The Android Platform Security Model*

RENÉ MAYRHOFER, Google and Johannes Kepler University Linz
JEFFREY VANDER STOEP, Google
CHAD BRUBAKER, Google
NICK KRALEVICH, Google

Android is the most widely deployed end-user focused operating system. With its growing set of use cases encompassing communication, navigation, media consumption, entertainment, finance, health, and access to sensors, actuators, cameras, or microphones, its underlying security model needs to address a host of practical threats in a wide variety of scenarios while being useful to non-security experts. The model needs to strike a difficult balance between security, privacy, and usability for end users, assurances for app developers, and system performance under tight hardware constraints. While many of the underlying design principles have implicitly informed the overall system architecture, access control mechanisms, and mitigation techniques, the Android security model has previously not been formally published. This paper aims to both document the abstract model and discuss its implications. Based on a definition of the threat model and Android ecosystem context in which it operates, we analyze how the different security measures in past and current Android implementations work together to mitigate these threats. There are some special cases in applying the security model, and we discuss such deliberate deviations from the abstract model.

CCS Concepts: • **Security and privacy** → **Software and application security**; **Domain-specific security and privacy architectures**; **Operating systems security**; • **Human-centered computing** → **Ubiquitous and mobile devices**.

Additional Key Words and Phrases: Android, security, operating system, informal model

1 INTRODUCTION

Android is, at the time of this writing, the most widely deployed end-user operating system. With more than 2.5 billion monthly active devices [7] and a general trend towards mobile use of Internet services, Android is now the most common interface for global users to interact with digital services. Across different form factors (including e.g. phones, tablets, wearables, TV, Internet-of-Things, automobiles, and more special-use categories) there is a vast – and still growing – range of use cases from communication, media consumption, and entertainment to finance, health, and physical sensors/actuators. Many of these applications are increasingly security and privacy critical, and Android as an OS needs to provide sufficient and appropriate assurances to users as well as developers.

To balance the different (and sometimes conflicting) needs and wishes of users, application developers, content producers, service providers, and employers, Android is fundamentally based on a multi-party consent[1] model: *an action should only happen if all involved parties consent to it.* If any party does not consent, the safe-by-default choice is for that action to be blocked. This is different to the security models that more traditional operating systems implement, which are focused on user access control and do not explicitly consider other stakeholders.

While the multi-party model has implicitly informed architecture and design of the Android platform from the beginning, it has been refined and extended based on experience gathered from

*Last updated in December 2020 based on Android 11 as released.

[1]Throughout the paper, the term 'consent' is used to refer to various technical methods of declaring or enforcing a party's intent, rather than the legal requirement or standard found in many privacy legal regimes around the world.

Authors' addresses: René Mayrhofer, Google and Johannes Kepler University Linz, rmayrhofer@google.com; Jeffrey Vander Stoep, Google, jeffv@google.com; Chad Brubaker, Google, cbrubaker@google.com; Nick Kralevich, Google, nnk@google.com.

past releases. This paper aims to both document the Android security model and determine its implications in the context of ecosystem constraints and historical developments. Specifically, we make the following contributions:

(1) We motivate and for the first time define the Android security model based on security principles and the wider context in which Android operates. Note that the core multi-party consent model described and analyzed in this paper has been implicitly informing Android security mechanisms since the earliest versions, and we therefore systematize knowledge that has, in parts, existed before, but that was not formally published so far.
(2) We define the threat model and how the security model addresses it and discuss implications as well as necessary special case handling.
(3) We explain how AOSP (Android Open Source Project, the reference implementation of the Android platform) enforces the security model based on multiple interacting security measures on different layers.
(4) We identify currently open gaps and potential for future improvement of this implementation.

Android as a platform. This paper focuses on security and privacy measures in the Android platform itself, i.e. code running on user devices that is part of AOSP. Within the scope of this paper, we define the *platform* as the set of AOSP components that together form an Android system passing the Compatibility Test Suite (CTS). While some parts of the platform may be customized or proprietary for different vendors, AOSP provides reference implementations for nearly all components, including the e.g. Linux kernel[2], Trusty as an ARM TEE[3], or `libavb` for boot loader side verified boot[4] that are sufficient to run a fully functional Android system on reference development hardware[5]. Note that Google Mobile Services (GMS), including Google Play Services (also referred to as `GmsCore`), Google Play Store, Google Search, Chrome, and other standard apps are sometimes considered part of the platform, as they provide dependencies for common services such as location estimation or cloud push messaging. Android devices that are certified to support GMS are publicly listed[6]. While replacements for these components exist (including an independent, minimal open source version called `microG`[7]), they may not be complete or behave differently. Concerning the security model described in this paper, we do not consider GMS to be part of the platform, as they are also subject to the security policy defined and enforced by AOSP components.

In terms of higher-level security measures, there are services complementary to those implemented in AOSP in the form of Google Play Protect (GPP) scanning applications submitted to Google Play and on-device (Verify Apps or Safe Browsing as opt-in services) as well as Google Play policy and other legal frameworks. These are out of scope of the current paper, but are covered by related work [16, 51, 78, 130]. However, we explicitly point out one policy change in Google Play with potentially significant positive effects for security: Play now requires that new apps and app updates target a recent Android API level, which will allow Android to deprecate and remove APIs known to be abused or that have had security issues in the past [63].

Structure. In the following, we will first introduce the ecosystem context and threat analysis that are the basis of the Android security model (Section 2). Then, we define the central security model (Section 3) and its implementation in the form of OS architecture and enforcement mechanisms on different OS layers (Section 4). Note that all implementation specific sections refer to Android 11 at

[2] https://android.googlesource.com/kernel/common/
[3] https://android.googlesource.com/trusty/vendor/google/aosp/
[4] https://android.googlesource.com/platform/external/avb/
[5] https://source.android.com/setup/build/devices
[6] https://storage.googleapis.com/play_public/supported_devices.html
[7] https://github.com/microg/android_packages_apps_GmsCore/wiki

the time of its initial release unless mentioned otherwise (cf. [43] for relevant changes in Android 10 and [114] for changes in Android 9). We will refer to earlier Android version numbers instead of their code names: 4.1–4.3 (Jelly Bean), 4.4 (KitKat), 5.x (Lollipop), 6.x (Marshmallow), 7.x (Nougat), 8.x (Oreo), and 9.x (Pie). All tables are based on an analysis of security relevant changes to the whole AOSP code base between Android releases 4.x and 11 (inclusive), spanning about 10 years of code evolution. Finally, we discuss special cases (Section 5) and related work in terms of other security models (Section 6).

2 ANDROID BACKGROUND

Before introducing the security model, we explain the context in which it needs to operate, both in terms of ecosystem requirements and the resulting threat model.

2.1 Ecosystem context

Some of the design decisions need to be put in context of the larger *ecosystem*, which does not exist in isolation. A successful ecosystem is one where all parties benefit when it grows, but also requires a minimum level of mutual trust. This implies that a platform must create safe-by-default environments where the main parties (end user, application developer, operating system) can define mutually beneficial terms of engagement. If these parties cannot come to an agreement, then the most trust building operation is to disallow the action (default-deny). The Android platform security model introduced below is based on this notion.

This section is not comprehensive, but briefly summarizes those aspects of the Android ecosystem that have direct implications to the security model:

Android is an end user focused operating system. Although Android strives for flexibility, the main focus is on typical users. The obvious implication is that, as a consumer OS, it must be useful to users and attractive to developers.

The end user focus implies that user interfaces and workflows need to be safe by default and require explicit intent for any actions that could compromise security or privacy. This also means that the OS must not offload technically detailed security or privacy decisions to non-expert users who are not sufficiently skilled or experienced to make them [15].

The Android ecosystem is immense. Different statistics show that in the last few years, the majority of a global, intensely diverse user base already used mobile devices to access Internet resources (i.e. 63% in the US [4], 56% globally [5], with over 68% in Asia and over 80% in India). Additionally, there are hundreds of different OEMs (Original Equipment Manufacturers, i.e. device manufacturers) making tens of thousands of Android devices in different form factors [115] (including, but not limited to, standard smartphones and tablets, watches, glasses, cameras and many other Internet of things device types, handheld scanners/displays and other special-purpose worker devices, TVs, cars, etc.). Some of these OEMs do not have detailed technical expertise, but rely on ODMs (Original Device Manufacturers) for developing hardware and firmware and then re-package or simply re-label devices with their own brand. Only devices shipping with Google services integration need to get their firmware certified, but devices simply based off AOSP can be made without permission or registration. Therefore, there is no single register listing all OEMs, and the list is constantly changing with new hardware concepts being continuously developed. One implication is that changing APIs and other interfaces can lead to large changes in the device ecosystem and take time to reach most of these use cases.

