AppVeto:
Mobile Application Self-Defense through Resource Access Veto

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

ABSTRACT

Modern mobile operating systems such as Android and Apple iOS allow apps to access various system resources, with or without explicit user permission. Running multiple concurrent apps is also commonly supported, although the OS generally maintains strict separation between apps. However, an app can still get access to another app's private information, such as the user input, through numerous side-channels, mostly enabled by having access to permissioned or permission-less (sometimes even unrelated) resources, e.g., inferring keystroke and swipe gestures from a victim app via the accelerometer or gyroscope. Current mobile OSes do not empower an app to defend itself from such implicit interference from other apps; few exceptions exist such as blocking screenshot captures in Android. 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 to easily configure its 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, and delegate the responsibility of the resource access decision (for vetoing) from users to app developers. We implement AppVeto on Android using the Xposed framework, without changing Android APIs. Furthermore, we show that AppVeto imposes negligible overhead, while being effective against several well-known side-channel attacks.

CCS CONCEPTS

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

Contact email: t_osma@ciise.concordia.ca.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359839

KEYWORDS

Mobile applications security, mobile operating systems, permission management, side-channel attacks

1 INTRODUCTION

Modern smartphones are commonly equipped with various hardware sensors to learn and interact with the physical world (e.g., microphone, GPS, light sensor, and accelerometer). Smartphones also have access to personal and security-sensitive user information such as the contact list, photos, and passwords. Accessing these sensors/resources and user information by third-party apps is controlled by the mobile platform operating systems (OSes), and sometimes access is granted only with explicit user approval at the install-time of the app or during its runtime (e.g., see [1, 16]). 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. Permission models have been studied in detail, and unsurprisingly found to be inadequate in terms of protecting users' privacy and security [5, 36]. Even for runtime permissions, once a permission is granted, an app can use it until the app is uninstalled. 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 [40, 45, 48], as well as resources that require explicit user consent [34, 41]. Current user-approval based permission models cannot tackle these stealthy but highly-effective attacks; even if the user is asked for permission while accessing these resources, it would be infeasible for a typical user to understand the possibility of such side-channel attacks.

Vendors of operating systems have started to 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 if they are running in the background [9]. To access these resources from the background, apps need to launch a special component, a foreground service [20]. Users are notified of the presence of this service with an icon. However, if the user does not notice this icon, then the user will remain unaware of the resources accessed by the foreground service. The AudioPlaybackCapture API in Android 10 will allow an app to capture audio from another app [46]. Similar to screenshot blocking, this API also has defences against accessing sensitive audio through opt-out methods with which an app can prevent other apps from accessing its audio. 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.

It has also been suggested to introduce noise or reduce the sampling rate of information that might be exploited in a side-channel attack before providing the sensor data to an app [28, 31, 33, 38, 39, 43]. However, it is difficult to determine the right amount of noise or sampling rate that will guarantee that a side-channel attack will fail while keeping the sensor output useful for apps that legitimately access the output in the background, e.g., a step counter. Adding too much noise will lead to a legitimate app making a wrong conclusion and, in turn, misleading the user. 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 taken place [31, 41, 43, 52], 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 asks for a PIN/unlock pattern [2, 3, 28, 39, 41, 52], is similarly limited. Temporarily blocking other apps while the victim app is running [51] can severely limit the usefulness of legitimate background apps for long-running victim apps. Introducing permissions that protect access to sensor resources [28, 33, 49] is unlikely to work given the inadequacies of current explicit permission-granting approaches.

We propose AppVeto, a generic approach that augments the current permission models and empowers apps against side-channel attacks on mobile platforms. AppVeto promotes application self-defence, assuming app developers are aware when their application 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 have implemented a prototype for our framework on Android. In particular, we have used the Xposed framework [47] to develop an AppVeto prototype, so that it can be easily distributed, and security enthusiasts and researchers can install and test it on major Android distributions. Resources that we currently enable vetoing include: all built-in 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. To control resource access dynamically through the Xposed framework, we hook the Android application 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 mobile application self-defence framework that complements important limitations of existing permission models. It enables finer-grained resource allocation compared to current models. The approach is developer-centric and user-agnostic—i.e., AppVeto empowers app developers to control resource access by other simultaneously running apps without explicit user decisions. 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.

(2) AppVeto enables app developers to comprehensively protect their apps against known and anticipated side-channel attacks. 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). AppVeto thus promotes a new paradigm for application *self-defence*. Both permissioned and permission-less resources can be blocked, or exclusively accessed by a requesting app, while the app is in the foreground.

