these headers are implemented in the system library `libcamera2ndk.so`. Android loads this library in the memory space of an app when it accesses the camera using native APIs [17]. The native camera API follows a process similar to the Java camera APIv2, where an app first needs to create a session (`ACameraCaptureSession`) by providing a target output (`ANativeWindow`) [16]. Next, one of the following two functions can be used to capture a still image using the camera: `ACameraCaptureSession_capture` and `ACamera-CaptureSession_logicalCamera_capture`. An app can also use one of the `ACameraCaptureSession_set-RepeatingRequest` or `ACameraCaptureSession_logicalCamera_setRepeatingRequest` functions to capture repeating images for camera preview, video call, etc. It is also possible to call `ACameraCaptureSession_stop-Repeating` function to stop any ongoing repeating capture of images. Also, the `ACameraCaptureSession_close` function can be called by an app to close a session.

As discussed in this section, Android has two Java APIs corresponding to two different HALs to access its cameras. We hook the two Java APIs separately since there are many apps still using the legacy camera API. However, in the case of NDK, the deprecated (legacy) camera HAL is not supported by the native camera API. The native camera API is available as of Android 7.0. Perhaps to support older devices, many well-known apps, e.g., WhatsApp, Skype, and Line, use the Java legacy camera API to access the camera, although these apps use native APIs to access the microphone. Fig. 2-b depicts a simplified camera access from the native API.

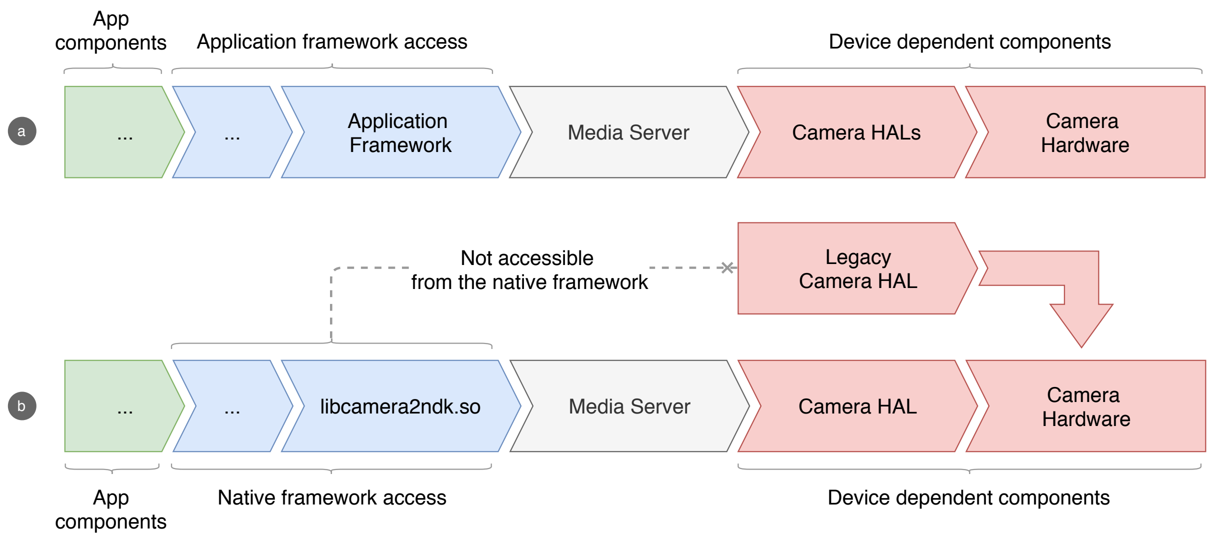


Fig. 2. Simplified access to camera: (a) access from the application framework; (b) access from the native framework.

2.2.3 Android Audio API. Android offers the `AudioRecord` class to allow apps to access microphones [18]. This class is one of the application interfaces to access the microphone. Similar to the camera, the media server is also responsible for microphone access. Apps can read audio data using the AudioRecord class. The app needs to start recording, and after that, the AudioRecord class allows the app to get audio data in three formats: `byte array`, `short array`, and `ByteBuffer` [41], using three function calls. It is strictly recommended in the Android developer's documentation that after recording, apps must release the audio resources. Otherwise, no other app will be able to access that audio resource.

Native Audio API. Fig. 3 shows an overview of microphone access from the native framework. Android has three sets of APIs, AAudio, OpenSL ES, and Oboe [11, 16], to allow access to the microphone from the native framework. AAudio and OpenSL ES are independent native libraries that allow an app to access the microphone from native binary independently. Whereas Oboe is a C++ wrapper library that uses the AAudio API when available, and otherwise, falls back to OpenSL ES [11]. Hence, we focus on the AAudio and OpenSL ES library to intercept accesses to the microphone from the native framework. Note that past Android projects such as XPrivacyLua [36], XPrivacy [35] (intended for privacy protection) were unable to hook native microphone access.

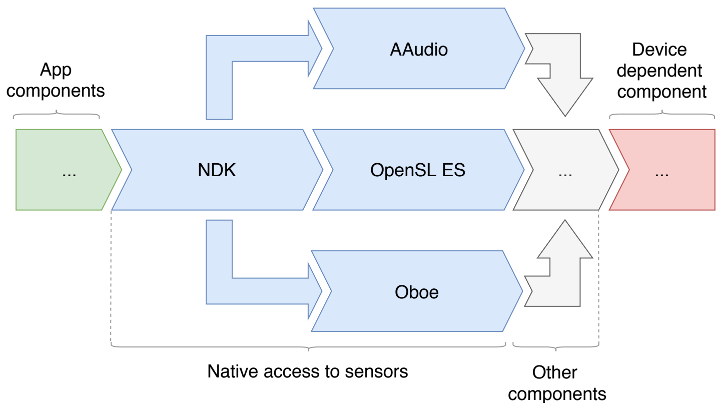


Fig. 3. Simplified access to microphone from the native framework.

Android ships with an implementation of OpenSL ES [33] specific to Android. Android apps need to include the `OpenSLES.h` header file to use this library (implemented in `libOpenSLES.so`). When an app uses the OpenSL ES API, this library is loaded in the memory space of the app [17]. Function `slCreateEngine` is the entry point for using this API. When an app calls this function it returns a structure of function pointers, and when these pointers are invoked, they return more pointer structures. All these function pointers are generated dynamically and an app needs to call a chain of function pointers to use microphone from the native framework (see also Appendix D for details). Because of dynamically generated function pointers and the chain of calls to these pointers, which can occur anytime in the application life cycle, traditional hooking techniques fail to intervene OpenSL ES's accesses to microphone. However, our design of hooks enables AppVeto to access the dynamically generated pointers before they are accessed by the app to be hooked and guarantee the interception of OpenSL ES (see also Sec. 6.4).

AAudio is a newer API for audio apps with a minimalist design and better support for high-performance audio with low latency. An app needs to include the `AAudio.h` header file to use this API (implemented in the library `libaaudio.so` [17]). This library is loaded in the memory space of an app when it uses the AAudio API to access the microphone from the native framework. When using this library, an app needs to call the function `AAudioStream_requestStart` to start recording, `AAudioStream_requestStop` to stop recording, and `AAudioStream_requestPause` to pause recording. Intercepting the calls to these functions can constrain the microphone access from the native framework using the AAudio library.

2.2.4 Android Media Recorder. Android offers another application interface, the `MediaRecorder` class, to access both the camera and microphone. This class is independent of other access methods and is used for recording audio and video. The workflow of this class is very similar to AudioRecord: apps can specify the camera and microphone source, and their recording needs (audio, video). Also, apps must specify a file location and output format for the recording. The MediaReorder next must be initialized to camera, microphone or both to allocate resources. Afterwards, the MediaRecorder must be started and it will then interact with the media server to start recording the specified media. Finally, the media server will process the record instruction and save the recording in the specified file. It is also strictly recommended that apps must release the resources after the recording is complete, or else these resources cannot be used by other apps.

2.3 Android Hooking

Bytecode Hooks. When it comes to hooking Android, techniques for its bytecode hooking are well explored. A prominent example is the Xposed framework [60], which extends the `/system/bin/app_process` executable [64], enabling it to load a JAR file at startup and, excluding few exceptions, hook any method of any class.[2] Xposed provides a set of APIs with which apps can hook Android runtime method calls. It allows modules to be developed that can be installed using Xposed management app. Users need to install the Xposed framework on their phone to be able to use any Xposed module. Xposed requires a rooted phone and as of Android 5.0 it needs to be installed from the recovery mode of Android [29]. At the time of this writing, official versions of Xposed are available for Android 4.0.3 to Android 8.1. (Unofficial versions are also available for Android 9 and 10 [37].) Xposed modules are invoked when the system boots up. These modules can register hooks for any Java methods of any app on the phone. Next, when an app is executed on the Android Runtime (ART), the Xposed framework intervenes the method call for the registered hooks. Xposed allows a module to entirely replace a method with a new one, call a different method after the original method call, or call a different method before invoking the hooked method. It currently can only hook Java method calls. When a hooked method is invoked, it is executed within the same process as the original method.

Native Hooks. There have been many attempts to hook Android native code (see, e.g., [30]), but it remains a difficult challenge. These techniques are very limited compared to Android bytecode hooking or machine code hooking on other platforms. Two well explored techniques for hooking Android's native code are PLT (Procedure Linkage Table) hooking and inline hooking [39]. We choose PLT hooking in our work and use the xHook[3] project as the foundation of our PLT hooks. We also explored inline hooking techniques discussed in Appendix C.3.

PLT hooking allows us to figure out the run time address of a function for a given shared library (see Appendix C.2 for details on PLT hooking). However, Address Space Layout Randomization (ASLR) [1] in Android prevents an app from knowing other apps' memory space and prevents the app from accessing the memory address of a target function. We solve this problem by making apps hook themselves. In particular, we use Xposed to make an app's bytecode invoke a native library of ours that then inserts hooks into the app's native code. Furthermore, in case of native hooking, if a program calls the function to be hooked before being hooked, then the hook becomes ineffective. This is because the app can receive a reference pointer—e.g., if a program calls the `slCreateEngine` function of the audio library and receives the structure of function pointers before the hook is placed, the program can keep using this structure and the hook becomes ineffective. On the other hand, if a program accesses the

[2]https://forum.xda-developers.com/xposed/xposed-installer-versions-changelog-t2714053
[3]https://github.com/iqiyi/xHook

relocation pointer before the hook is placed, then it can have the actual memory location of the target function, which also makes the hook inactive. We define this as *the time-of-hook problem*, and solve it by placing the hooks in such a way that the functions to be hooked will always be hooked before being called; see Sec. 6.