However, devices using Android as a trademarked name to advertise their compatibility with Android apps need to pass the Compatibility Test Suite (CTS). Developers rely on this compatibility when writing apps for this wide variety of different devices. In contrast to some other platforms,

Android explicitly supports installation of apps from arbitrary sources, which led to the development of different app stores and the existence of apps outside of Google Play. Consequently, there is a long tail of apps with a very specific purpose, being installed on only few devices, and/or targeting old Android API releases. Definition of and changes to APIs need to be considerate of the huge number of applications that are part of the Android ecosystem.

Apps can be written in any language. As long as apps interface with the Android framework using the well-defined Java language APIs for process workflow, they can be written in any programming language, with or without runtime support, compiled or interpreted. Android does not currently support non-Java language APIs for the basic process lifecycle control, because they would have to be supported in parallel, making the framework more complex and therefore more error-prone. Note that this restriction is not directly limiting, but apps need to have at least a small Java language wrapper to start their initial process and interface with fundamental OS services. The important implication of this flexibility for security mechanisms is that they cannot rely on compile-time checks or any other assumptions on the build environment. Therefore, Android security needs to be based on runtime protections around the app boundary.

2.2 Threat model

Threat models for mobile devices are different from those commonly used for desktop or server operating systems for two major reasons: by definition, mobile devices are easily lost or stolen, and they connect to untrusted networks as part of their expected usage. At the same time, by being close to users at most times, they are also exposed to even more privacy sensitive data than many other categories of devices. Recent work [108] previously introduced a layered threat model for mobile devices which we adopt for discussing the Android security model within the scope of this paper, but (where meaningful) order threats in each category with lower numbers representing more constrained and higher numbers more capable adversarial settings:

Adversaries can get physical access to Android devices. For all mobile and wearable devices, we have to assume that they will potentially fall under physical control of adversaries at some point. The same is true for other Android form factors such as things, cars, TVs, etc. Therefore, we assume Android devices to be either directly accessible to adversaries or to be in physical proximity to adversaries as an explicit part of the threat model. This includes loss or theft, but also multiple (benign but potentially curious) users sharing a device (such as a TV or tablet). We derive specific threats due to *physical* or *proximal (P)* access:

- **T.P1** (Screen locked or unlocked) devices in physical proximity to (but not under direct control of) an adversary (with the assumed capability to control all available radio communication channels, including cellular, WiFi, Bluetooth, GPS, NFC, and FM), e.g. direct attacks through Bluetooth [2, 61]. Although NFC could be considered to be a separate category to other proximal radio attacks because of the scale of distance, we still include it in the threat class of proximity instead of physical control.
- **T.P2** Powered-off devices under complete physical control of an adversary (with potentially high sophistication up to nation state level attackers), e.g. border control or customs checks.
- **T.P3** Screen locked devices under complete physical control of an adversary, e.g. thieves trying to exfiltrate data for additional identity theft.
- **T.P4** Screen unlocked (shared) devices under control of an authorized but different user, e.g. intimate partner abuse, voluntary submission to a border control, or customs check.

Network communication is untrusted. The standard assumption of network communication under complete control of an adversary certainly also holds for Android devices. This includes the first hop

of network communication (e.g. captive WiFi portals breaking TLS connections and malicious fake access points) as well as other points of control (e.g. mobile network operators or national firewalls), summarized in the usual Dolev-Yao model [67] (or more rigorously formalized as indistinguishable under chosen plaintext attacks, respectively chosen ciphertext attacks (IND-CPA/IND-CCA) for formal cryptographic protocol verification [49]) with additional relay threats for short-range radios (e.g. NFC or BLE wormhole attacks [119]). For practical purposes, we mainly consider two *network-level (N)* threats:

T.N1 Passive eavesdropping and traffic analysis, including tracking devices within or across networks, e.g. based on MAC address or other device network identifiers.
T.N2 Active manipulation of network traffic, e.g. on-path attacks (OPA, also called MITM) on TLS connections or relaying.

These two threats are different from [T.P1] (proximal radio attacks) in terms of scalability of attacks. Controlling a single choke point in a major network can be used to attack a large number of devices, while proximal (last hop) radio attacks require physical proximity to target devices.

Untrusted code is executed on the device. One fundamental difference to other mobile operating systems is that Android intentionally allows (with explicit consent by end users) installation of *application (A) code* from arbitrary sources, and does not enforce vetting of apps by a central instance. This implies attack vectors on multiple levels (cf. [108]):

T.A1 Abusing APIs supported by the OS with malicious intent, e.g. spyware.
T.A2 Abusing APIs supported by other apps installed on the device [10].
T.A3 Untrusted code from the web (i.e. JavaScript) is executed without explicit consent.
T.A4 Mimicking system or other app user interfaces to confuse users (based on the knowledge that standard in-band security indicators are not effective [66, 117]), e.g. to input PIN/password into a malicious app [77].
T.A5 Reading content from system or other app user interfaces, e.g. to screen-scrape confidential data from another app [90, 97].
T.A6 Injecting input events into system or other app user interfaces [80].
T.A7 Exploiting bugs in the OS, e.g. kernel, drivers, or system services [3, 8, 9, 11].

Untrusted content is processed by the device. In addition to directly executing untrusted code, devices process a wide variety of untrusted data, including rich (in the sense of complex structure) media. This directly leads to threats concerning processing of *data (D)* and metadata:

T.D1 Abusing unique identifiers for targeted attacks (which can happen even on trusted networks), e.g. using a phone number or email address for spamming or correlation with other data sets, including locations.
T.D2 Exploiting code that processes untrusted content in the OS or apps, e.g. in media libraries [1]. This can be both a local as well as a remote attack surface, depending on where input data is taken from.

3 THE ANDROID PLATFORM SECURITY MODEL

The basic security model described in this section has informed the design of Android, and has been refined but not fundamentally changed. Given the ecosystem context and threat model explained above, the Android security model balances security and privacy requirements of users with security requirements of applications and the platform itself. The threat model described above includes threats to all stakeholders, and the security model and its enforcement by the Android platform aims to address all of them. The Android platform security model is informally defined by 5 rules:

① *Multi-party consent.* No action should be executed unless all main parties agree — in the standard case, these are *user*, *platform*, and *developer* (implicitly representing stakeholders such as content producers and service providers). Any one party can veto the action. This multi-party consent spans the traditional two dimensions of subjects (users and application processes) vs. objects (files, network sockets and IPC interfaces, memory regions, virtual data providers, etc.) that underlie most security models (e.g. [128]). Any party (or more generally actor) that creates a data item is implicitly granted control over this particular instance of data representation. Focusing on (regular and pseudo) files as the main category of objects to protect, the default control over these files depends on their location and which party created them:

- Data in shared storage is controlled by users.
- Data in private app directories and app virtual address space is controlled by apps.
- Data in special system locations is controlled by the platform (e.g. list of granted permissions).

Data in run-time memory (RAM) is by default controlled by the respective platform or app process. However, it is important to point out that, under multi-party consent, even if one party primarily controls a data item, it may only act on it if the other involved parties consent. *Control over data also does not imply ownership* (which is a legal concept rather than a technical one and therefore outside the scope of an OS security model).

While this principle has long been the default for filesystem access control (DAC, cf. section 4.3.1 below), we consider it a global model rule and exceptions such as device backup (cf. section 5) can be argued about within the scope of the security model. There are other corner cases in which only a subset of all parties may need to consent (for actions in which the user only uses platform/OS services without involvement of additional apps) or an additional party may be introduced (e.g. on devices or profiles controlled by a mobile device management, this policy is also considered as a party for consenting to an action).

> Public information and resources are out of scope of this access control and available to all parties; particularly all static code and data contained in the AOSP system image and apps (mostly in the Android Package (APK) format) is considered to be public (cf. Kerckhoff's principle) — if an actor publishes the code, this is interpreted as implicit consent to access. However, it is generally accepted that such public code and data is read-only to all parties and its integrity needs to be protected, which is explicitly in scope of the security measures.

② *Open ecosystem access.* Both users and developers are part of an open ecosystem that is not limited to a single application store. Central vetting of developers or registration of users is not required. This aspect has an important implication for the security model: generic app-to-app interaction is explicitly supported. Instead of creating specific platform APIs for every conceivable workflow, app developers are free to define their own APIs they offer to other apps.