(3) Our current AppVeto prototype has been implemented on Android using the Xposed framework, without modifying the OS source code. Our implementation is open-source, which can be easily distributed, deployed, tested and extended by the community, without replacing stock Android distributions. 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.

(4) We evaluated 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., 0.43% CPU overhead on a Pixel 3) and should not deter real-world deployment.

2 BACKGROUND

In this section, we present a few definitions that we use throughout the paper, and provide a brief overview of Android resource access and the Xposed framework.

A *resource* is an end-point where an application can get access to information that is not provided by the application itself. Accessing these end-points to get relevant information is termed as accessing a resource. 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

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

turned off, the activity loses its state as a foreground activity. We consider any application as a *foreground application* whenever any of its activity becomes a foreground activity. Similarly, a *background application* is any application that has no foreground activity on the device’s display.

2.1 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 [7]). Below we discuss resources available in Android mobile platform that can lead to side-channel attacks.

2.1.1 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. Applications access these sensors using the *sensor framework* [19]. Fig. 1 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 [13]) 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.

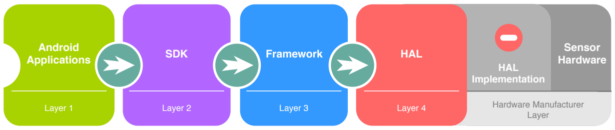

Figure 1: Simplified workflow of an Android app’s sensor access.

Android’s 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 `sensors.h` [23], the device driver, and other intermediate components. This abstraction makes the Android OS implementation independent of these hardware device specifications. 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.

2.1.2 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 9.0). Furthermore, camera APIv1 is still used by many popular apps.

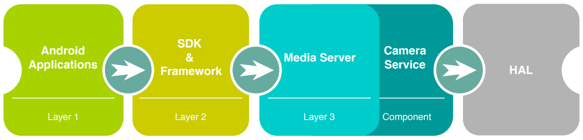

Figure 2: Simplified workflow of camera APIs (v1 and v2).

Unlike the sensor framework, multiple applications 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 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 application. 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 crate 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 bounded to a file.

On the other hand, the 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.

2.1.3 Android Audio Record. Android offers the `AudioRecord` class to allow apps to access microphones [8]. 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 application 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` [30], 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.

2.1.4 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.

2.2 The Xposed Framework

Xposed [47] provides a set of APIs with which other applications can hook Android run-time method calls. It allows modules to be developed that can be installed using Xposed management applications. Users need to install the Xposed framework on their phone to be able to use any Xposed module. The Xposed requires a rooted phone and the latest version of it needs to be installed from the recovery mode of Android [25]. Official versions of Xposed are available for Android 5.0 to Android 8.1 (unofficial versions are also available for Android 9 [27]). 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.

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

Shen et al. [37] 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. [2] 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 [26] to train their system. Spreitzer [44] 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 [41] 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. [34] 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. [24] 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. [43] 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. [38] introduced Slogger to defend against sensor-based keystroke inference attacks, which is similar to the solution proposed by Song et al. [43]. 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. [4] presented a new security system called SEACAT, which extends the current security module of Android, SEAndroid [42]. 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 application and can provide mutually exclusive access to those resources from the bounded application only.

Xu et al. [48] 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 application 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 [50] 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 run-time 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 [3] 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 [51] 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 [35] 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 to protect their sensitive apps.

AuDroid [32] 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, which is assumed to be trusted. 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. We are currently limited to hooking an app's Java code only due to our use of Xposed, which, on the other hand, enables easy deployment and testing. However, native code can be easily addressed e.g., by modifying the Android source.

AppVeto treats all apps equally and fairly, 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, the following goals can be achieved:

1. Prevent malicious apps from inferring sensitive information that a user enters into a to-be protected (victim) app.
2. Prevent malicious apps from inferring sensitive information output by a to-be protected app.

A to-be protected app is an app that is protected with AppVeto while running in the foreground.

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) [29]. 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 [51], and while the app is outputting sensitive information. However, this may affect 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 [43]. However, with this approach, usability suffers, e.g., the time to enter information increases [43].

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 [38], blur the video or audio stream delivered to an app, or introduce fake tap sounds [39]. 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 [3]. 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 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 [3], 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 DESIGN AND IMPLEMENTATION

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

Overview. AppVeto enables a resource access policy that allows an application 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 applications 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 [6, 15], 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 the 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. We implemented AppVeto as an Xposed module, which allows our code to be easily integrated with the Android runtime (ART). Fig. 3 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 [27]).