3 RELATED WORK: KNOWN ATTACKS AND SOLUTIONS

In this section, we first review relevant attacks exploiting Android resources, which is also necessary to understand our design choices (Sec. 5). We then discuss a few existing solutions.

3.1 Attacks Based on Resource Access

Recently, Ba et al. [5] introduced *AccelEve*, a state-of-the-art side-channel attack to eavesdrop on a phone's on-board speaker by using accelerometer. They show that the accelerometer on most smart phone covers the fundamental band of human speech. They use deep learning algorithms to reconstruct speech played on a phone's speaker from data recorded on accelerometer.

Shen et al. [50] analyzed the characteristics of Android's accelerometer and magnetometer sensors, and designed a system that can infer a user's touch input. They collected 32,400 keypresses from their studied participants on numeric and alphanumeric virtual keyboards. Then they used this data to train a machine learning model using SVM, KNN, Random Forest, and Neural Network. With this model, they could infer user input with an accuracy up to 83.9%. Aviv et al. [4] showed that in addition to the input taps, the swap gesture of Android pattern locks can be inferred from the accelerometer data. They used logistic regression, and combined it with Hidden Markov Models [31] to train their system.

Spreitzer [57] exploited a less obvious resource, the ambient light sensor of an Android smartphone, to infer a user's PIN. The author first observed that a minor change in the orientation of the phone results in a notable change in the data captured by the ambient light sensor. Next, this leakage in the sensor data was exploited to gain a significant success rate when guessing the user PIN. Logistic regression, discriminant analysis, and KNN were used to train data, and an accuracy of 65% was achieved with only five guesses.

Simon and Anderson [54] demonstrated a system named PIN Skimmer, using the video camera and microphone of a smartphone, to predict PINs from software keyboards. They observed movements from a video to detect the part of the screen that has been used while typing the PIN. They also recorded sound from the touch pad using a microphone, and combined this audio and video data to train their system. An accuracy of 30% was achieved with two guesses (50% accuracy for five guesses). Raguram et al. [47] also exploited the video camera, but from a different perspective. They found that it is possible to reconstruct the text typed on a virtual keyboard just by observing the reflection of the phone's screen (e.g., reflection on the victim's sunglasses). They demonstrated that even with a low-cost camera, this side-channel attack can be launched. They used image processing and a Bayesian framework for their attack. An accuracy of 92% was achieved for retrieving text from a victim's sunglasses. Hasan et al. [28] exploited the magnetic sensor to establish a hidden communication channel with other devices and exchange information without a user's consent.

3.2 Defences

Song et al. [56] proposed two defenses against motion-based keystroke inference attacks. They found that reducing the accuracy of the motion sensors can significantly reduce the accuracy of these attacks. They also observed that the majority of these attacks rely on the fixed layout of the virtual keyboard, and therefore, randomizing the layout can successfully prevent these attacks. However, the input time on such a randomized keyboard can increase by three times compared to a regular keyboard.

Shrestha et al. [51] introduced Slogger to defend against sensor-based keystroke inference attacks, which is similar to the solution proposed by Song et al. [56]. In contrast to reduced accuracy, Slogger injects personalized random noise to sensor data. Slogger also avoids customizing the OS source. It uses an app to take sample inputs from the user when launched and calculates some threshold values. It then injects random noise in between the range of pre-calculated thresholds in the accelerometer and gyroscope sensor data readings.

Demetriou et al. [8] presented a new security system called SEACAT, which extends the current security module of Android, SEAndroid [55]. They demonstrated several flaws of the existing permission model, and showed how an attacker can exploit these flaws to gain access to personal data. As a solution, they extended the Android OS and proposed a new policy management that can permanently bind external resources (i.e., smart accessories) with a specific app and can provide mutually exclusive access to those resources from the bounded app only.

Xu et al. [61] demonstrated several flaws in the implementation of the Android Bluetooth security mechanism, by showing that Bluetooth peripherals have the capability to change their device profile with the help of a malicious app running on the device. Then a malicious app can allow a Bluetooth peripheral to communicate with the Android OS without any user consent. They introduced a new policy management system in the Android OS as a solution.

SemaDroid [63] has been proposed as a privacy-aware sensor management framework. SemaDroid relies on users to monitor sensor usage, and manually control sensor access. Furthermore, SemaDroid is implemented by hooking the Android source code, as opposed to hooking the runtime system. SemaDroid essentially allows advanced users to put restrictions on resource misuse. On the other hand, AppVeto introduces a general purpose resource access policy by delegating the responsibility of decision making from users to app developers. Note that manual access management for privacy-concerning resources is available in the latest Android distribution. SemaDroid also allows users to manually define conditions, e.g., location, and time, on which defined constraints on resource access should be enforced. It can return mock data (user defined results), add noise to data, reduce the accuracy of data, or keep the sensor data unaltered for different resource access.

FlaskDroid [6] has been proposed as a mandatory access control architecture for Android. It can prevent sensors from being accessed while the phone is in a user-defined *security-sensitive state*, such as when the keyboard/PIN pad is displayed. However, keyboard input is not always sensitive. AppVeto lets an app developer decide when to block resource access.

App Guardian [65] temporarily blocks *suspicious* apps while a protected app is running. Suspicious apps are detected based on certain activities, e.g., a recording activity or frequent CPU usage. Blocking an app is rather heavy-handed. AppVeto selectively prevents resource access, and parts of a background app unrelated to the vetoed resources can continue to function properly.

PINPOINT [48] provides virtualized per-app sensor services to allow returning perturbed or no sensor information to certain apps. PINPOINT relies on the user to set up virtualized sensor services, which fails to protect users who are unaware of the possibility of side-channel attacks.AppVeto instead relies on app developers, who have the knowledge of sensitive components of their app, to protect the users.

AuDroid [45] detects and resolves *unsafe* information flows involving a phone's speaker or microphone. It prevents two different processes from accessing the speaker and microphone at the same time to prevent, e.g., an app with microphone access from learning the output of another app that is using the speaker (e.g., what is being played by a music player app). AppVeto is a more generic approach to control resources, and also allows the establishment of this type of exclusive access policies. For example, the developer of an app that outputs potentially sensitive information over the speaker can veto apps that want to access the microphone at the same time.

4 THREAT MODEL, GOALS AND ASSUMPTIONS

We currently implement AppVeto through the Xposed framework and PLT hooking. We assume Xposed modules are trusted and they have system level privilege. Ideally, we would want AppVeto to be incorporated in the OS source, enforced from within the OS itself, and thus need to trust only the OS.

Our AppVeto prototype can handle any app accessing vulnerable resources from both the application framework and the native framework. However, AppVeto currently does not handle apps that disable the flag `android:extractNativeLibs`.[4] When this flag is disabled, an app keeps uncompressed shared libraries in the app's APK files and loads these libraries from the APK files directly (see Sec 9). This can be effectively addressed, e.g., by modifying the Android source.

AppVeto treats all apps equally, and limits abuse by respecting veto powers of foreground apps alone (i.e., apps that are being used actively), restricting the period of denying access or exclusive access, and notifying users if the defined period is crossed. AppVeto-enabled apps distrust all other concurrent apps, and we expect developers to understand their apps' security and privacy requirements, and correctly specify their veto needs within the Android manifest file.

AppVeto enables a developer the following capabilities: (i) specify any or all sensors, camera and microphone (as well as other resources) for exclusive access or denying access to other apps; and (ii) specify certain classes of known side-channel attacks that an app needs protection from. AppVeto can be extended to cover any resource, when a new side-channel attack exploiting a new resource is discovered. With these new capabilities, we set the following goals for AppVeto:

(1) Input inference protection: prevent malicious apps from inferring sensitive information that a user enters into an AppVeto-protected app while running in the foreground—e.g. keystroke inference.
(2) Output inference protection: prevent malicious apps from inferring sensitive information output by an AppVeto-protected app while running in the foreground—e.g. eavesdropping speakers.

5 DESIGN OPTIONS

One possible approach to defend against inference attacks is to rely on detection and then removal of malicious apps (cf. traditional antivirus programs) [40]. However, this approach may fail against new variants of old malware and novel attacks.

Alternatively, concurrent apps can be temporarily suspended from running while the user is entering sensitive information into the to-be-protected app [65], and while the app is outputting sensitive information. However, this may affect the functionality of benign apps that legitimately run in the background (e.g., a music player). In addition, it is a heavy-weight approach that blocks even activities of concurrent apps that are not related to accessing resources, like a stopwatch app counting time.

We can also make static information exploited in inference attacks dynamic and inaccessible to apps. For example, randomize the keyboard layout to defend against input inference attacks [56]. However, with this approach, usability suffers, e.g., the time to enter information increases [56].

Additionally, we can perturb dynamic information exploited in inference attacks before delivering it to apps, e.g., reduce the sampling rate or add noise to a sensor [51], blur the video or audio stream delivered to an app, or introduce fake tap sounds [52]. However, finding the right amount of perturbance is non-trivial. Benign apps that legitimately run in the background may also infer wrong results (e.g., wrong step count) from the perturbed information, which in turn may confuse the user.

Finally, we can block dynamic information exploited in inference attacks from being delivered to apps [6]. Blocking access is arguably better than perturbing information since well-designed apps should be able to deal with lack of information. For example, Android delivers information from sensors via callback functions so apps should be able to deal with non-triggered callback functions. The drawback of this approach is that it may affect the functionality of benign apps that legitimately run in the background and access blocked information. We choose the last approach for our solution to limit the negative impact on apps; we also allow blocking only for a

[4]https://developer.android.com/guide/topics/manifest/application-element#extractNativeLibs; this flag is enabled by default in older versions of the Gradle open-source build system (before 3.6.0).

short configurable duration to avoid denial-of-service. Note that the to-be protected app suffers no usability or performance penalties.

In terms of when to trigger blocking of resources, one approach can be relying on the OS to infer potentially vulnerable situations—e.g., when the keyboard or a password box is prompted [6], or sensitive information such as a credit card number is displayed. When the keyboard is used for a while such as writing a long email, other apps may suffer. It is also difficult to distinguish between sensitive and non-sensitive information being output or input (from the OS perspective). One may also involve the user for explicit blocking requests, e.g., before entering a credit card number or accessing her banking app; this will entail both negative security and usability impacts. Instead, we choose to block when developers ask for it, assuming that developers of security-sensitive apps (like banking apps) should be familiar with their security requirements—at least more familiar than average users.

6 APPVETO: HOOKING RESOURCE APIS

In this section, we discuss how we identify Android resources and how we hook APIs to restrict access to these resources.