③ *Security is a compatibility requirement.* The security model is part of the Android specification, which is defined in the Compatibility Definition Document (CDD) [19] and enforced by the Compatibility (CTS), Vendor (VTS), and other test suites. Devices that do not conform to CDD and do not pass CTS are not Android. Within the scope of this paper, we define *rooting* as modifying the system to allow starting processes that are not subject to sandboxing and isolation. Such rooting, both intentional and malicious, is a specific example of a non-compliant change which violates CDD. As such, only CDD-compliant devices are considered. While many devices support unlocking their bootloader and flashing modified firmware[8], such modifications may be considered incompatible under CDD if security assurances do not hold. Verified boot and hardware key attestation can be

[8] Google Nexus and Pixel devices as well as many others support the standard `fastboot oem unlock` command to allow flashing any firmware images to actively support developers and power users. However, executing this unlocking workflow

used to validate if currently running firmware is in a known-good state, and in turn may influence consent decisions by users and developers.

④ *Factory reset restores the device to a safe state.* In the event of security model bypass leading to a persistent compromise, a factory reset, which wipes/reformats the writable data partitions, returns a device to a state that depends only on integrity protected partitions. In other words, system software does not need to be re-installed, but wiping the data partition(s) will return a device to its default state. Note that the general expectation is that the read-only device software may have been updated since originally taking it out of the box, which is intentionally not downgraded by factory reset. Therefore, more specifically, factory reset returns an Android device to a state that only depends on system code that is covered by Verified Boot, but does not depend on writable data partitions.

⑤ *Applications are security principals.* The main difference to traditional operating systems that run apps in the context of the logged-in user account is that Android apps are not considered to be fully authorized agents for user actions. In the traditional model typically implemented by server and desktop OS, there is often no need to even exploit the security boundary because running malicious code with the full permissions of the main user is sufficient for abuse. Examples are many, including file encrypting ransomware [93, 121] (which does not violate the OS security model if it simply re-writes all the files the current user account has access to) and private data leakage (e.g. browser login tokens [105], history or other tracking data, cryptocurrency wallet keys, etc.).

Summary. Even though, at first glance, the Android security model grants less power to users compared to traditional operating systems that do not impose a multi-party consent model, there is an immediate benefit to end users: if one app cannot act with full user privileges, the user cannot be tricked into letting it access data controlled by other apps. In other words, requiring application developer consent – enforced by the platform – helps avoid user confusion attacks and therefore better protects private data.

The Android platform security model does not currently have a simple, consistent representation in formal notation because these rules evolved from practical experience instead of a top-down theoretical design. Balancing the different requirements of a complex ecosystem is a large scale engineering problem that requires layers of abstraction. Therefore, we have to combine multiple different security controls (such as memory isolation, filesystem DAC/MAC, biometric user authentication, or network traffic encryption) that operate under their own respective models and are not necessarily consistent with each other (see e.g. [84] for interactions between only the DAC and MAC policies). The five rules are, at the time of this writing, the simplest expression of how these different security controls combine at the meta level. However, appendix A gives a first, albeit incomplete, formalization of the access control properties of these rules. It is subject to future work to model all important aspects more formally and to reason about the cross-abstraction interactions of these rules with lower level models of underlying security controls.

4 IMPLEMENTATION

Android's security measures implement the security model and are designed to address the threats outlined above. In this section we describe security measures and indicate which threats they mitigate, taking into account the architectural security principles of 'defense in depth' and 'safe by design':

will forcibly factory reset the device (wiping all data) to make sure that security guarantees are not retroactively violated for data on the device.

Defense in depth. A robust security system is not sufficient if the acceptable behavior of the operating system allows an attacker to accomplish all of their goals without bypassing the security model (e.g. ransomware encrypting all files it has access to under the access control model). Specifically, violating any of the above principles should require such bypassing of controls on-device (in contrast to relying on off-device verification e.g. at build time).

Therefore, the primary goal of any security system is to enforce its model. For Android operating in a multitude of environments (see above for the threat model), this implies an approach that does not immediately fail when a single assumption is violated or a single implementation bug is found, even if the device is not up to date. Defense in depth is characterized by rendering individual vulnerabilities more difficult or impossible to exploit, and increasing the number of vulnerabilities required for an attacker to achieve their goals. We primarily adopt four common security strategies to prevent adversaries from bypassing the security model: *isolation and containment* (section 4.3), *exploit mitigation* (section 4.6), *integrity* (section 4.7), and *patching/updates* (section 4.8).

Safe by design/default. Components should be safe by design. That is, the default use of an operating system component or service should always protect security and privacy assumptions, potentially at the cost of blocking some use cases. This principle applies to modules, APIs, communication channels, and generally to interfaces of all kinds. When variants of such interfaces are offered for more flexibility (e.g. a second interface method with more parameters to override default behavior), these should be hard to abuse, either unintentionally or intentionally. Note that this architectural principle targets developers, which includes device manufacturers, but implicitly includes users in how security is designed and presented in user interfaces. Android targets a wide range of developers and intentionally keeps barriers to entry low for app development. Making it hard to abuse APIs not only guards against malicious adversaries, but also mitigates genuine errors resulting e.g. from incomplete knowledge of an interface definition or caused by developers lacking experience in secure system design. As in the defense in depth approach, there is no single solution to making a system safe by design. Instead, this is considered a guiding principle for defining new interfaces and refining – or, when necessary, deprecating and removing – existing ones. For guarding user data, the basic strategies for supporting safety by default are: *enforced consent* (section 4.1), *user authentication* (section 4.2), and *by-default encryption at rest* (section 4.4) and *in transit* (section 4.5).

4.1 Enforcing meaningful consent

Methods of giving meaningful consent vary greatly between actors, as well as potential issues and constraints.

We use two examples to better describe the consent parties:

- Sharing data from one app to another requires:
  - user consent through the user selecting a target app in the share dialog;
  - developer consent of the source app by initiating the share with the data (e.g. image) they want to allow out of their app;
  - developer consent of the target app by accepting the shared data; and
  - platform consent by arbitrating the data access between different components and ensuring that the target app cannot access any other data than the explicitly shared item through the same link, which forms a temporary trust relationship between two apps.
- Changing mobile network operator (MNO) configuration option requires:
  - user consent by selecting the options in a settings dialog;

- (MNO app) developer consent by implementing options to change these configuration items, potentially querying policy on backend systems; and
- platform consent by verifying e.g. policies based on country regulations and ensuring that settings do not impact platform or network stability.

Actors consenting to any action must be empowered to base their decision on information about the action and its implications and must have meaningful ways to grant or deny this consent. This applies to both users and developers, although very different technical means of enforcing (lack of) consent apply. Consent is not only required from the actor that created a data item, but from all involved actors. Consent decisions should be enforced and not self-policed, which can happen at run-time (often, but not always, through platform mediation) or build respectively distribution time (e.g. developers including or not including code in particular app versions).

4.1.1 Developer(s)

Unlike traditional desktop operating systems, Android ensures that the developer consents to actions on their app or their app's data. This prevents large classes of abusive behavior where unrelated apps inject code into or access/leak data from other applications on a user's device.

Consent for developers, unlike the user, is given via the code they sign and the system executes, uploading the app to an app store and agreeing to the associated terms of service, and obeying other relevant policies (such as CDD for code by an OEM in the system image). For example, an app can consent to the user sharing its data by providing a respective mechanism, e.g. based on OS sharing methods such as built-in implicit `Intent` resolution chooser dialogs [29]. Another example is debugging: as assigned virtual memory content is controlled by the app, debugging from an external process is only allowed if an app consents to it (specifically through the `debuggable` flag in the app manifest). By uploading an app to the relevant app store, developers also provide the consent for this app to be installed on devices that fetch from that store under appropriate pre-conditions (e.g. after successful payment).

Meaningful consent then is ensuring that APIs and their behaviors are clear and the developer understands how their application is interacting with or providing data to other components. Additionally, we assume that developers of varying skill levels may not have a complete understanding of security nuances, and as a result APIs must also be safe by default and difficult to incorrectly use in order to avoid accidental security regressions. One example of a lesson learned in these regards is that early Android apps occasionally used meant-to-be-internal APIs for unsupported purposes and often in an insecure way. Android 9 introduced a major change by only supporting access to APIs explicitly listed as external (https://developer.android.com/reference/packages) and putting restrictions on others [33]. Developer support was added e.g. in the form of specific log messages to point out internal API usage for debuggable versions of apps. This has two main benefits: a) the attack surface is reduced, both towards the platform and apps that may rely on undefined and therefore changing internal behavior; and b) refactoring of internal platform interfaces and components from one version to another is enabled with fewer app compatibility constraints.