6.1 Method Hooks

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 Android source. 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 commonly available Android versions.

6.1.1 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` [22] 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` [17] object as value. Using this map, `SensorEventQueue` can distinguish between callbacks of different sensors. Hence, we hook `dispatchSensorEvent`, and replace it with our method.

6.1.2 Camera APIv2 Hooks.

For the camera, the callback method of `CaptureRequest` [12] 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` [11] 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.

6.1.3 Camera APIv1 Hooks.

With camera APIv1, Android apps capture images by calling the `takePicture` method of the `Camera` [10] class. Similar to APIv2, intercepting this method call and preventing it from being called can prevent an application from taking pictures. The `Camera` class has `setPreviewDisplay` and `setPreviewTexture` methods to display a preview, the

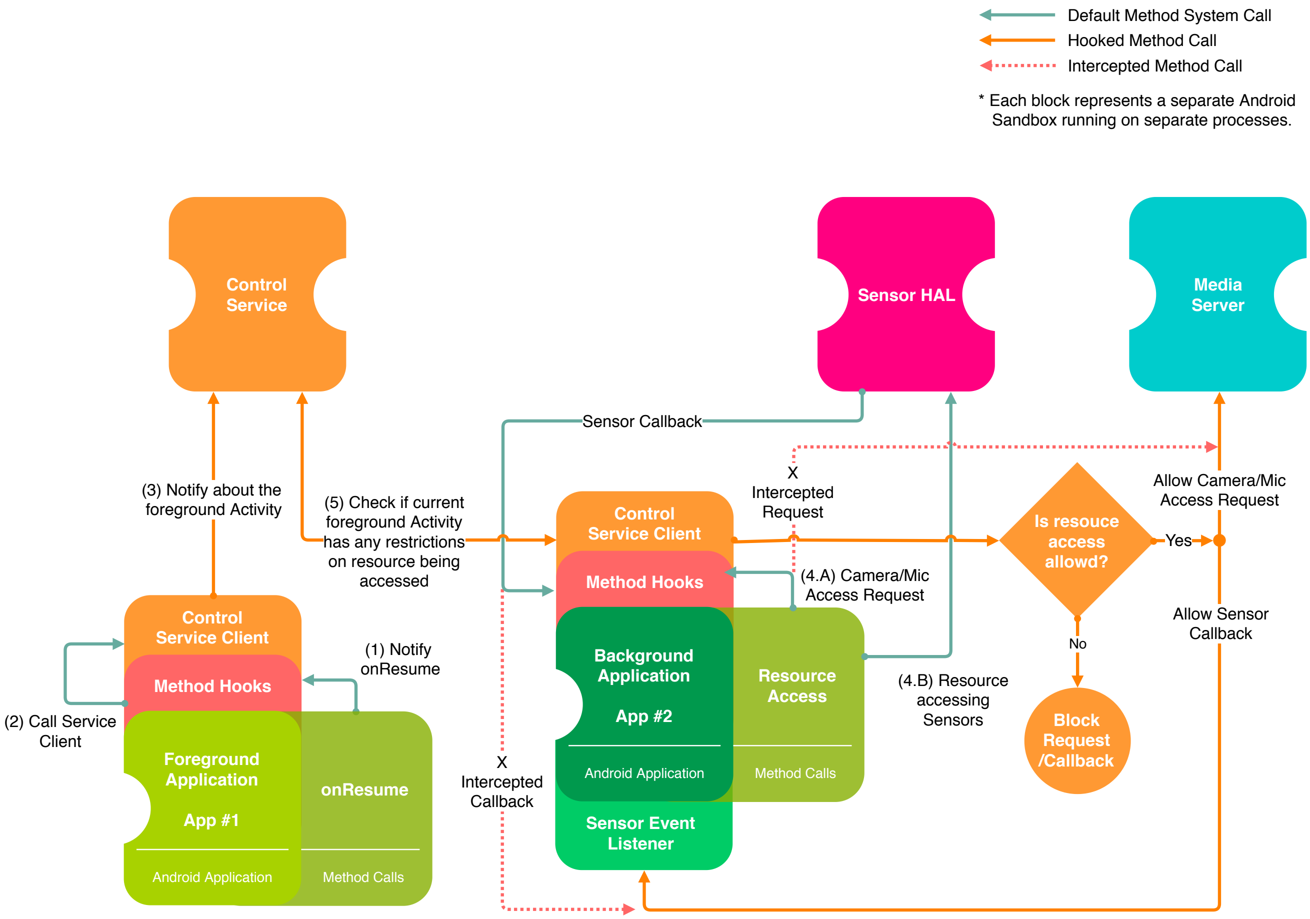

Figure 3: AppVeto overview.