6.1 Overview and Challenges in Hooking

FFirst, we traverse the Android Open Source Project (AOSP) to understand the workflow of the resources of our interest, for both Android 5.0 (released in 2014) and 9.0 (2018). We rely on Java Reflections and hooks in the run time to learn the object structures in the ART. We then construct our method hooks and implement them in our framework. Developing these hooks in a backward-compatible manner is non-trivial as some data fields and system level method declarations are no longer the same in Android 9.0 compared to Android 5.0, even though the released APIs in the Application Framework remained unchanged. Additional effort may be needed to make AppVeto fully compatible with other Android versions.

We next investigate the native APIs mentioned in Sec. 2 in Android source to identify places we need to hook, how to handle the data, and how to put seamless restrictions on resources without breaking any app. First, we decompile various apps having native binaries using Android supported bytecode decompilers.[5] We then reverse engineering the corresponding native libraries to better understand the ecosystem.[6]

As Android prevents an app from accessing another app's memory space, a benign app thus cannot perform PLT hooking on other apps–i.e., a single controller app cannot perform PLT hooking on other apps. Regardless, an app can access its own memory space at run time. Hence, an app can perform PLT hooking on itself. Furthermore, an app's code is executed starting from its bytecode, and an app-specific native library must be loaded from the bytecode first. Later on, these native libraries can load other native libraries. Therefore, we implement our native hooks as a shared library and use the Xposed framework to load this library in all apps' memory. The idea of hooking native from bytecode has been explored before [27], but ensuring hooks on all native libraries of an app and hooking the relevant APIs, specially the native audio API, has not been achieved before.

Android allows an app to access the pseudo-file `/proc/self/maps`, which has the list of all loaded libraries of the current process, their file location, their starting memory addresses, etc. The starting address of a library is its base address. As discussed in Sec. 2.3, the relocation pointer of the target function is the sum of the base address and the offset of the function defined in the `.got` section. Our hook makes an app traverse through its list of the loaded libraries, get the location of the target library, find the offset address in the shared library, calculate the relocation address of the function to be hooked, and finally hook the function. One caveat here is that an app can load native libraries at any arbitrary time of its execution. So we trigger the hooks in such a way that all the target native APIs would always get hooked after the library is loaded and before being called from the app.

[5]We use ApkStudio (https://github.com/vaibhavpandeyvpz/apkstudio) and Bytecode Viewer (https://github.com/Konloch/bytecode-viewer).
[6]We use Ghidra (https://ghidra-sre.org/) and Radare2 (https://github.com/radareorg/radare2).

6.2 Sensor Hooks

Application Framework Sensor Hooks. As discussed in Sec. 2, the sensor service keeps track of the registered sensor listeners. This service acts as the primary interface to all sensors, and `SensorEventListener` is called for all sensor callbacks. However, this common callback method does not distinguish the individual callbacks from each sensor. From the AOSP, we found a system level class called `SystemSensorManager` [24] with an inner class called `SensorEventQueue`, which queues the `SensorEventListeners` calls and passes them to the native implementation. This class has a method called `dispatchSensorEvent`, which is invoked by the native code whenever there is some new data available for any sensor. This method receives an integer value called `handle`. The `SystemSensorManager` class has a data field called `mHandleToSensor`, which is a HashMap with `handle` as key and a `Sensor` [18] object as value. Using this map, `SensorEventQueue` can distinguish between callbacks of different sensors. Hence, we hook `dispatchSensorEvent`, and replace it with our method.

Native Framework Sensor Hooks. Android also allows to access on-board motion and environmental sensors using the NDK. Android apps continuously need to check the sensor event queue for new sensor data to access sensors from the native framework—refer to Sec. 2.2.1. An app needs to call the function `ASensorEventQueue-_hasEvents` to check avabality of new data and call the function `ASensorEventQueue_getEvents` to get the data from the event queue.

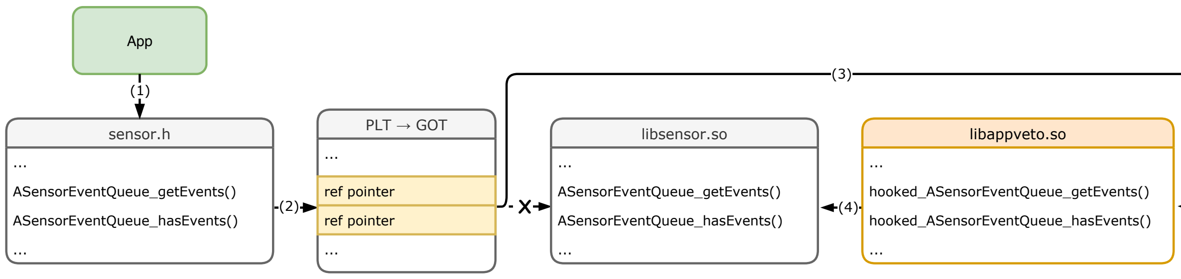

App
(1)
sensor.h
...
ASensorEventQueue_getEvents()
ASensorEventQueue_hasEvents()
...
(2)
PLT → GOT
...
ref pointer
ref pointer
...
X
(3)
libsensor.so
...
ASensorEventQueue_getEvents()
ASensorEventQueue_hasEvents()
...
(4)
libappveto.so
...
hooked_ASensorEventQueue_getEvents()
hooked_ASensorEventQueue_hasEvents()
...

Fig. 4. Hook flow of native sensor APIs.

To provide fine-grained control over resource access, we must intercept each individual sensor, and differentiate calls to `ASensorEventQueue_getEvents` for each sensor separately. One of the arguments passed while an app calls this function is the structure `ASensorEvent`. Unlike the Java API, this structure has a member `type` that corresponds to the sensor type. Hence, we change the relocation pointer of this function to a new function located in our framework's loaded library. The new hook function receives all the parameters of the original call and checks if access to the corresponding sensor is allowed. If there is no constraint on the sensor access, then we call the original `ASensorEventQueue_getEvents` and let it alter the arguments passed to hooked function; otherwise, we leave the arguments as it is and return the constant `PERMISSION_DENIED` [26]. We also hook the function `ASensorEventQueue_hasEvents`, and return `0` when there exists some constraint (otherwise we call the original function). Fig. 4 shows the flow diagram of hooks for native sensor APIs.

6.3 Camera Hooks

As discussed in Sec. 2, Android has two HALs to access the camera. Android supports multiple cameras and they can be accessed from the application framework and the native framework. Below, we discuss our hooking techniques on Android camera.

Application Framework Camera APIv2 Hooks. For the camera, the callback method of CaptureRequest [18] does not have the data, and preventing it from being invoked does not stop the media server from taking a picture. However, to access the camera using camera APIv2, an app must make a capture request. Cancelling this request prevents apps from accessing the camera. Apps must call CaptureRequest using the CameraCaptureSession [18] class of camera APIv2. This class has the following methods to make a capture request: (1) capture, (2) captureBurst, (3) captureBurstRequests, (4) captureSingleRequest, (5) setRepeatingRequest, (6) setRepeatingBurst, (7) setRepeatingBurstRequest, and (8) setSingleRepeatingRequest.

We hook all these methods and replace them with our own code. Four of these capture methods are used to make capture requests to make the camera take pictures repeatedly. A common use case for these methods is to display the camera view before capturing an image; they also enable a background app to repeatedly capture images without making a new capture request. Therefore, our framework needs to stop these repeating requests when a foreground app defines a constraint over camera access. CameraCaptureSession offers the abortCaptures method to abort any ongoing capture requests. We thus hook the constructor of the CameraCaptureSessionImpl (system level implementation of the abstract CameraCaptureSession class), and whenever a new object of this class is initialized, we store it in a HashSet for each app. We iterate through all the active sessions for all apps, and invoke the abortCaptures method to terminate existing capture requests when necessary.

Application Framework Camera APIv1 Hooks. With camera APIv1, Android apps capture images by calling the takePicture method of the Camera [18] class. Similar to APIv2, intercepting this method call and preventing it from being called can prevent an app from taking pictures. The Camera class has setPreviewDisplay and setPreviewTexture methods to display a preview, the startPreview method to start the preview, and the stopPreview to stop the camera preview.

Preview of this API can also be used to create videos or take still pictures [18]. Hence, we need to stop the preview for background apps, when a foreground app vetoes camera access. The Camera class has a method called open to create an instance of this class. We hook this method and create a HashSet of Camera instances for each app. Like APIv2, when a foreground app vetoes camera access, our framework invokes the stopPreview method for all Camera instances. When the veto on camera access is released, we restart the preview (via startPreview).

Native Framework Camera Hooks. Intercepting the functions calls referred in Sec. 2.2.2 (under "Native Camera API") can impose constraints on the camera access from the native framework. Hence, we hook the four functions to control the capture requests from the native framework. We hook these functions similar to the native sensor API hooks (see Sec. 2.3). When the current foreground activity imposes constraints on the camera, we return the error code ACAMERA_ERROR_PERMISSION_DENIED, and leave the arguments unchanged; otherwise, we call the original request function. In the case of a repeating request, we abort it by calling the ACameraCaptureSession_stopRepeating function. We also hook the ACameraCaptureSession_stopRepeating function to know when an app requests to stop the repeating capture of a session and the ACameraCaptureSession_close function to know when a session is closed. This information is required to manage the active sessions (see Sec. 7.5).

6.4 AudioRecord Hooks

Android allows to access the microphone from both the application framework and the native framework. AppVeto can handle microphone access for both as discussed below.

Application Framework Audio Hooks. We hook the AudioRecord [18] class to allow constraints on microphone access. With this class, apps need to invoke one of the overloaded read methods corresponding to audio data in a specified format (see Sec. 2.2.3). Unlike the case for the camera, AudioRecord has no continuous capture request. Hence, intercepting the read method is sufficient to prevent microphone access using

AudioRecord, and therefore we hook all overloaded methods for read. Apps must call the startRecording method to make the microphone start recording. We also hook this method so that apps are prevented from making the microphone from capturing audio when the foreground app vetoes such requests.

Native Framework Audio Hooks. Android offers three sets of default APIs to access the microphone from the native framework (see Sec. 2.2.3). The OpenSL ES is the legacy API for accessing the microphone using the native framework. A chain of function calls from the returned structure of the function slCreateEngine gives access to the microphone. We reverse-engineered the libOpenSLES.so library and our investigation shows that the library of our test OS (Android 8.0 for Google Pixel 3) does not have any named functions (symbols) in the library corresponding to these pointers. Moreover, a library named libwilhelm.so is required by the libOpenSLES.so library and this library has classes that are likely to be called behind the scenes, e.g., an AudioRecord class with start and stop member methods. Hence, the functions pointed to by the function pointers cannot be hooked as these functions are not exported to the app's native libraries. Also, the symbols corresponding to these functions in the symbol table of the native libraries are not known and can be device dependent, e.g., functions can have random symbols on each device. Hence, we can only hook the slCreateEngine function using the PLT hooking technique. However, if we can alter the result of this function call, this can sufficiently overcome the problem of hooking chained calls. However, the result is returned as a read-only memory block and if altered, even after changing the memory permission, the program breaks and stops execution.