In order to ensure that it is the app developer and not another party that is consenting, applications are signed by the developer. This prevents third parties — including the app store — from replacing or removing code or resources in order to change the app's intended behavior. However, the app signing key is trusted implicitly upon first installation, so replacing or modifying apps in transit when a user first installs them (e.g. when initially side-loading apps) is currently out of scope of the platform security model. Previous Android versions relied on a single developer certificate that was trusted on initial install of an app and therefore made it impossible to change the underlying private key e.g. in the case of the key having become insecure [47]. Starting with Android 9, independently

developed key rotation functionality was added with APK Signature Scheme v3 [39] to support delegating the ability to sign to a new key by using a key that was previously granted this ability by the app using so-called *proof-of-rotation* structs[9].

These two examples (controlled access to internal Android platform components and developer signing key rotation) highlight that handling multi-party consent in a complex ecosystem is challenging even from the point of a single party: some developers may wish for maximum flexibility (access to all internal components and arbitrarily complex key handling), but the majority tends to be overwhelmed by the complexity. As the ecosystem develops, changes are therefore necessary to react to lessons learned. In these examples, platform changes largely enabled backwards compatibility without changing (no impact when key rotation is not used by a developer) or breaking (most apps do not rely on internal APIs) existing apps. When changes for developers are necessary, these need to be deployed over a longer period to allow adaptation, typically with warnings in one Android release and enforced restrictions only in the next one.

4.1.2 The Platform

While the platform, like the developer, consents via code signing, the goals are quite different: the platform acts to ensure that the system functions as intended. This includes enforcing regulatory or contractual requirements (e.g. communication in cell-based networks) as well as taking an opinionated stance on what kinds of behaviors are acceptable (e.g. mitigating apps from applying deceptive behavior towards users). Platform consent is enforced via Verified Boot (see below for details) protecting the system images from modification, internal compartmentalization and isolation between components, as well as platform applications using the platform signing key and associated permissions, much like applications.

> *Note on the platform as a party:* Depending on how the involved stakeholders (parties for consent) and enforcing mechanisms are designated, either an inherent or an apparent asymmetry of power to consent may arise:
>
> (a) If the Android "platform" is seen as a single entity (composed of hardware, firmware, OS kernel, system services, libraries, and app runtime), then it may be considered omniscient in the sense of having access to and effectively controlling all data and processes on the system. Under this point of view, the conflict of interest between being one party of consent and simultaneously being the enforcing agent gives the platform overreaching power over all other parties.
>
> (b) If Android as a platform is considered in depth, it consists of many different components. These can be seen as individual representatives of the platform for a particular interaction involving multi-party consent, while other components act as enforcing mechanism for that consent. In other words, the Android platform is structured in such a way as to minimize trust in itself and contain multiple mechanisms of isolating components from each other to enforce each other's limitations (cf. section 4.3). One example is playing media files: even when called by an app, a media codec cannot directly access the underlying resources if the user has not granted this through the media server, because MAC policies in the Linux kernel do no allow such bypass (cf. section 4.3.3). Another example is storage of cryptographic keys, which is isolated even from the Linux kernel itself and enforced through hardware separation (cf. section 4.3.5). While this idealized model of platform parties requiring consent for their actions is the abstract goal of the security model we describe, in practice there still are individual components that sustain the asymmetry between the parties. Each new version of Android continues to further

[9] The Google Play app store now explicitly supports key rotation through Play Signing, but does not yet support key rotation with multiple developer-held keys. The Android platform itself is prepared for arbitrarily complex key rotation strategies.

strengthen the boundaries of platform components among each other, as described in more detail below.

Within the scope of this paper, we take the second perspective when it comes to notions of consent involving the platform itself, i.e. considering the platform to be multiple parties whose consent is being enforced by independent mechanisms (mostly the Linux kernel isolating platform components from each other, but also including out-of-kernel components in a trusted execution environment). However, when talking about the whole system implementing our Android security model, in favor of simpler expression we will generally refer to the platform as the combination of all (AOSP) components that together act as an enforcing mechanism for other parties, as defined in the introduction.

Lessons learned over the evolution of the Android platform are clearly visible through the introduction of new security mitigations and tightening of existing controls, as summarized in tables 1 to 4 and too extensive to describe here. Other examples include use of strings, namespaces, links, etc. provided by apps with the potential to misguide or outright deceive users into providing consent against their wishes. The platform not only manages consent for its own components, but mediates user and developer consent responses, and therefore has to adapt to changes in the ecosystem.

4.1.3 User(s)

Achieving meaningful user consent is by far the most difficult and nuanced challenge in determining meaningful consent. Some of the guiding principles have always been core to Android, while others were refined based on experiences during the 10 years of development so far:

- **Avoid over-prompting.** Over-prompting the user leads to prompt fatigue and blindness (cf. [17]). Prompting the user with a yes/no prompt for every action does not lead to meaningful consent as users become blind to the prompts due to their regularity.
- **Prompt in a way that is understandable.** Users are assumed not to be experts or understand nuanced security questions (cf. [76]). Prompts and disclosures must be phrased in a way that a non-technical user can understand the effects of their decision.
- **Prefer pickers and transactional consent over wide granularity.** When possible, we limit access to specific items instead of the entire set. For example, the Contacts Picker allows the user to select a specific contact to share with the application instead of using the Contacts permission. These both limit the data exposed as well as present the choice to the user in a clear and intuitive way.
- **The OS must not offload a difficult problem onto the user.** Android regularly takes an opinionated stance on what behaviors are too risky to be allowed and may avoid adding functionality that may be useful to a power user but dangerous to an average user.
- **Provide users a way to undo previously made decisions.** Users can make mistakes. Even the most security and privacy-savvy users may simply press the wrong button from time to time, which is even more likely when they are being tired or distracted. To mitigate against such mistakes or the user simply changing their mind, it should be easy for the user to undo a previous decision whenever possible. This may vary from denying previously granted permissions to removing an app from the device entirely.

Additionally, it is critical to ensure that the user who is consenting is the legitimate user of the device and not another person with physical access to the device ([T.P2]-[T.P4]), which directly relies on the next component in the form of the Android lockscreen. Implementing model rule ① (multi-party consent) is cross-cutting on all system layers.

For devices that do not have direct, regular user interaction (embedded IoT devices, shared devices in the infrastructure such as TVs, etc.), user consent may be given slightly differently depending on the specific form factor. A smart phone may often act as a UI proxy to configure consent/policy for other embedded devices. For the remainder of this paper but without loss of generality, we primarily assume smart phone/tablet type form factors with direct user interaction.

As with developer consent, lessons learned for user consent over the development of the ecosystem will require changes over time. The biggest changes for user consent were the introduction of runtime permissions with Android 6.0 and non-binary, context dependent permissions with Android 10 (cf. section 4.3.1), another example are restrictions to accessibility service APIs (which require user consent but were abused) as well as clipboard access and background activity starting in Android 10 (cf. table 1).

4.2 Authentication

Authentication is a gatekeeper function for ensuring that a system interacts with its owner or legitimate user. On mobile devices the primary means of authentication is via the lockscreen. Note that a lockscreen is an obvious trade-off between security and usability: On the one hand, users unlock phones for short (10-250 seconds) interactions about 50 times per day on average and even up to 200 times in exceptional cases [73, 87], and the lockscreen is obviously an immediate hindrance to frictionless interaction with a device [85, 86]. On the other hand, devices without a lockscreen are immediately open to being abused by unauthorized users ([T.P2]-[T.P4]), and the OS cannot reliably enforce user consent without authentication.

In their current form, lockscreens on mobile devices largely enforce a binary model — either the whole phone is accessible, or the majority of functions (especially all security or privacy sensitive ones) are locked. Neither long, semi-random alphanumeric passwords (which would be highly secure but not usable for mobile devices) nor swipe-only lockscreens (usable, but not offering any security) are advisable. Therefore, it is critically important for the lockscreen to strike a reasonable balance between security and usability, as it enables further authentication on higher levels.

4.2.1 Tiered lockscreen authentication

Towards this end, recent Android releases use a tiered authentication model where a secure knowledge-factor based authentication mechanism can be backed by convenience modalities that are functionally constrained based on the level of security they provide. The added convenience afforded by such a model helps drive lockscreen adoption and allows more users to benefit both from the immediate security benefits of a lockscreen and from features such as file-based encryption that rely on the presence of an underlying user-supplied credential. As an example of how this helps drive lockscreen adoption, starting with Android 7.x we see that 77% of devices with fingerprint sensors have a secure lockscreen enabled, while only 50% of devices without fingerprints have a secure lockscreen[10].

As of Android 10, the tiered authentication model splits modalities into three tiers.