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 [10]. 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).

6.1.4 AudioRecord Hooks. We hook the AudioRecord [8] 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. 2.1.3). 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.

6.1.5 MediaRecorder Hooks. The MediaRecorder [14] 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.

6.2 AppVeto Components

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 few groups, but new groups can be easily defined. Below, we detail the major components of AppVeto.

6.2.1 *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 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.

6.2.2 *Hook Manager.*

This is the entry point for our framework into the run-time of the Android OS. It allows intercepting Android function calls on the 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 [21]. Also, whenever an activity leaves the screen or loses focus, the `onPause` method inherited from the `Activity` class is called [21]. 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 application and its focused activity.

We also must intercept the resource access by all the apps to enforce vetoes. We create separate modules in the 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. 6.1.1) 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. 6.1.2). 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 (see Sec. 6.1.2). We also 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 (see Sec. 6.1.3).

The hook manager uses a separate module for hooking the `read` methods in `AudioRecord` (see Sec. 6.1.4). 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. 6.1.5). 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.

6.2.3 *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 application (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 notify them when the foreground activity changes.

6.2.4 *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. 6.2.2). 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 applications 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.

7 EVALUATION

We tested the developed framework using both real-world and test applications. We also evaluated its performance overhead on Nexus 4 (Quad–core 1.5 GHz, 2GB RAM) and Google Pixel 3 (Octa–core 4x2.5 GHz, 4GB RAM) devices.

7.1 Side-channel Evaluation

To perform our side-channel experiments, we developed a few test applications that use AppVeto to defend themselves, and some apps that access resources from the background. Our test results show that the AppVeto framework successfully prevents background applications from receiving sensor data when the protected application becomes a foreground application. Figure 4 shows the accelerometer data received by the background application. As indicated by the flat region in the figure, no sensor data was received by the background application when the protected application became a foreground application, showing that the background apps are indeed denied access to the data required for sensors-based side-channel attacks.

We also developed a few apps that veto camera access, microphone access, and both. We evaluated these apps with our test background apps, as well as real-world apps (a few popular apps from Google Play). We recorded audio and video with built-in system apps and, as expected, recording and image capture were interrupted. We also evaluated AppVeto while making video calls on Facebook messenger, Skype, and WhatsApp. When the camera veto was present, AppVeto successfully prevented the camera access, as imposed by the foreground app. The video display of the call was paused during the camera veto and then the video resumed when the veto was released. We have 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. We have also tested the microphone veto for well-known apps for audio calling. In general, the results were as expected but there were a few exceptions due to specific app implementations (e.g., Facebook messenger), or the use of the Native Development Kit (see Sec. 8). We have also tested and verified AppVeto by combining camera and microphone access vetoes. We have tested AppVeto with a few step counting apps, such as Pedometer by Pacer Health; a possible use case in this scenario is to log in to an app while walking. Our preliminary experiment shows that while walking, it takes about 30 seconds for the login, and about 40 steps will be missed.

7.2 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 AppVeto. Therefore, in our experimental setup, we first rebooted our test devices and ran a test application 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 0.43% for Pixel 3 and 5.28% 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.

We also monitored the memory usage during a 5 second interval, took 10 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 application that retrieves accelerometer data and measures the time difference between each data retrieval point. The latency values for both Nexus 4 and Pixel 3 are small (0.2ms and 0.1ms, respectively).

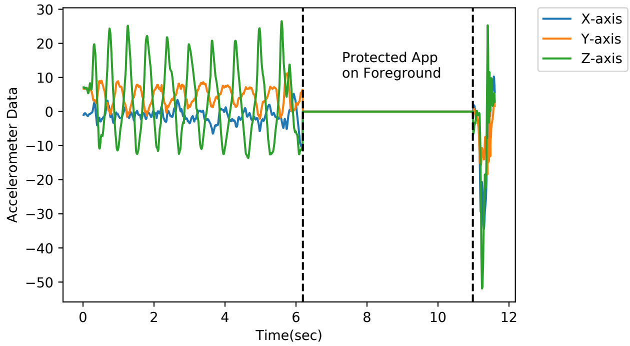

Figure 4: An example illustrating a background application denied from accessing the accelerometer by the foreground application.