We thus copy the memory block to a separate location, alter the result and return the altered result as read-only memory. As discussed in the Sec. 2.2.3, the result of slCreateEngine is a structure. We alter the result by assigning pointers to our hook functions to the members Realize, GetInterface, and Destroy of the structure. As a result, whenever an app calls these member function pointers, our hook functions get executed. In the hooked functions, we call the original function and similarly change its returned value. We continue this process until we reach the structure SLRecordItf where we hook the SetRecordState and GetRecordState functions. In this scenario, whenever an app calls any of these functions in the chain to access the microphone, our framework's hooked function gets executed first and allows our framework to control access to the microphone. Fig. 5 demonstrates our hooked call chain.

This approach still does not solve the timing of hook problem (see Sec. 2.3). Conventionally, the function slCreateEngine is called once and the returned structure is reused. Hence, if an app gets the returned structure before our hook, it may not call the slCreateEngine function after the hook and keep on using the unhooked structure. In contrast to the previous hooks, e.g. ASensorEventQueue_getEvents, even if the app calls the function to be hooked before the hook, the hooked function is guaranteed to be called after hooking the function. We solve this issue with our hook implementation, see Sec. 7.3 for details.

Hooking the AAudio API is straightforward. As discussed in Sec. 2.2.3, when accessing the microphone from the native framework using this API, the app needs to call the mentioned AAudioStream_requestStart function. We hook this functions, and when access is constrained, we return the constant value AAUDIO_ERROR_NO_SERVICE; otherwise, we call the original function. Similar to the native camera API, we also hook the AAudioStream_requestStop and AAudioStream_requestPause functions for managing the memory; see Section 7.5. As the Oboe API is a wrapper around the OpenSL ES and AAudio API, hooking these two API puts the same constraints on the Oboe API.

6.5 MediaRecorder Hooks

The MediaRecorder [18] class allows apps to record audio and video. When using this interface, apps must invoke its start method to start recording, which will start the media recording and save the data in the specified file path. MediaRecorder provides the pause and resume methods to pause and resume recording accordingly. Similar to the camera hooks, we hook the constructor of this class and maintain a HashMap of MediaRecorder instances for all running apps. If an app wants to use this class to record audio or video, the app must set an audio or video source accordingly. However, `MediaRecorder` provides no method to output if a video source is set. Thus, we hook the `setCamera` method (deprecated in API level 21) to know if the instance of the MediaRecorder records video, and store this information in the HashMap. When the foreground app puts constraints on camera or microphone access, we invoke the `pause` method of the instances of `MediaRecorder` that access audio/video resources. When the veto is released, our framework invokes the `resume` method to re-enable resource access for other apps.

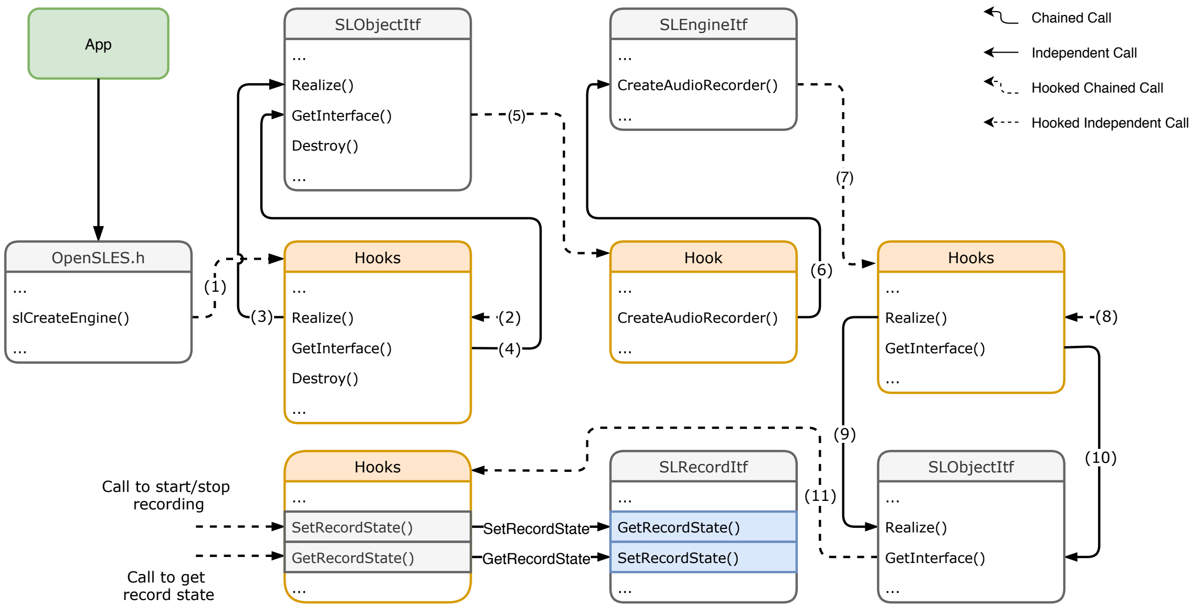


Fig. 5. Hook flow of OpenSL ES.

7 APPVETO: DESIGN, COMPONENTS, AND IMPLEMENTATION

In this section, we present the details of AppVeto design and our prototype implementation.

7.1 Design Overview

AppVeto enables a resource access policy that allows an app in the foreground to have privilege over resource access. When the foreground activity leaves the screen and becomes a background activity, the app's veto powers are removed. App developers can select an activity or a group of activities, and define what resource access should be prevented for background apps when the selected activity or activity group comes to the foreground. Developers can also simply specify what known side-channels should be prevented when the selected activity is in the foreground. Developers specify these constraints through Android application meta-data [19], i.e., the `AndroidManifest.xml` file. A constraint on a resource can be introduced by using any of the keys shown in Table 3 (Appendix A) as meta-data name and fully-qualified target activity class names concatenated with the pipeline character ("|") as the meta-data value. We have provided a code-snippet in Listing 1 (Appendix B). When an app is loaded, AppVeto checks the defined meta-data and constructs the veto needs to be applied on resource access, when the app is in the foreground.

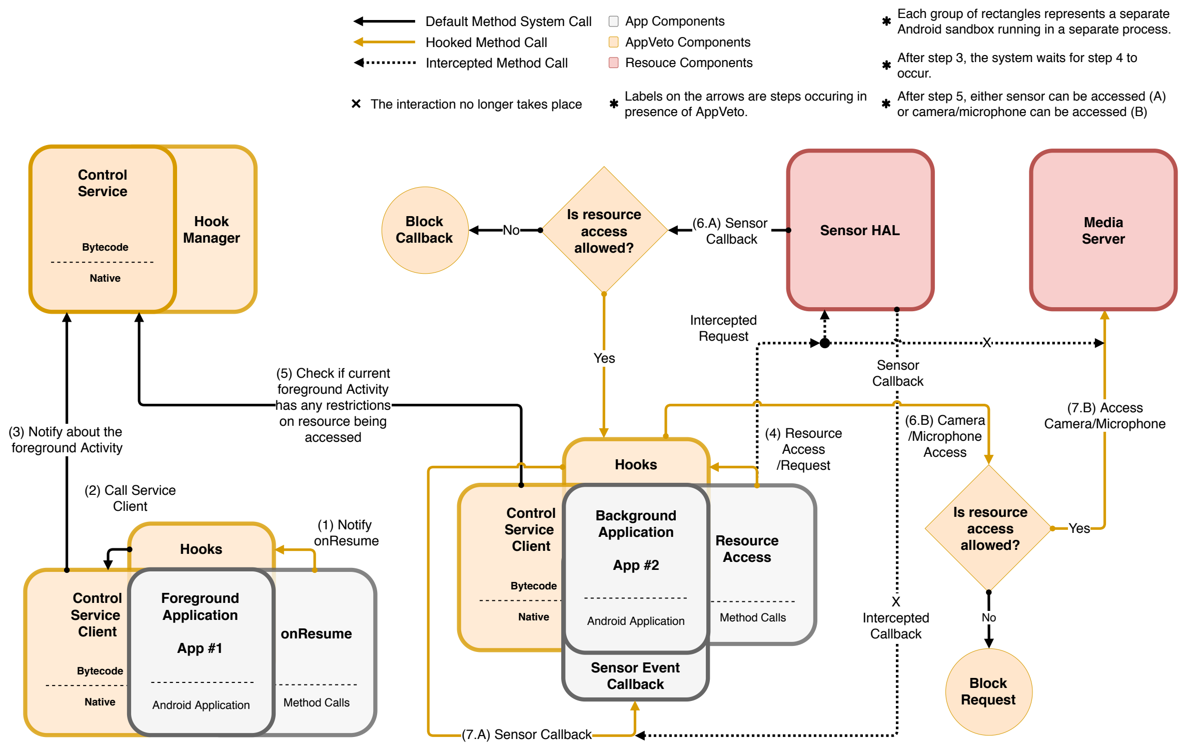

Default Method System Call
Hooked Method Call
Intercepted Method Call
The interaction no longer takes place
App Components
AppVeto Components
Resouce Components
Labels on the arrows are steps occuring in presence of AppVeto.
Each group of rectangles represents a separate Android sandbox running in a separate process.
After step 3, the system waits for step 4 to occur.
After step 5, either sensor can be accessed (A) or camera/microphone can be accessed (B)
Control Service
Bytecode
Native
Hook Manager
Block Callback
No
Is resource access allowed?
(6.A) Sensor Callback
Sensor HAL
Media Server
Intercepted Request
Sensor Callback
Yes
(5) Check if current foreground Activity has any restrictions on resource being accessed
(3) Notify about the foreground Activity
(2) Call Service Client
(1) Notify onResume
Hooks
Control Service Client
Foreground Application
App #1
onResume
Android Application
Method Calls
Background Application
App #2
Resource Access
Sensor Event Callback
(4) Resource Access /Request
(6.B) Camera /Microphone Access
(7.B) Access Camera/Microphone
Intercepted Callback
Block Request
(7.A) Sensor Callback

Fig. 6. AppVeto overview.

AppVeto enforces resource access policies for both native APIs and Java APIs. We implement AppVeto as an Xposed module, which allows our code to be easily integrated with the Android runtime (ART). However, Xposed is unable to hook the native Android binaries, which we address by incorporating PLT hooking techniques. Fig. 6 shows an overview of AppVeto. Below we detail the resource access and system calls that we hook to implement AppVeto. We use a Nexus 4 phone with Android 5.0 for our primary development and testing. We also use a Pixel 3 phone with Android 9.0 for evaluation (with EdXposed [37]).