- *Primary Authentication* modalities are restricted to knowledge-factors and by default include password, PIN, and pattern[11]. Primary authentication provides access to all functions on the phone. It is well-known that the security/usability-balance of these variants is different: complex passwords have the highest entropy but worst usability, while PINs and patterns are a middle balance but may suffer e.g. from smudge [45] ([T.P2]-[T.P3]) or should surfing attacks [69, 92] ([T.P1]). However, a knowledge-factor is still considered a trust anchor for

[10] These numbers are from internal analysis that has not yet been formally published.

[11] We explicitly refer to patterns connecting multiple dots in a matrix, not the whole-screen swipe-only lockscreen interaction that does not offer any security.

device security and therefore the only one able to unlock a device from a previously fully locked state (e.g. from being powered off).

- *Secondary Authentication* modalities are biometrics — which offer easier, but potentially less secure (than Primary Authentication), access into a user's device. Secondary modalities are themselves split into sub-tiers based on how secure they are, as measured along two axes:
  (1) *Spoofability* as measured by the Spoof Acceptance Rate (SAR) of the modality [113]. Accounting for an explicit attacker in the threat model on the level of [T.P2]-[T.P3] helps reduce the potential for insecure unlock methods [110].
  (2) *Security of the biometric pipeline*, where a biometric pipeline is considered secure if neither platform or kernel compromise confer the ability to read raw biometric data or inject data into the biometric pipeline to influence an authentication decision.

  These axes are used to categorize secondary authentication modalities into three sub-tiers, where each sub-tier has constraints applied in proportion to the level of security they provide. Secondary modalities are also prevented from performing some actions — for example, they do not decrypt file-based or full-disk encrypted user data partitions (such as on first boot) and are required to fallback to primary authentication once every 72 hours. If a weak biometric does not meet either of the criteria (spoofability and pipelines security), then they cannot unlock Keymaster auth-bound keys and have a shorter fallback period. Android 10 introduced support for implicit biometric modalities in `BiometricPrompt` for modalities that do not require explicit interaction, for example face recognition. Android 11 further introduces new features such as allowing developers to specify the authentication types accepted by their apps and thus the preferred level of security [42].
- *Tertiary Authentication* modalities are alternate modalities such as unlocking when paired with a trusted Bluetooth device, or unlocking at trusted locations. Tertiary modalities are subject to all the constraints of secondary modalities. Additionally, like the weaker secondary modalities, tertiary modalities are also restricted from granting access to Keymaster auth-bound keys (such as those required for payments) and also require a fallback to primary authentication after any 4-hour idle period.

The Android lockscreen is currently implemented by Android system components above the kernel, specifically `Keyguard` and the respective unlock methods (some of which may be OEM specific). User knowledge factors of secure lockscreens are passed on to Gatekeeper/Weaver (explained below) both for matching them with stored templates and deriving keys for storage encryption. One implication is that a kernel compromise could lead to bypassing the lockscreen — but only after the user has logged in for the first time after reboot.

4.2.2 Authenticating to third parties: Android devices as a second factor

As of April 2019, lockscreen authentication on Android 7+ can now be used for FIDO2/WebAuthn [12, 141] authentication to web pages, additionally making Android phones second authentication factors for desktop browsers through implementing the Client to Authenticator Protocol (CTAP) [126]. While this support is currently implemented in Google Play Services [79], the intention is to include support directly in AOSP in the future when standards have sufficiently settled down to become stable for the release cycle of multiple Android releases.

4.2.3 Authenticating to third parties: Identity Credential

While the lockscreen is the primary means for user-to-device (U2D) authentication and various methods support device-to-device (D2D) authentication (both between clients and client/server authentication such as through WebAuthn), identifying the device owner to other parties has

not been in focus so far. Through the release of a JetPack library[12], apps can make use of a new "Identity Credential" subsystem to support privacy-first identification [89] (and, to a certain degree, authentication). One example are upcoming third-party apps to support mobile driving licenses (mDL) according to the ISO 18013-5 standard [13]. The first version of this subsystem targets in-person presentation of credentials, and identification to automated verification systems is subject to future work.

Android 11 includes the Identity Credential subsystem in the form of a new HAL, a new system daemon, and API support in AOSP [25, 148]. If the hardware supports direct connections between the NFC controller and tamper-resistant dedicated hardware, credentials will be able to be marked for "Direct Access"[13] to be available even when the main application processor is no longer powered (e.g. in a low-battery case).

4.3 Isolation and Containment

One of the most important parts of enforcing the security model is to enforce it at runtime against potentially malicious code already running on the device. The Linux kernel provides much of the foundation and structure upon which Android's security model is based. Process isolation provides the fundamental security primitive for sandboxing. With very few exceptions, the process boundary is where security decisions are made and enforced — Android intentionally does not rely on in-process compartmentalization such as the Java security model. The security boundary of a process is comprised of the process boundary and its entry points and implements rule ⑤ (apps as security principals) and rule ② (open ecosystem): an app does not have to be vetted or pre-processed to run within the sandbox. Strengthening this boundary can be achieved by a number of means such as:

- Access control: adding permission checks, increasing the granularity of permission checks, or switching to safer defaults (e.g. default deny) to address the full range of threats [T.A1]-[T.A7] and [T.D1]-[T.D2].
- Attack surface reduction: reducing the number of entry points, particularly [T.A1], [T.A2], and [T.A7], i.e. the principle of least privilege.
- Containment: isolating and de-privileging components, particularly ones that handle untrusted content as in [T.A3] and [T.D2].
- Architectural decomposition: breaking privileged processes into less privileged components and applying attack surface reduction for [T.A2]-[T.A7] and [T.D2].
- Separation of concerns: avoiding duplication of functionality.

In this section we describe the various sandboxing and access control mechanisms used on Android on different layers and how they improve the overall security posture.

4.3.1 Access control

Android uses three distinct permission mechanisms to perform access control:

- **Discretionary Access Control (DAC):** Processes may grant or deny access to resources that they own by modifying permissions on the object (e.g. granting world read access) or by passing a handle to the object over IPC. On Android this is implemented using UNIX-style permissions that are enforced by the kernel and URI permission grants. Processes running as the `root` user often have broad authority to override UNIX permissions (subject to MAC

[12] Available at https://developer.android.com/jetpack/androidx/releases/security

[13] See the HAL definition at https://android-review.googlesource.com/c/platform/hardware/interfaces/+/1151485/30/identity/1.0/IIdentityCredentialStore.hal.

permissions – see below). URI permission grants provide the core mechanism for app-to-app interaction allowing an app to grant selective access to pieces of data it controls.

- **Mandatory Access Control (MAC):** The system has a security policy that dictates what actions are allowed. Only actions explicitly granted by policy are allowed. On Android this is implemented using SELinux [125] and primarily enforced by the kernel. Android makes extensive use of SELinux to protect system components and assert security model requirements during compatibility testing.
- **Android permissions** gate access to sensitive data and services. Enforcement is primarily done in userspace by the data/service provider (with notable exceptions such as `INTERNET`). Permissions are defined statically in an app's `AndroidManifest.xml` [23], though not all permissions requested may be granted.
  Android 6.0 brought a major change by no longer guaranteeing that all requested permissions are granted when an application is installed. This was a direct result of the realization that users were not sufficiently equipped to make such a decision at installation time (cf. [75, 76, 118, 143]).
  The second major change in Android permissions was introduced with Android 10 in the form of non-binary, context dependent permissions: in addition to *Allow* and *Deny*, some permissions (particularly location, and starting with Android 11 others like camera and microphone) can now be set to *Allow only while using the app*. This third state only grants the permission when an app is in the foreground, i.e. when it either has a visible activity or runs a foreground service with permanent notification [59]. Android 11 extended this direction with one-time permissions that form another variant in the context dependent state between unconditional allow and deny.

  At a high level Android permissions fall into one of five classes in increasing order of severity, whose availability is defined by their `protectionLevel` attribute [24] with two parts (the protection level itself and a number of optional flags):

(1) *Audit-only permissions*: These are install time permissions with protection level `normal` that do not pose much privacy or security risk and are granted automatically at install time. They are primarily used for auditability of app behavior.
(2) *Runtime permissions*: These are permissions with protection level `dangerous` and apps must both declare them in their manifest as well as request users grant them during use. These permissions are guarding commonly used sensitive user data, and depending on how critical they are for the current functioning of an application, different strategies for requesting them are recommended [21]. While runtime permissions are fairly fine-grained to support auditing and enforcement in-depth, they are grouped into logical permissions using the `permissionGroup` attribute. When requesting runtime permissions, the group appears as a single permission to avoid over-prompting.
(3) *Special Access permissions*: For permissions that expose more or are higher risk than runtime permissions there exists a special class of permissions with much higher granting friction that the application cannot show a runtime prompt for. Specific examples are device admin, notification listeners, or installing packages. In order for a user to allow an application to use a special access permission, the user must go to settings and manually grant the permission to the application.
(4) *Privileged permissions*: These permissions are for pre-installed applications only and allow privileged actions such as modifying secure settings or carrier billing. They typically cannot be granted by users during run-time but OEMs grant them by whitelisting the `privileged` permissions for individual apps [32] in the system image.