8 LIMITATIONS

Our current prototype does not consider Android native libraries and the Android Native Development Kit (NDK). The Android NDK allows access to sensors and other resources without using the Android application framework APIs. During our side-channel evaluation (Sec. 7.1), we noticed that the microphone access in Skype resulted in unexpected outputs. Skype could access the microphone even after preventing all the callbacks. By decompiling Skype, we confirmed that it uses the NDK for accessing the microphone. Current versions of the Xposed framework cannot hook native binary libraries. However, this limitation can be easily addressed if AppVeto is incorporated in the OS source.

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.,

	% CPU usage		Memory usage (KB)		Sensor-access latency (ms)	
	Without AppVeto	With AppVeto	Without AppVeto	With AppVeto	Without AppVeto	With AppVeto
Average	2.71	3.14	3479527.2	3528691.6	10.0	10.1
Std. dev.	0.073	0.86	39691.61	384.63	1.95	1.22
Overhead	0.426%		49164.4 KB		0.1 ms	

Table 1: Performance overhead for Pixel 3.

	% CPU usage		Memory usage (KB)		Sensor-access latency (ms)	
	Without AppVeto	With AppVeto	Without AppVeto	With AppVeto	Without AppVeto	With AppVeto
Average	9.19	14.47	1827821.6	1831245.2	10.0	10.3
Std. dev.	1.17	1.49	2303.18	2231.96	1.72	0.82
Overhead	5.28%		3423.6 KB		0.2 ms	

Table 2: Performance overhead for Nexus 4.

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 [9]. 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.

9 CONCLUSION

We introduce 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 more security-critical decisions. Our current implementation does not address native code based resource abuse; we also leave out a few other resources such as Bluetooth. With more engineering efforts, these limitations can be addressed. 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, i.e., no custom OS image is needed. We believe that the AppVeto approach is a step towards a more effective permission model for mobile operating systems.

ACKNOWLEDGEMENT

This work was partly supported by an NSERC Discovery grant (second author). We thank the anonymous ACSAC 2019 reviewers for their insightful comments and suggestions. We also thank the members of Concordia's Madiba Security Research Group.

REFERENCES

[1] Apple Inc. 2019. Requesting Permission - App Architecture - iOS - Human Interface Guidelines - Apple Developer. https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Google Developers. 2019. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro

[7] Google Developers. 2019. App resources overview. https://developer.android.com/guide/topics/resources/providing-resources.

[8] Google Developers. 2019. AudioRecord. https://developer.android.com/reference/android/media/AudioRecord.html.

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

[10] Google Developers. 2019. Camera. https://developer.android.com/reference/android/hardware/Camera.html

[11] Google Developers. 2019. CameraCaptureSession. https://developer.android.com/reference/android/hardware/camera2/CameraCaptureSession

[12] Google Developers. 2019. CaptureRequest. https://developer.android.com/reference/android/hardware/camera2/CaptureRequest

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

[14] Google Developers. 2019. MediaRecorder. https://developer.android.com/reference/android/media/MediaRecorder

[15] Google Developers. 2019. <meta-data>. https://developer.android.com/guide/topics/manifest/meta-data-element

[16] Google Developers. 2019. Permissions overview. https://developer.android.com/guide/topics/permissions/overview.

[17] Google Developers. 2019. Sensor. https://developer.android.com/reference/android/hardware/Sensor.html

[18] Google Developers. 2019. SensorEventListener. https://developer.android.com/reference/kotlin/android/hardware/SensorEventListener

[19] Google Developers. 2019. Sensors Overview. https://developer.android.com/guide/topics/sensors/sensors_overview.

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

[21] Google Developers. 2019. Understand the Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle

[22] 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

[23] 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

[24] 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.

[25] 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

[26] 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.

[27] MagiskRoot. 2019. Download Xposed for Android Pie 9.0. https://magiskroot.net/download-xposed-for-android-pie/

[28] 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.

[29] 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.

[30] Oracle Corporation. 2019. ByteBuffer (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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

[36] 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.

[37] 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.

[38] 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.

[39] 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

[40] 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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] XDA Developers. 2019. Xposed Framework Hub. https://www.xda-developers.com/xposed-framework-hub/

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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 rouge communication [24, 32].

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 inferance on `LoginActivity and RegisterActivity.`

```
<?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