Extendability is a major design goal for AppVeto, so that constraints on new resources can easily be incorporated as a new module to the framework without changing its existing components. In addition to restricting a specific resource, we also allow easy grouping of resources that are often exploited for a specific side-channel (e.g., several sensors and microphone can be used to infer password inputs). We add a few groups, but new groups can be easily defined. The components of AppVeto include a bytecode part to handle resource access from the application framework and a native counter part residing in a shared library to handle resource access from the native framework. These two counterparts communicate over JNI. Below, we detail the major components of AppVeto.

7.2 Meta-data Manager

The meta-data manager is responsible for defining meta-keys for different resources, and retrieving the declared meta-data from apps installed on the system. Table 3 (appendix A) lists the meta instructions we have defined in the current prototype. The meta-data manager is also responsible for mapping meta-keys with their associated resource access. Adding a new key is as simple as adding a new `enum` field and a string identifier in the meta-data manager. It also allows defining group meta keys that will prevent resource access for group of resources when specified in the Android manifest file. Defining a new group meta-key is also as simple as defining a new `enum` field, a string identifier, and previously defined meta-keys associated with certain resources.

7.3 Hook Manager

This is the entry point for our framework into the runtime of the Android OS. It allows intercepting Android function calls at run time, and augments the behavior of the OS without modifying the OS source directly. We must know which app is in the foreground and what is the current foreground activity. Every app window displayed on the screen is a subclass of the `Activity` class. All children of `Activity` inherit a method named `onResume`, which is called by the OS every time that activity appears on the screen and gains focus [19]. Also, whenever an activity leaves the screen or loses focus, the `onPause` method inherited from the `Activity` class is called [19]. Hence, the hook manager intercepts these two methods and injects our code before the original call. Whenever an app window changes, our injected methods are called, and AppVeto becomes aware of the current foreground app and its focused activity.

We must also intercept the resource access by all the apps to enforce vetoes. We create separate modules in the hook manager for hooking resource components, containing the methods that are to be injected in the hooked methods. The hook manager intercepts `dispatchSensorEvent` (see Sec. 6.2) for capturing the sensor callbacks. Whenever there is a call for this method from sensors, our injected methods are executed first. The injected methods check if the current foreground app has any veto on the corresponding sensor callback; if not, the injected methods invoke the original hooked methods. However, if a constraint is present on sensor access, then only the injected methods are executed.

For camera APIv2, we first hook all the capture request methods (see also Sec. 6.3). Similar to the sensors, an injected method checks for access restrictions; if there is no veto, the original hooked method is called, otherwise the `CameraAccessException` with parameter `CAMERA_DISABLED` is returned. The hook manager module receives a callback when the foreground activity changes. On that callback, if the responsible module finds that the current foreground activity has a veto on camera access, it will invoke the `abortCapture` method. We follow a similar approach for camera APIv1. We prevent calls to `takePicture`, and throw an `Exception` when camera access is disallowed. Furthermore, on a foreground activity change notification, the module responsible for the camera hooks will call the `stopPreview` and `startPreview` methods.

The hook manager uses a separate module for hooking the `read` methods in `AudioRecord` (see Sec. 6.4). When access is vetoed, the audio data is replaced with the `null` value, and the error code `ERROR_INVALID_OPERATION` is returned; similarly, calls to the `startRecording` method are also prevented and an `IllegalStateException` is thrown.

We also have a module for `MediaRecorder` that hooks the relevant methods (see Sec. 6.5). We hook the `start` method, and when the foreground app vetoes the camera or microphone access, the injected method prevents the original method from being called. We also prevent background apps from recording audio/video using the `MediaRecorder` API.

Native Hook Manager. We add hooks in the bytecode to initiate the PLT hooking. Adding a hook on the `onStart` and `onPause` callback of activities can intercept all but the OpenSL ES library. Because, if an app calls the `slCreateEngine` function sometime after starting the app but before putting the app in the background—e.g., the `libOpenSLES.so` library is loaded when tapping on the call button of a VOIP app for the first time after starting the app—then our hooks become ineffective. For the example scenario, when the app starts, the library has not been loaded yet so the hook is ineffective, and when the app goes to the background, the function has already been called once and so again the hook fails. As discussed in Sec. 2.3, an app's execution starts from its bytecode, and then the app can load native libraries from bytecode using the Java methods `System.loadLibrary` or System.load. However, these two methods cannot be hooked by the Xposed framework. We found that these methods internally call the Runtime.loadLibrary and Runtime.load methods, respectively, while loading a library. Hence, we hook these two methods and as soon as a native library is loaded from bytecode, our framework is notified and if our target functions exist in that library, we hook them.

However, a loaded native library can load other native libraries from the native framework using the function dlopen. Libraries loaded from the native framework will escape our bytecode hooks. Regardless, there must be at least one native library loaded from bytecode which then can load other libraries from the native framework. As just discussed, AppVeto can hook native libraries loaded from bytecode. So, we hook the dlopen function of all libraries loaded from the bytecode. Later on, when these libraries try to load other libraries from the native framework, our hooked method is called. Next, our hooked method calls the original dlopen and load the library but before giving the control back to the app, AppVeto hooks the dlopen and other target functions of the newly loaded library. This approach works on older Android OS (before 8.0).

On the Linker Namespace[7] mechanism on Android 8.0 prevents dlopen from loading libraries from arbitrary locations. The hooked dlopen function is defined in a library located in AppVeto's package directory. When this hooked dlopen tries to open a library located in other apps' package directory, the namespaces conflict, and the library fails to load. In Android 8.0 and above, a new function android_dlopen_ext[8] has been introduced that allows specifying the namespace while loading a library. However, there is no publicly accessible function to create a custom namespace. We found another function named android_create_namespace[9] can be accessed in the runtime to create a namespace that allows the hooked dlopen function to load a library located from our chosen locations. On Android 8.0 and above, we hook the android_dlopen_ext function as well, as it also allows an app to load a library from the native framework. For both dlopen and to android_create_namespace, we call the original android_create_namespace function with a custom namespace to load a library, and hook the loaded library before giving control to the app.

7.4 Control Service

Whenever the hook manager hooks a method, the injected method is not called immediately. Rather, the injected methods are called from the process of the hooked app (i.e., not from the hook manager process). As a result, with our injected methods, it is possible to know when an app is in the foreground, which activity of the app is in the foreground, and when the app is trying to access some specific resources only from the process of that activity. However, other apps in the background cannot get this information or the current restrictions being applied on resource access. Therefore, we develop a control service for all apps to communicate and stay informed about their present status. This service also decides what policy to apply for the current foreground activity, and makes the policy available for all other apps running on the system. Hence, this service requires Inter Process Communication (IPC) between processes. The Android Bound Service leverages the Binder API and uses the Android Interface Definition Language (AIDL) to provide IPC over application sandboxes. We create a two-way communication channel between the control service and an app, using two AIDL definitions: one for all apps to communicate with the control service to receive/provide necessary information, and the other AIDL for communicating with previously bounded apps and to notify them when the foreground activity changes.

Native Control Service. When apps access resources from the native framework, our native hooks intercepts these accesses. IPC and JNI calls are expensive, and hence we minimize these calls, and develop a native counterpart for Control Service. For the native API, sensor data is not delivered with a callback but an app explicitly checks for the availability of sensor data. In case of the application framework, we can prevent the sensor callback from

[7]https://source.android.com/devices/architecture/vndk/linker-namespace
[8]https://developer.android.com/ndk/reference/structandroid/dlextinfo
[9]https://android.googlesource.com/platform/art/+/refs/heads/master/libnativeloader/include/nativeloader/dlext_namespaces.h#86

occurring whereas for the native framework, the apps keep on checking for data. So to minimize the IPC and a sensor's access latency, the native counterpart of Control Service keeps a copy of the resource access policies, especially for sensors, in the native and bytecode separately. This reduces IPC calls and JNI calls.

7.5 Control Service Client

Our framework also offers a client component that allows the injected methods to communicate with the control service. This client also receives a notification from the control service when the foreground app changes. The client then delegates this notification to the hook manager (Sec. 7.3). The client enables communication between the sandboxed Android apps and the control service. The client module is passed into the injected methods and it becomes a part of the hooked apps when accessed by the injected methods. Injected methods then use this client to communicate with the control service to inform it about the app's status. Also, the injected methods use this client to query about the policy to be applied on resource access.

Native Service Client. The control server client has a native counterpart to handle the native hooks. A notification sent by the *control service* is received by the bytecode of the client. The client then makes the access constraints available to the native counterpart. Hence, the native only checks its copy of the access policies when the app is trying to access some resources.

In the case of the camera and microphone, resources can be accessed continuously (cf. audio and video recording) without continuous API calls. Hence, repeating or continuous resource access needs to be stopped or paused when access is restricted by the foreground activity. To achieve this, we keep a reference to all `ACameraCaptureSession` for the native camera API, `AAudioStream` for native AAudio API, and all results of `slCreateEngine` for the `OpenSL ES` library. Whenever access to any of these resources is constrained, we call the stop/pause function on the corresponding reference for all apps. As soon as the constraint is lifted, we call the start/resume function. Furthermore, if an app destroys SLObjectItf for the OpenSL ES library, we release the memory it allocated. For camera APIv2 and AAudio, we remove the reference when an app itself calls the event release function to release the resource. These steps help prevent memory leaks.

8 EVALUATION

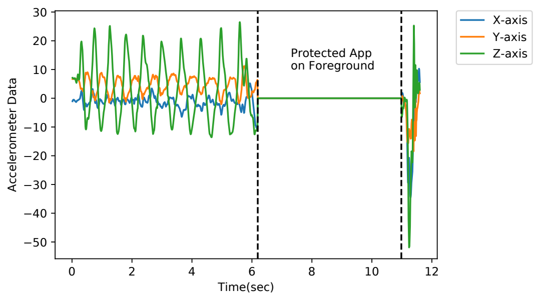
We tested the developed framework using both real-world and some of our experimental apps. We also measured the performance overhead of AppVeto on Google Nexus 4 (Quad–core 1.5 GHz, 2GB RAM) and Google Pixel 3 (Octa–core 4x2.5 GHz, 4GB RAM) devices.