`Privileged` protection level permissions are usually coupled with the `signature` level.

(5) *Signature permissions*: These permissions with protection level `signature` are only available to components signed with the same key as the (platform or application) component which declares the permission — which is the platform signing key for platform permissions. They are intended to guard internal or highly privileged actions, e.g. configuring the network interfaces and are granted at install time if the application is allowed to use them.

Additionally, there are a number of protection flags that modify the grantability of permissions. For example, the `BLUETOOTH_PRIVILEGED` permission has a `protectionLevel` of `signature|privileged`, with the privileged flag allowing privileged applications to be granted the permission (even if they are not signed with the platform key).

Each of the three permission mechanisms roughly aligns with how one of the three parties of the multi-party grant consent (rule ①). The platform utilizes MAC, apps use DAC, and users consent by granting Android permissions. Note that permissions are not intended to be a mechanism for obtaining consent in the legal sense but a technical measure to enforce auditability and control. It is up to the app developer processing personal user data to meet applicable legal requirements.

4.3.2 Application sandbox

Android's original DAC application sandbox separated apps from each other and the system by providing each application with a unique UNIX user ID (UID) and a directory owned by the app. This approach was quite different from the traditional desktop approach of running applications using the UID of the physical user. The unique per-app UID simplifies permission checking and eliminates per-process ID (PID) checks, which are often prone to race conditions. Permissions granted to an app are stored in a centralized location (`/data/system/packages.xml`). to be queried by other services. For example, when an app requests location from the location service, the location service queries the permissions service to see if the requesting UID has been granted the location permission.

Starting with Android 4, UIDs are also used for separating multiple physical device users. As the Linux kernel only supports a single numeric range for UID values, device users are separated through a larger offset (`AID_USER_OFFSET=100000` as defined in AOSP source[14]) and apps installed for each user are assigned UIDs in a defined range (from `AID_APP_START=10000` to `AID_APP_END=19999`) relative to the device user offset. This combination is referred to as the Android ID (AID).

The UID sandbox had a number of shortcomings. Processes running as root were essentially unsandboxed and possessed extensive power to manipulate the system, apps, and private app data. Likewise, processes running as the system UID were exempt from Android permission checks and permitted to perform many privileged operations. Use of DAC meant that apps and system processes could override safe defaults and were more susceptible to dangerous behavior, such as symlink following or leaking files/data across security boundaries via IPC or `fork`/`exec`. Additionally, DAC mechanisms can only apply to files on file systems that support access controls lists (respectively simple UNIX access bits). The main implication is that the FAT family of file systems, which is still commonly used on extended storage such as (micro-) SD cards or media connected through USB, does not directly support applying DAC. On Android, each app has a well-known directory on external storage devices, where the package name of the app is included into the path (e.g. `/sdcard/Android/data/com.example`). Since the OS already maintains a mapping from package name to UID, it can assign UID ownership to all files in these well-known directories, effectively creating a DAC on a filesystem that doesn't natively support it. From Android 4.4 to Android 7.x, this mapping was implemented through FUSE, while Android 8.0 and later implement an in-kernel

[14] See `system/core/include/private/android_filesystem_config.h` in the AOSP source tree.

`sdcardfs` for better performance. Both are equivalent in maintaining the mapping of app UIDs to implement effective DAC. Android 10 introduced *scoped storage*, which limits app access to its own external directory path as well as media files that itself created in the shared media store.

Table 1. Application sandboxing improvements in Android releases

Release	Improvement	Threats mitigated
≤ 4.3	Isolated process: Apps may optionally run services in a process with no Android permissions and access to only two binder services. For example, the Chrome browser runs its renderer in an isolated process for rendering untrusted web content.	[T.A3] access to [T.N1] [T.A2][T.A5] [T.A6][T.A7]
5.x	SELinux: SELinux was enabled for all userspace, significantly improving the separation between apps and system processes. Separation between apps is still primarily enforced via the UID sandbox. A major benefit of SELinux is the auditability/testability of policy. The ability to test security requirements during compatibility testing increased dramatically with the introduction of SELinux.	[T.A7][T.D2]
5.x	Webview moved to an updatable APK, independent of a full system OTA.	[T.A3]
6.x	Runtime permissions were introduced, which moved the request for dangerous permission from install to first use (cf. above description of permission classes).	[T.A1]
6.x	Multi-user support: SELinux categories were introduced for a per-physical-user app sandbox.[15]	[T.P4]
6.x	Safer defaults on private app data: App home directory moved from `0751` UNIX permissions to `0700` (based on `targetSdkVersion`).	[T.A2]
6.x	SELinux restrictions on `ioctl` system call: 59% of all app reachable kernel vulnerabilities were through the ioctl() syscall, and these restrictions limit reachability of potential kernel vulnerabilities from user space code [137, 138].	[T.A7][T.D2]
6.x	Removal of app access to `debugfs` (9% of all app-reachable kernel vulnerabilities).	[T.A7][T.D2]
6.x	Moving SYSTEM_ALERT_WINDOW, WRITE_SETTINGS, and CHANGE_NETWORK_STATE to special permission category	[T.A1][T.A4]
7.x	`hidepid=2`: Remove `/proc/<pid>` side channel used to infer when apps were started.	[T.A4]
7.x	perf-event-hardening (11% of app reachable kernel vulnerabilities were reached via `perf_event_open()`).	[T.A7]
7.x	Safer defaults on `/proc` filesystem access.	[T.A1][T.A4]
7.x	OPA/MITM CA certificates are not trusted by default.	[T.N2]
8.x	Safer defaults on `/sys` filesystem access.	[T.A1][T.A4]
8.x	All apps run with a `seccomp` filter intended to reduce kernel attack surface.	[T.A7][T.D2]

[15] See https://marc.info/?l=seandroid-list&m=141150669811670&w=4 for a detailed explanation of the use of SELinux MLS categories by the original author.

8.x	Webviews for all apps move into the isolated process.	[T.A3]
8.x	Apps must opt-in to use cleartext network traffic.	[T.N1]
9.0	Per-app SELinux sandbox (for apps with `targetSdkVersion=P` or greater).	[T.A2][T.A4]
10	Apps can only start a new activity with a visible window, in the foreground activity 'back stack', or if more specific exceptions apply [41].	[T.A2][T.A3] [T.A4][T.A7]
10	File access on external storage is scoped to app-owned files.	[T.A1][T.A2]
10	Reading clipboard data is only possible for the app that currently has input focus or is the default IME app.	[T.A5]
10	`/proc/net` limitations and other side channel mitigations.	[T.A1]
11	Legacy access of non-scoped external storage is no longer available.	[T.A1][T.A2]

Despite its deficiencies, the UID sandbox laid the groundwork and is still the primary enforcement mechanism that separates apps from each other. It has proven to be a solid foundation upon which to add additional sandbox restrictions. These shortcomings have been mitigated in a number of ways over subsequent releases, especially through the addition of MAC policies with SElinux in enforcing mode starting with Android 5, but also including many other mechanisms such as runtime permissions and attack surface reduction (cf. Table 1). In addition to SELinux, `seccomp` filters complement the MAC policy on a different level of syscall granularity. While the Chrome app is currently the main user of fine-grained `seccomp` filters, others can also use them to internally minimize attack surface for their components.

Another particular example for the interplay between DAC and MAC policies and changes based on lessons learned are the more recent restrictions to `ioctl`, `/proc`, and `/sys` since Android 7. As described more generally in section 4.1, limiting access to such internal interfaces improves app compatibility between platform versions and supports easier internal refactoring. For these kernel interfaces, restricting access had another benefit towards user privacy: while few apps used these kernel interfaces for legitimate purposes that could not be fulfilled with existing Android APIs, they were also abused by other apps for side-channel attacks [116] on data not otherwise accessible through their lack of required Android permissions (e.g. network hardware MAC addresses). Restricting access to these interfaces to follow an allow- instead of block-list approach is therefore a logical development in line with the defense-in-depth principle.

Rooting, as defined above, has the main aim of enabling certain apps and their processes to break out of this application sandbox in the sense of granting "root" user privileges [88], which override the DAC rules (but not automatically MAC policies, which led to extended rooting schemes with processes intentionally exempt from MAC restrictions). Malware may try to apply these rooting approaches through temporary or permanent exploits and therefore bypass the application sandbox.