8.1 Side-channel Evaluation

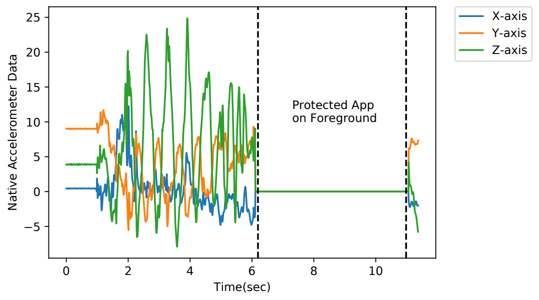
To perform our side-channel experiments, we developed a few test apps that use AppVeto to defend themselves, and some apps that use the application framework and the native framework separately to access resources from the background. Our test results show that the AppVeto framework successfully prevents background apps from accessing the sensor data when the protected app becomes a foreground app. Figure 7a shows the accelerometer data received by a test background app using the application framework. As indicated by the flat region in the figure, no sensor data is received by the background app when the protected app becomes a foreground app, showing that the background apps are indeed denied access to the data required for sensors-based side-channel attacks. We repeated the same experiment (Fig. 7b) but with a test app accessing the accelerometer with the native framework. The result was similar; no sensor data was received by the background app when the protected app became the foreground app.

We tested several popular apps, including Pedometer,[10] Facebook Messenger, WhatsApp, Line, Viber, and Skype, from the Android Play Store that access different resources, and evaluated the effectiveness of AppVeto on

[10] 10M+ installs, see: https://play.google.com/store/apps/details?id=cc.pacer.androidapp



(a) Access over application framework



(b) Access over native framework

Fig. 7. An example illustrating a background app denied from accessing the accelerometer by the foreground app.

these apps. We prevented the accelerometer while running the Pedometer app in the background. However, the app still counted steps properly, and reverse engineering of the app shows that it uses Android's built-in step counter.[11] As we do not prevent the OS from accessing the sensor data, the OS can successfully provide the step counting service. The built-in step counter is a part of Android's sensor framework and hence AppVeto allows constraints on the step counter as well. Although no known side-channel using the step counter exists yet, for validation of AppVeto, we block the step counter using `appveto_sensor_step_counter` (refer to Appendix A) meta declaration and prevent step counting. In this scenario, the OS still gets accelerometer data but we prevent the Pedometer from receiving step counts from the OS.

We tested AppVeto on popular video and audio calling apps to evaluate the microphone and camera interception. We reverse–engineered WhatsApp and found that it uses the legacy *Camera API* (see Sec. 2.2.2) from bytecode to access the camera and *OpenSL ES* (see Sec. 2.2.3) to access the microphone while making VOIP calls. When we veto the camera or microphone access, the video or audio of the call pauses, and when the veto is released the audio or video is resumed accordingly. Our experiments with Facebook Messenger show that it also uses the legacy *Camera API* and *Audio Record* API from bytecode to access the camera and microphone. Similar to WhatsApp, Facebook Messenger's video pauses in presence of a camera veto and the call resumes when the veto is removed. Even though Facebook Messenger does not use the native framework to access the camera or microphone, hooking its native libraries stops it from executing. Our further investigation shows that Facebook Messenger has mechanisms to detect PLT hooks; see Sec. 9 for further discussion. Reverse engineering of Skype, Viber, Line shows these apps use the legacy *Camera API* to access the camera and *OpenSL ES* to access the microphone. Similar to Facebook Messenger, they can detect the PLT hook and stop from executing (although in the presence of our hook Skype and Viber cannot access the microphone). In case of Line, it accesses native libraries directly from APK, which is not in our threat model (see Sec. 4 and Sec. 9).We also used different recording apps, including Audio Recorder by Sony, for testing the microphone veto. We counted from one to ten, and the numbers uttered during the microphone veto were missing in the recording.

8.2 Performance Evaluation

We measured AppVeto's overhead on CPU, memory usage, and latency on sensor data access, using Pixel 3 and Nexus 4 phones; see Tables 1 and 2 for a summary of our results. Interception for the camera and microphone is performed when the requests to access these resources are made. On the other hand, interception for sensors is done in the sensor data retrieval callbacks. Hence, the performance of sensor data access is more affected by

[11]https://developer.android.com/reference/android/hardware/Sensor#TYPE_STEP_COUNTER

AppVeto. Therefore, in our experimental setup, we first rebooted our test devices and ran a test app that retrieves accelerometer and gyroscope sensor data. Next, we measured the overall CPU usage of both test devices during an interval of 1 second for 60 seconds and took the average of these 60 samples. The experiment is repeated for 10 times with and without the AppVeto framework. We observe a CPU overhead of 0.64% for Pixel 3 and 5.57% for Nexus 4. It should be noted that a significant portion of the observed processing cost is due to the Xposed framework and runtime hooking. When integrated with an OS distribution, this overhead is expected to be much less.

	% CPU usage		Memory usage (GB)		Application framework sensor-access latency (ms)		Native framework sensor-access latency (ms)	
AppVeto	✗	✓	✗	✓	✗	✓	✗	✓
Average	14.63	15.27	1.90	1.96	10.0	10.1	9.977	9.985
Std. dev.	0.51	0.75	0.69	0.67	1.95	1.22	0.75	0.67
Overhead	0.64%		0.06 GB		0.1 ms		0.008 ms	

Table 1. Performance overhead for Pixel 3.

	% CPU usage		Memory usage (GB)		Application framework sensor-access latency (ms)		Native framework sensor-access latency (ms)	
AppVeto	✗	✓	✗	✓	✗	✓	✗	✓
Average	8.07	13.64	1.743	1.746	10.0	10.3	10.0	10.071
Std. dev.	1.32	1.44	0.027	0.014	1.72	0.82	1.35	0.58
Overhead	5.57%		0.3 GB		0.3 ms		0.071 ms	

Table 2. Performance overhead for Nexus 4.

We also monitored the memory usage during a 30 second interval, took 20 samples, and calculated the additional memory usage. The latency presented in Tables 1 and 2 was calculated by measuring the accelerometer access latency, which was done by using a test app that retrieves accelerometer data from the application framework and native framework and measures the time difference between each data retrieval point. The latency values for both Nexus 4 and Pixel 3 are small (see Tables 1 and 2).

9 LIMITATIONS

One of the limitations of AppVeto is when native libraries are directly loaded from APK files. Android allows the flag `android:extractNativeLibs` in its manifest file, which is enabled by default (until the recent release of Gradle 3.6.0). When disabled, instead of extracting native, shared libraries from APK files, an app keeps uncompressed shared libraries in its APK files and at run time loads the libraries directly from the APK files. This reduces the installed app size but significantly increases the APK file size. We currently do not handle native libraries accessed directly from APK files. This can be easily addressed if AppVeto is integrated with the OS. Also, during our side-channel evaluation (Sec. 8.1), we noticed that the microphone access in Line[12] resulted in unexpected outputs. Line could access the microphone even after preventing access from both the application and native frameworks. After decompilation, we confirmed that Line uses the native shared libraries to access the microphone but disables the flag `android:extractNativeLibs` and accesses these libraries directly from the APK at run time.

We also noticed that a few apps (e.g., Skype, Viber) stopped their execution after hooking their native libraries. We found that these apps use some detection techniques, e.g, hashing/signing a library's memory block after loading the library into the memory, which allow these apps to detect our runtime native hooks and stop execution. Hence, we can ensure to block any apps from accessing any restricted resource but an app can detect and stop executing entirely.

Also, AppVeto may be abused by malicious apps to deny legitimate apps access to Android resources, which might make them malfunction–e.g., fitness apps might miss count steps. We limit the possibility of a DoS attack to only when an app is in the foreground. To further mitigate this threat, we set a timeout on an app's veto powers (configurable by the OS/AppVeto distributors). Our study shows that in most cases resource access veto is required mostly on login or authentication forms where users stay for a short amount of time. Also, developers should use AppVeto only on activities that handle critical information that may be subjected to side-channel attacks. We are also experimenting with a negative reinforcement strategy, which will make apps pay some *price*, e.g., warning messages, notification warnings, and process throttling, to limit DoS possibility.

We have designed AppVeto such that it has minimal impact on other apps. Regardless, legitimate apps can malfunction, specially if the apps do not check a resource's availability before using it. We also broadcast a resource's availability state, which legitimate apps can receive to handle interruptions. We observed that the Facebook messenger app drops the call after receiving no input from the microphone for a while; apparently, this app drops the call if it receives zero bytes from the microphone for a defined amount of time.

We rely on developers to understand the security needs of their apps to benefit from AppVeto. However, many Android developers may have little grasp on security. On the other hand, many apps may not require the additional security through AppVeto. Also, configuring an app for AppVeto is similar to current permission settings in the Android manifest file, which we believe will help developers to easily incorporate veto powers in their apps.

Mobile OS vendors may also consider enhancing protections against side-channel attacks; cf. recent changes to sensor access in Android 9.0 [12]. If password input prompts are reliably detected, the OS itself can apply a veto on accessing side-channel-prone resources for all background apps, even if the foreground app requests no such restrictions.

10 CONCLUSION

We present AppVeto, a generic OS-level framework to enable finer-grained control on mobile device resources. Compared to existing runtime and install-time models, such enhanced access restrictions allow us to design a comprehensive defense against several side-channel attacks that exploit both permissioned (e.g., microphone) and permission-less (e.g., accelerometer) resources. We bring developers to the forefront of securing their apps against these stealthy but highly effective attacks, without burdening users with additional security-critical decisions. Our current implementation addresses both application framework and native code based resource abuses. However, we leave out a few other resources such as Bluetooth. We also leave out hooking of native libraries directly accessed from APKs. With more engineering efforts, more resources and hooking of these libraries can be added to our prototype. We also overview and document Android's native hooking techniques and present a technique to hook major Android APIs. We are making AppVeto available to app developers and security-enthusiasts, who can test and extend AppVeto as it is based on the Xposed framework and PLT hooking, i.e., no custom OS image is needed. We believe that the AppVeto approach is a step towards a more effective

[12]https://play.google.com/store/apps/details?id=jp.naver.line.android

permission model for mobile operating systems. Our native code hooking techniques may also help understand other apps with native binaries, and implement stricter privacy protection frameworks—e.g., [30, 39].

REFERENCES

[1] Chris Anley, John Heasman, Felix Lindner, and Gerardo Richarte. 2007. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (2 ed.). Wiley.

[2] Apple Inc. 2020. About privacy and Location Services in iOS and iPadOS. https://support.apple.com/en-us/HT203033.

[3] Apple Inc. 2020. Requesting Permission - App Architecture - iOS - Human Interface Guidelines - Apple Developer. https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/.

[4] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. 2012. Practicality of accelerometer side channels on smartphones. In *28th Annual Computer Security Applications Conference (ACSAC'12)*. Orlando, Florida, USA, 41–50.

[5] Zhongjie Ba, Tianhang Zheng, Xinyu Zhang, Zhan Qin, Baochun Li, Xue Liu, and Kui Ren. 2020. Learning-based Practical Smartphone Eavesdropping with Built-in Accelerometer. In *Network and Distributed System Security Symposium (NDSS'20)*. San Diego, CA, USA.

[6] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. Washington, D.C., USA, 131–146.

[7] caikelun. 2020. Android PLT hook overview. https://github.com/iqiyi/xHook/blob/master/docs/overview/android_plt_hook_overview.zh-CN.md (In Chinese).

[8] Soteris Demetriou, Zhou Xiaoyong, Muhammad Naveed, Yeonjoon Lee, Kan Yuan, XiaoFeng Wang, and Carl A Gunter. 2015. What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources. In *Network and Distributed System Security Symposium (NDSS'15)*. San Diego, CA, USA.

[9] Ele7enxxh. 2020. Android ARM Inline Hook. http://ele7enxxh.com/Android-Arm-Inline-Hook.html (In Chinese).

[10] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS'12)*. Washington, D.C., USA, 3:1–3:14.

[11] Google. 2020. Oboe. https://github.com/google/oboe

[12] Google Developers. 2019. Behavior changes: all apps. https://developer.android.com/about/versions/pie/android-9.0-changes-all

[13] Google Developers. 2019. HAL interface. https://source.android.com/devices/sensors/hal-interface.html

[14] Google Developers. 2019. Services overview. https://developer.android.com/guide/components/services.html#Foreground

[15] Google Developers. 2020. Android ABIs. https://developer.android.com/ndk/guides/abis

[16] Google Developers. 2020. Android NDK. https://developer.android.com/ndk/.

[17] Google Developers. 2020. Android NDK Native APIs. https://developer.android.com/ndk/guides/stable_apis.html.

[18] Google Developers. 2020. API reference. https://developer.android.com/reference.

[19] Google Developers. 2020. Developer Guides. https://developer.android.com/guide/.

[20] Google Developers. 2020. Permissions overview. https://developer.android.com/guide/topics/permissions/overview.

[21] Google Developers. 2020. Platform Architecture. https://developer.android.com/guide/platform/.

[22] Google Developers. 2020. Privacy changes in Android 10. https://developer.android.com/about/versions/10/privacy/changes#app-access-device-location.

[23] Google Developers. 2020. Privacy in Android 11. https://developer.android.com/preview/privacy.

[24] Googlesource.com. 2019. core/java/android/hardware/SystemSensorManager.java - platform/frameworks/base - Git at Google. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/java/android/hardware/SystemSensorManager.java#778

[25] Googlesource.com. 2019. include/hardware/sensors.h - platform/hardware/libhardware - Git at Google. https://android.googlesource.com/platform/hardware/libhardware/+/master/include/hardware/sensors.h

[26] Googlesource.com. 2020. Error.h. https://android.googlesource.com/platform/system/core/+/refs/heads/master/libutils/include/utils/Errors.h

[27] GToad. 2020. Android Native Hook tool practice. https://gtoad.github.io/2018/07/06/Android-Native-Hook-Practice/ (In Chinese).

[28] Ragib Hasan, Nitesh Saxena, Tzipora Haleviz, Shams Zawoad, and Dustin Rinehart. 2013. Sensing-enabled Channels for Hard-to-detect Command and Control of Mobile Devices. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. Hangzhou, China, 469–480.

[29] Andrew Hoog. 2011. Chapter 6 - Android forensic techniques. In *Android Forensics*, Andrew Hoog (Ed.). Syngress, Boston, 195–284. http://www.sciencedirect.com/science/article/pii/B9781597496513100068

[30] Hongwei Jiang, Kai Yang, Lianfang Wang, Jinbao Gao, and Sikang Hu. 2019. A Code Protection Scheme via Inline Hooking for Android Applications. In *Cyberspace Safety and Security*. Cham, 102–116.

[31] Dan Jurafsky and James H. Martin. 2009. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition* (2 ed.). Pearson Prentice Hall. 988 pages.
[32] Michael Kerrisk. 2010. *The Linux programming interface a Linux und UNIX system programming handbook.* No Starch Press, Inc.
[33] Khronos Group. 2020. OpenSL ES. https://www.khronos.org/opensles/
[34] John R. Levine. 1999. *Linkers & Loaders* (1st ed.). Morgan Kaufmann.
[35] M66B. 2020. XPrivacy. https://github.com/M66B/XPrivacy
[36] M66B. 2020. XPrivacyLua. https://github.com/M66B/XPrivacyLua
[37] MagiskRoot. 2020. Download Xposed for Android Pie 9.0. https://magiskroot.net/download-xposed-for-android-pie/
[38] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. Tapprints: Your Finger Taps Have Fingerprints. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12).* Ambleside, United Kingdom, 323–336.
[39] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. PatchDroid: Scalable Third-Party Security Patches for Android Devices. In *Proceedings of the 29th Annual Computer Security Applications Conference.* New Orleans, Louisiana, USA.
[40] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2019. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *ACM Trans. Priv. Secur.* 22, 2 (April 2019), 14:1–14:34.
[41] Oracle.com. 2020. ByteBuffer (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.
[42] Oracle.com. 2020. Java native interface specification. https://docs.oracle.com/en/java/javase/13/docs/specs/jni/intro.html
[43] Tousif Osman, Mohammad Mannan, Urs Hengartner, and Amr Youssef. 2019. AppVeto: Mobile Application Self-Defense through Resource Access Veto. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19).* New York, NY, USA, 366–377.
[44] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. 2012. ACCessory: Password Inference Using Accelerometers on Smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications (HotMobile'12).* San Diego, California, USA, 9:1–9:6.
[45] Giuseppe Petracca, Yuqiong Sun, Trent Jaeger, and Ahmad Atamli. 2015. AuDroid: Preventing Attacks on Audio Channels in Mobile Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15).* Los Angeles, CA, USA, 181–190.
[46] Dan Ping, Xin Sun, and Bing Mao. 2015. TextLogger: Inferring Longer Inputs on Touch Screen Using Motion Sensors. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec'15).* New York, USA, 24:1–24:12.
[47] Rahul Raguram, Andrew M. White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. 2011. iSpy: Automatic Reconstruction of Typed Input from Compromising Reflections. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11).* Chicago, Illinois, USA, 527–536.
[48] Paul Ratazzi, Ashok Bommisetti, Nian Ji, and Wenliang Du. 2019. PINPOINT: Efficient and Effective Resource Isolation for Mobile Security and Privacy. *CoRR* abs/1901.07732 (2019). http://arxiv.org/abs/1901.07732
[49] Gian Luca Scoccia, Stefano Ruberto, Ivano Malavolta, Marco Autili, and Paola Inverardi. 2018. An Investigation into Android Run-time Permissions from the End Users' Perspective. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft'18).* Gothenburg, Sweden, 45–55.
[50] Chao Shen, Shichao Pei, Zhenyu Yang, and Xiaohong Guan. 2015. Input extraction via motion-sensor behavior analysis on smartphones. *Computers and Security* 53 (2015), 143–155.
[51] Prakash Shrestha, Manar Mohamed, and Nitesh Saxena. 2016. Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec'16).* Darmstadt, Germany, 67–77.
[52] Ilia Shumailov, Laurent Simon, Jeff Yan, and Ross Anderson. 2019. Hearing your touch: A new acoustic side channel on smartphones. *CoRR* abs/1903.11137 (2019). http://arxiv.org/abs/1903.11137
[53] Amit Kumar Sikder, Giuseppe Petracca, Hidayet Aksu, Trent Jaeger, and A. Selcuk Uluagac. 2018. A Survey on Sensor-based Threats to Internet-of-Things (IoT) Devices and Applications. *CoRR* abs/1802.02041 (2018). http://arxiv.org/abs/1802.02041
[54] Laurent Simon and Ross Anderson. 2013. PIN Skimmer: Inferring PINs Through the Camera and Microphone. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices (SPSM'13).* Berlin, Germany, 67–78.
[55] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS'13).* San Diego, CA, USA.
[56] Yihang Song, Madhur Kukreti, Rahul Rawat, and Urs Hengartner. 2014. Two Novel Defenses against Motion-Based Keystroke Inference Attacks. In *IEEE Mobile Security Technologies Workshop (MoST'14).* San Jose, CA, USA.
[57] Raphael Spreitzer. 2014. PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM'14).* Scottsdale, AZ, USA, 51–62.
[58] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. 2018. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys Tutorials* 20, 1 (2018), 465–488.

[59] Don Turner. 2019. Android Developers Blog: Capturing Audio in Android Q. https://android-developers.googleblog.com/2019/07/capturing-audio-in-android-q.html

[60] XDA Developers. 2020. Xposed Framework Hub. https://www.xda-developers.com/xposed-framework-hub/

[61] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. 2019. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. In *Network and Distributed System Security Symposium (NDSS'19)*. San Diego, CA, USA.

[62] Zhi Xu, Kun Bai, and Sencun Zhu. 2012. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*. Tucson, Arizona, USA, 113–124.

[63] Zhi Xu and Sencun Zhu. 2015. SemaDroid: A Privacy-Aware Sensor Management Framework for Smartphones. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*. San Antonio, TX, USA, 61–72.

[64] Karim Yaghmour. 2013. *Embedded Android*. O'Reilly Media, Inc.

[65] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. 2015. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA, 915–930.

[66] Man Zhou, Qian Wang, Jingxiao Yang, Qi Li, Feng Xiao, Zhibo Wang, and Xiaofeng Chen. 2018. PatternListener: Cracking Android Pattern Lock Using Acoustic Signals. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. Toronto, Canada, 1775–1787.

A APPVETO KEYWORDS

Table 3 lists the meta keys supported by our system. These keys are used to define constraints on resource access. The key `appveto_sensor_all` blocks all sensors from the *Android Sensor Framework*, `appveto_inference_keystroke` blocks all resources that have been exploited for past keystroke inference side-channel attacks, and `appveto_rogue_communication` blocks the microphone and magnetic sensor, exploited by past work for rogue communication [28, 45].

Type	**Meta Key**	**Constraint**
Group	appveto_sensor_all	All Sensors
	appveto_inference_keystroke	Keystroke Inference
	appveto_rogue_communication	Rogue Communication Channel
Individual	appveto_sensor_magnetic_field	Magnetic Sensor Access
	appveto_sensor_accelerometer	Accelerometer Sensor Access
	appveto_sensor_significant_motion	Significant Motion Access
	appveto_sensor_gyroscope	Gyroscope Access
	appveto_sensor_light	Light Sensor Access
	appveto_sensor_proximity	Proximity Sensor Access
	appveto_sensor_gravity	Gravity Sensor Access
	appveto_sensor_pressure	Pressure Sensor Access
	appveto_sensor_temperature	Temperature Sensor Access
	appveto_sensor_humidity	Humidity Sensor Access
	appveto_sensor_step_detector	Step Detector Access
	appveto_sensor_step_counter	Step Counter Access
	appveto_sensor_heart_rate	Heart Rate Sensor Access
	appveto_camera	Camera Access
	appveto_mic	Microphone Access

Table 3. Configurable constraints/veto powers.