4.3.3 Sandboxing system processes

In addition to the application sandbox, Android launched with a limited set of UID sandboxes for system processes. Notably, Android's architects recognized the inherent risk of processing untrusted media content and so isolated the media frameworks into UID `AID_MEDIA`, and this sandboxing has been strengthened from release to release with continuously more fine-grained isolation [127]. Figure 1 gives an overview of specifically the sandboxing and isolation improvements for the media server and codecs. Other processes that warranted UID isolation include the telephony stack, WiFi, and Bluetooth (cf. Table 2).

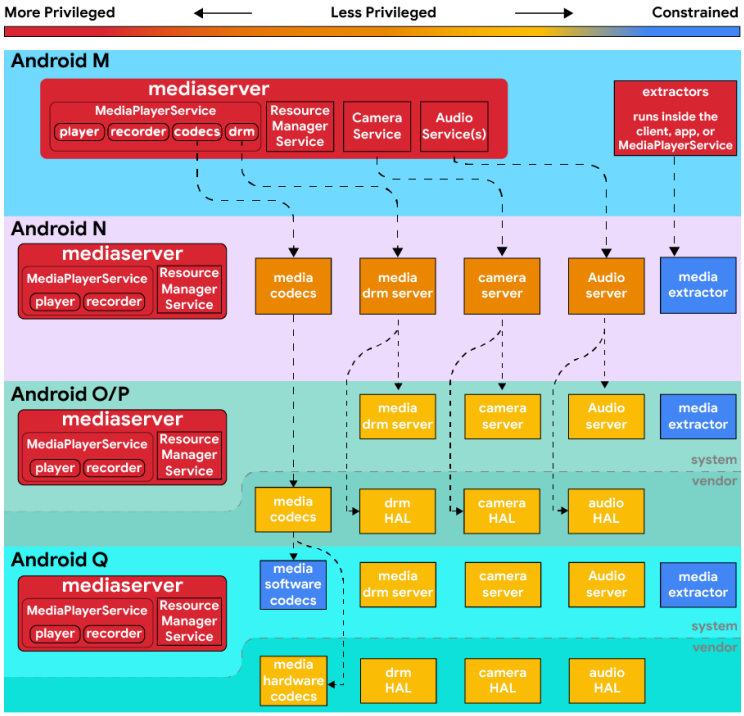

Fig. 1. Changes to mediaserver and codec sandboxing from Android 6 to Android 10

4.3.4 *Sandboxing the kernel*

Security hardening efforts in Android userspace have increasingly made the kernel a more attractive target for privilege escalation attacks [138]. Hardware drivers provided by System on a Chip (SoC) vendors account for the vast majority of kernel vulnerabilities on Android [140]. Reducing app/system access to these drivers was described above, but kernel-level drivers cannot be sandboxed within the kernel themselves, as Linux still is a monolithic kernel (vs. microkernel approaches). However, mitigation against exploiting weaknesses in all code running within kernel mode (including the core Linux kernel components and vendor drivers) was improved significantly over the various releases (cf. Table 3).

4.3.5 *Sandboxing below the kernel*

In addition to the kernel, the trusted computing base (TCB) on Android devices starts with the boot loader (which is typically split into multiple stages) and implicitly includes other components below the kernel, such as the trusted execution environment (TEE), hardware drivers, and userspace components `init`, `ueventd`, and `vold` [34]. It is clear that the sum of all these creates sufficient complexity that, given current state of the art, we have to assume bugs in some of them. For highly sensitive use cases, even the mitigations against kernel and system process bugs described above may not provide sufficient assurance against potential vulnerabilities.

Therefore, we explicitly consider the possibility of a kernel compromise (e.g. through directly attacking some kernel interfaces based on physical access in [T.P1], [T.P3], and [T.P4] or chaining together multiple bugs from user space code to reach kernel surfaces in [T.A7]), misconfiguration (e.g. with incorrect or overly permissive SELinux policies [60]), or bypass (e.g. by modifying the

Table 2. System sandboxing improvements in Android releases

Release	Improvement	Threats mitigated
4.4	SELinux in enforcing mode: MAC for 4 root processes `installd`, `netd`, `vold`, `zygote`.	[T.A1][T.A7] [T.D2]
5.x	SELinux: MAC for all userspace processes.	[T.A1][T.A7]
6.x	SELinux: MAC for all processes.	
7.x	Architectural decomposition of mediaserver.	[T.A1][T.A7] [T.D2]
7.x	`ioctl` system call restrictions for system components [137].	[T.A1][T.A7] [T.D2]
8.x	*Treble* Architectural decomposition: Move HALs (Hardware Abstraction Layer components) into separate processes, reduce permissions, restrict access to hardware drivers [58, 139].	[T.A1][T.A7] [T.D2]
10	Software codecs (the source of approximately 80% of the critical/high severity vulnerabilities in media components) were moved into a constrained sandbox.	[T.A7][T.D2]
10	Bounds Sanitizer (BoundSan): Missing or incorrect bounds checks on arrays accounted for 34% of Android's userspace security vulnerabilities. Clang's BoundSan adds bounds checking on arrays when the size can be determined at compile time. BoundSan was enabled across the Bluetooth stack and in 11 software codecs.	[T.A7][T.D2]
10	Integer Overflow Sanitizer (IOSAN): The process of applying IOSAN to the media frameworks began in Android 7.0 and was completed in Android 10.	[T.A7][T.D2]
10	Scudo is a dynamic heap allocator designed to be resilient against heap related vulnerabilities.	[T.A7][T.D2]

Table 3. Kernel sandboxing improvements in Android releases

Release	Improvement	Threats mitigated
5.x	Privileged eXecute Never (PXN) [144]: Disallow the kernel from executing userspace. Prevents *return-to-user (ret2usr)* style attacks.	[T.A7][T.D2]
6.x	Kernel threads moved into SELinux enforcing mode, limiting kernel access to userspace files.	[T.A7][T.D2]
8.x	Privileged Access Never (PAN) and PAN emulation: Prevent the kernel from accessing any userspace memory without going through hardened `copy-*-user()` functions [133].	[T.A7][T.D2]
9.0	Control Flow Integrity (CFI): Ensures that front-edge control flow stays within a precomputed graph of allowed function calls [134].	[T.A7][T.D2]
10	Shadow Call Stack (SCS): Protects the backwards edge of the call graph by protecting return addresses [135].	[T.A7][T.D2]

boot chain to boot a different kernel with deactivated security policies) as part of the threat model for some select scenarios. To be clear, with a compromised kernel, Android no longer meets the

compatibility requirements and many of the security and privacy assurances for users and apps no longer hold. However, we can still defend against some threats even under this assumption:

- **Keymaster** implements the Android key store in TEE to guard cryptographic key storage and use in the case of a run-time kernel compromise [28]. That is, even with a fully compromised kernel, an attacker cannot read key material stored in Keymaster[16]. Apps can explicitly request keys to be stored in Keymaster, i.e. to be hardware-bound, to be only accessible after user authentication (which is tied to Gatekeeper/Weaver), and/or request attestation certificates to verify these key properties [26], allowing verification of compatibility in terms of rule ③ (compatibility).
- **Strongbox**, specified starting with Android 9.0, implements the Android keystore in separate tamper resistant hardware (TRH) for even better isolation. This mitigates [T.P2] and [T.P3] against strong adversaries, e.g. against cold boot memory attacks [83] or hardware bugs such as Spectre/Meltdown [95, 104], Rowhammer [57, 136], or Clkscrew [129] that allow privilege escalation even from kernel to TEE. From a hardware perspective, the main application processor (AP) will always have a significantly larger attack surface than dedicated secure co-processor. Adding a separate TRH affords another sandboxing layer of defense in depth. The Google Pixel 3 was the first device to support Strongbox with a dedicated TRH (Titan M [146]), and other OEM devices have since started to implement it (often using standard secure elements that have been available on Android devices for NFC payment and other use cases).

> Note that only storing and using keys in TEE or TRH does not completely solve the problem of making them unusable under the assumption of a kernel compromise: if an attacker gains access to the low-level interfaces for communicating directly with Keymaster or Strongbox, they can use it as an oracle for cryptographic operations that require the private key. This is the reason why keys can be authentication bound and/or require user presence verification, e.g. by pushing a hardware button that is detectable by the TRH to assure that keys are not used in the background without user consent.