B CODE SNIPPET TO ADD A VETO

Listing 1 depicts a code-snippet that shows how to add metadata in the AndroidManifest.xml file to veto keystroke inference on `LoginActivity` and `RegisterActivity`.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ...>
        <application ...>
                ...
                <meta-data
                        android:name="appveto_inference_keystroke"
                        android:value="com.example.LoginActivity|com.example.RegisterActivity"/>
                ...
                <activity android:name=".LoginActivity" ...>
                        ...
                </activity>

                <activity android:name=".RegisterActivity" ...>
                        ...
                </activity>
                ...
        </application>
</manifest>
```

Listing 1. Code-snippet to add a constraint in AndroidManifest.xml.

C ANDROID OS AND HOOKING TECHNIQUES

In this section, we briefly summarize the main components of the Android architecture, outline Android native APIs, and describe two hooking techniques for the Android OS [21].

C.1 Android OS Stack and Native Libraries.

Android is built on top of a custom Linux kernel, which is the lowest level foundation of the platform; see Fig. 8—simplified from [21]. The rest of the platform takes advantage of the kernel's key functionalities, e.g., memory management, and security. Android has adopted a Hardware Abstraction Layer (HAL [13]) to compartmentalize the lower-level driver implementation from the higher-level system implementation. HAL operates on top of the Linux kernel and interfaces the hardware devices, such as camera, mic, and Bluetooth, with the upper level of the platform. This allows the development of the Android OS independent of the hardware device specification. On the other hand, it delegates the responsibility of device-specific interfacing to the device vendors. The component above the HAL is the Android Runtime (ART), which instantiates multiple instances of a virtual machine that can execute bytecodes designed for Android. Each app on Android runs inside its own instance of the ART.

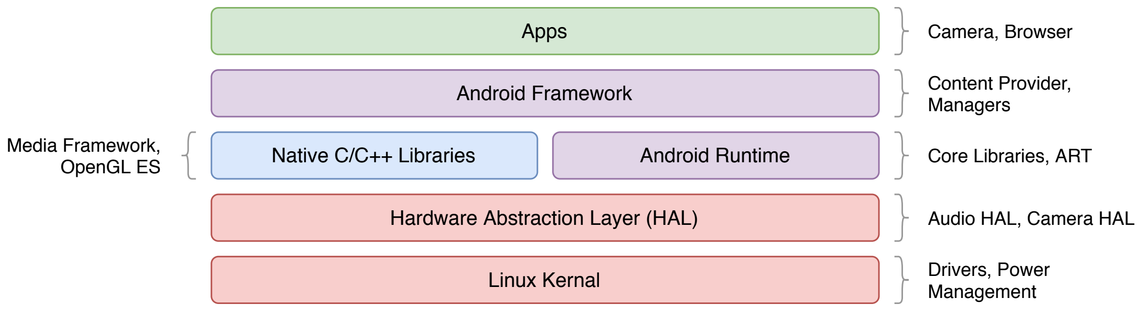


Fig. 8. Simplified platform architecture of Android, simplified from [21].

Several core Android functionalities must be directly executed at the native level. Thus, many components of the Android platform are made available as native libraries, e.g., OpenGL ES Android uses the Java Native Interface (JNI [42]) to access these native libraries from the ART. Android also has a Native Development Kit (NDK [16]) that allows third-party apps to develop native libraries using `C/C++`. In addition, Android has a separate set of APIs, called native APIs, to access some specific resources from the native libraries; e.g., the functions defined in the `sensor.h` header file allow native libraries to access Android's sensors using the native framework without executing any bytecode [16].

The Android platform separates apps from the bottom three layers with the Application Framework layer (refer to Fig. 8). This layer provides a set of Java APIs and allows the apps to access various components of Android—e.g., builtin UI, camera, MIC, etc. Among many critical tasks, this layer manages the app life-cycle and integrates Android's default user interface with apps.

C.2 Android PLT Hooking

Procedure Linkage Table (PLT) hooking is one of the oldest and most widely exploited hooking techniques for many OSes. As Android uses ELF scheme for its shared libraries, in this section of study, we will focus on hooking shared libraries of EFL scheme using PLT hooking techniques. Dynamic shared libraries of the ELF scheme, first introduced by Sun Microsystems in UNIX System V Release 4, can be viewed as a set of sections—including `.text` (read-only program code), `.data` (initialized data), `.bss` (uninitialized data), `.got` (*global offset table*), `.plt` (*procedure linkage table*), etc.[13]—interpreted by the program loader [34]. The Global Offset Table (GOT) is created by the linker and has pointers to all global data addressed by the executable file. Furthermore, each shared library has its own GOT [34]. At run time, the dynamic linker of the OS relocates and resolves all the GOT entries by using the PLT [1]. Any executable that uses a shared library has a PLT [34]. It performs the job of locating the symbol's address for the objects mapped in the memory. Fig. 9 shows the execution flow for a call to function `foo()` in a library `libBar.so`.

Android supports lazy binding where it resolves a library call only when it needs it[14]. When a program accesses an external function for the first time, the linker is called to resolve the actual address. On wards, execution can directly jump to the external function using the PLT. The `.got` section of the shared library file contains the offset address of its external functions. At run time, this offset of the library from the base address points to the relocation pointer that points to the beginning of the function's instructions loaded in the library's memory block. Hence, if at run time the base address of an executable is known, then its relocation pointers can be calculated and if altered, an arbitrary routine can be executed. The pseudo-file `/proc/<process id>/maps` lists all the libraries loaded

[13]http://www.sco.com/developers/gabi/latest/ch4.sheader.html
[14]Android linker source: https://android.googlesource.com/platform/bionic/+/master /linker

by a process, their virtual memory addresses (due to ASLR), file path, etc. Although, a process with root privilege can accesses this pseudo-file of any app but it cannot access any these memory addresses as these address are not memory absolute address. A process itself or the kernel level has the capability to access these memory address and anteater them. We solve this problem by making all app hook themselves and performed PLT hooking.

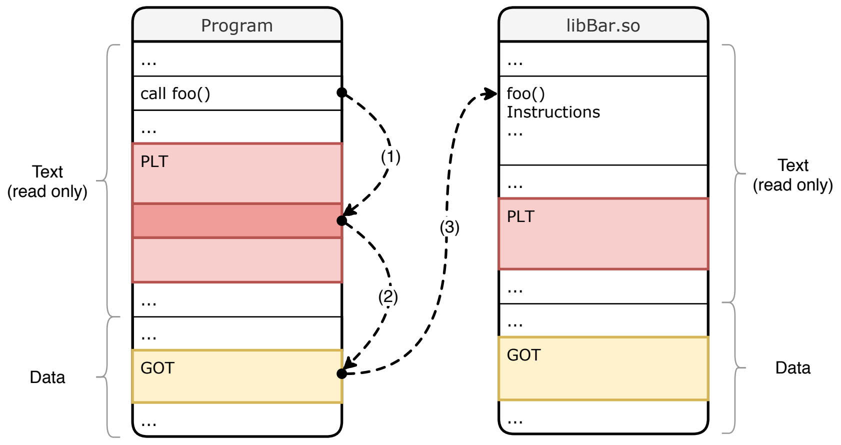


Fig. 9. Execution flow for call to external function **foo**, adapted from [34].

C.3 Android Inline Hooking

Inline hooking relies on the idea of replacing instruction at the beginning of a function to a jump instruction. When invoked, the function to be hooked (target function) then jumps to the injected hooked function. Finally, execution can return back to the target function after the execution of the hooked function. Inline hooking has fewer restrictions, and it virtually allows to hook any function. However, it depends on the instruction set of the phone's architecture and requires very fine-tuned book keeping. For low level details of inline hooking, the reader is referred to [9, 27].

D ANDROID OPENSL ES

OpenSL ES is an audio library for embedded devices [33]. Android's implementation of the library follows the standard specification and an app can use this library by including the `OpenSLES.h` header. As discussed in Sec. 2.2.3, the implementation of this library is located in the `libOpenSLES.so` shared library file and the `slCreateEngine` function is the entry point for this library. Calling this function returns a structure of function pointers (`SLObjectItf`). These pointers are uninitialized at first; the function pointed to by the `Realize` pointer in the structure has to be called first which initializes the rest of the pointers with the address of corresponding functions at run time. Calls to these function pointers also return a structure of function pointers that need to be initialized on run time. An app needs to call a specific chain of function pointers to access MIC from the native framework using the OpenSL ES library. At the end of this call chain, an app gets a reference to a structure (`SLRecordItf`) that has pointers to two functions: `SetRecordState` and `GetRecordSate`. Calling these function pointers allows an app to start/stop recording or get current state of recording. Fig. 10 shows the call chain of function pointers that an app needs to call in order to record using OpenSL ES.

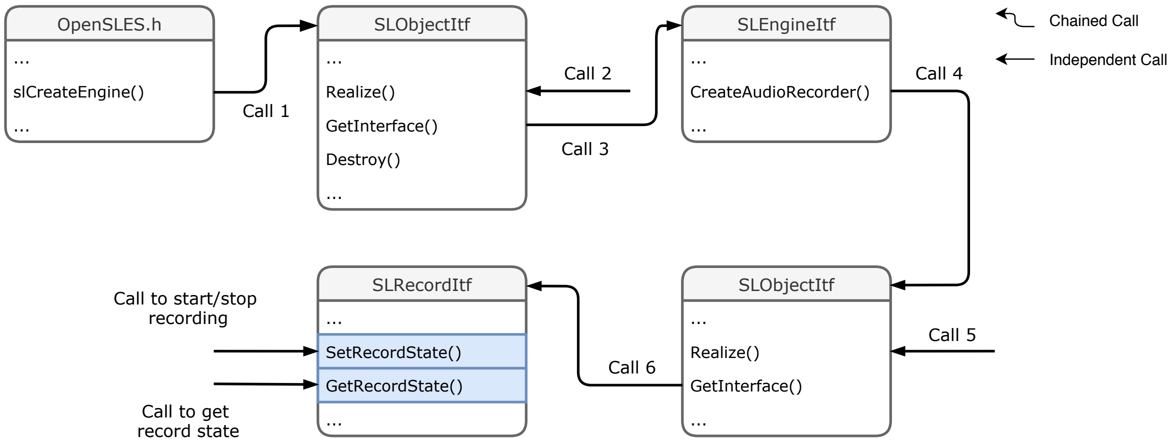


Fig. 10. Call chain of OpenSL ES function pointers to access the microphone from the native framework.