- **Gatekeeper** implements verification of user lock screen factors (PIN/password/pattern) in TEE and, upon successful authentication, communicates this to Keymaster for releasing access to authentication bound keys [27]. **Weaver** implements the same functionality in TRH and communicates with Strongbox. Specified for Android 9.0 and initially implemented on the Google Pixel 2 and newer phones, we also add a property called *Insider Attack Resistance* (IAR): without knowledge of the user's lock screen factor, an upgrade to the Weaver/Strongbox code running in TRH will wipe the secrets used for on-device encryption [109, 145]. That is, even with access to internal code signing keys, existing data cannot be exfiltrated without the user's cooperation.
- **Protected Confirmation**, also introduced with Android 9.0 [37], partially mitigates [T.A4] and [T.A6]. In its current scope, apps can tie usage of a key stored in Keymaster or Strongbox to the user confirming (by pushing a physical button) that they have seen a message displayed on the screen. Upon confirmation, the app receives a hash of the displayed message, which can be used to remotely verify that a user has confirmed the message. By controlling the screen output through TEE when protected confirmation is requested by an app, even a full kernel compromise (without user cooperation) cannot lead to creating these signed confirmations.

[16] Note: This assumes that hardware itself is still trustworthy. Side-channel attacks such as [99] are currently out of scope of this (software) platform security model, but influence some design decisions on the system level, e.g. to favor dedicated TRH over on-chip security partitioning.

4.4 Encryption of data at rest

A second element of enforcing the security model, particularly rules ① (multi-party consent) and ③ (compatibility), is required when the main system kernel is not running or is bypassed (e.g. by reading directly from non-volatile storage).

Full Disk Encryption (FDE) uses a credential protected key to encrypt the entire user data partition. FDE was introduced in Android 5.0, and while effective against [T.P2], it had a number of shortcomings. Core device functionality (such as emergency dialer, accessibility services, and alarms) were inaccessible until password entry. Multi-user support introduced in Android 6.0 still required the password of the primary user before disk access.

These shortcomings were mitigated by File Based Encryption (FBE) introduced in Android 7.0. On devices with TEE or TRH, all keys are derived within these secure environments, entangling the user knowledge factor with hardware-bound random numbers that are inaccessible to the Android kernel and components above.[17] FBE allows individual files to be tied to the credentials of different users, cryptographically protecting per-user data on shared devices [T.P4]. Devices with FBE also support a feature called *Direct Boot* which enables access to emergency dialer, accessibility services, alarms, and receiving calls all before the user inputs their credentials.

Android 10 introduced support for Adiantium [62], a new wide-block cipher mode based on AES, ChaCha, and Poly1305 to enable full device encryption without hardware AES acceleration support. While this does not change encryption of data at rest for devices with existing AES support, lower-end processors can now also encrypt all data without prohibitive performance impact. The significant implication is that all devices shipping originally with Android 10 are required to encrypt all data by default without any further exemptions, homogenizing the Android ecosystem in that aspect.

Note that encryption of data at rest helps significantly with enforcing rule ④ (safe reset), as effectively wiping user data only requires to delete the master key material, which is much quicker and not subject to the complexities of e.g. flash translation layer interactions.

4.5 Encryption of data in transit

Android assumes that all networks are hostile and could be injecting attacks or spying on traffic. In order to ensure that network level adversaries do not bypass app data protections, Android takes the stance that *all* network traffic should be end-to-end encrypted. Link level encryption is insufficient. This primarily protects against [T.N1] and [T.N2].

In addition to ensuring that connections use encryption, Android focuses heavily on ensuring that the encryption is used correctly. While TLS options are secure by default, we have seen that it is easy for developers to incorrectly customize TLS in a way that leaves their traffic vulnerable to OPA/MITM [71, 72, 81]. Table 4 lists recent improvements in terms of making network connections safe by default.

4.6 Exploit mitigation

A robust security system should assume that software vulnerabilities exist and actively defend against them. Historically, about 85% of security vulnerabilities on Android result from unsafe memory access (cf. [96, slide 54]). While this section primarily describes mitigations against memory unsafety ([T.P1-P4], [T.N2], [T.A1-A3,A7], [T.D2]) we note that the best defense is the memory safety offered by languages such as Java or Kotlin. Much of the Android framework is written in Java, effectively defending large swathes of the OS from entire categories of security bugs.

[17] A detailed specification and analysis of key entanglement is subject to related work and currently in progress. A reference to this detail will be added to a later version of this paper.

Table 4. Network sandboxing improvements in Android releases

Release	Improvement	Threats mitigated
6.x	`usesCleartextTraffic` in manifest to prevent unintentional cleartext connections [54].	[T.N1][T.N2]
7.x	Network security config [30] to declaratively specify TLS and cleartext settings on a per-domain or app-wide basis to customize TLS connections.	[T.N1][T.N2]
9.0	DNS-over-TLS [94] to reduce sensitive data sent over cleartext and made apps opt-in to using cleartext traffic in their network security config.	[T.N1][T.N2]
9.0	TLS is the default for all connections [55].	[T.N1][T.N2]
10	MAC randomization is enabled by default for client mode, SoftAP, and WiFi Direct [31].	[T.N1]
10	TLS 1.3 support.	[T.N1][T.N2]

Android mandates the use of a number of mitigations including ASLR [53, 124], RWX memory restrictions (e.g. $W \oplus X$, cf. [123]), and buffer overflow protections (such as stack-protector for the stack and allocator protections for the heap). Similar protections are mandated for Android kernels [133].

In addition to the mitigations listed above, Android is selectively enabling new mitigations, focusing first on code areas which are remotely reachable (e.g. the media frameworks [44]) or have a history of high severity security vulnerabilities (e.g. the kernel). Android has pioneered the use of LLVM undefined behavior sanitizer (UBSAN) and other address sanitizers [122] in production devices to protect against integer overflow vulnerabilities in the media frameworks and other security sensitive components. Android is also rolling out Control Flow Integrity (CFI) [134] in the kernel and security sensitive userspace components including media, Bluetooth, WiFi, NFC, and parsers [106] in a fine-grained variant as implemented by current LLVM [132] that improves upon previous, coarse-grained approaches that have been shown to be ineffective [64]. Starting with Android 10, the common Android kernel as well as parts of the Bluetooth stack can additionally be protected against backwards-edge exploitation through the use of Shadow Call Stack (SCS), again as implemented by current LLVM [127] as the best trade-off between performance overhead and effectiveness [56].

These code and runtime safety mitigation methods work in tandem with isolation and containment mechanisms (cf. tables 1 to 3 for added mitigations over time) to form many layers of defense; even if one layer fails, other mechanisms aim to prevent a successful exploitation chain. Mitigation mechanisms also help to uphold rules ② (open ecosystem) and ③ (compatibility) without placing additional assumptions on which languages apps are written in.

However, there are other types of exploits than apps directly trying to circumvent security controls of the platform or other apps: malicious apps can try to mislead users through deceptive UI tactics to either receive technical consent grants against users' interests (including clickjacking [80]) ([T.A4-A6], [T.D1]), existing legitimate apps can be repackaged together with malicious code ([T.A1-A2]), or look-alike and similarly named apps could try to get users to install them instead of other well-known apps. Such user deception is not only a problem in the Android ecosystem but more generally of any UI-based interaction. As deception attacks tend to develop and change quickly, platform mitigations are often too slow to roll out, making dynamic blocking more effective. Within

the Android ecosystem, mitigations against such kinds of exploits are therefore based on multiple mechanisms, notably submission-time checks on Goodle Play and on-device run-time checks with Play Google Protect. Nonetheless, platform security has adapted over time to make certain classes of UI deception exploits harder or impossible, e.g. through restricting SYSTEM_ALERT_WINDOW, background activity limitations, or scoped external storage (cf. table 1).

4.7 System integrity

Finally, system (sometimes also referred to as device) integrity is an important defense against attackers gaining a persistent foothold. AOSP has supported *Verified Boot* using the Linux kernel `dm-verity` support since Android KitKat, providing strong integrity enforcement for the Trusted Computing Base (TCB) and system components to implement rule ④ (safe reset). Verified Boot [35] has been mandated since Android Nougat (with an exemption granted to devices which cannot perform AES crypto above 50MiB/sec. up to Android 8, but no exemptions starting with Android 9.0) and makes modifications to the boot chain detectable by verifying the boot, TEE, and additional vendor/OEM partitions, as well as performing on-access verification of blocks on the system partition [38]. That is, attackers cannot permanently modify the TCB even after all previous layers of defense have failed, leading to a successful kernel compromise. Note that this assumes the primary boot loader as root of trust to still be intact. As this is typically implemented in a ROM mask in sufficiently simple code, critical bugs at that stage are less likely.

Additionally, rollback protection with hardware support (counters stored in tamper-proof persistent storage, e.g. a separate TRH as used for Strongbox or enforced through RPMB as implemented in a combination of TEE and eMMC controller [18]) prevents attacks from flashing a properly signed but outdated system image that has known vulnerabilities and could be exploited. Finally, the Verified Boot state is included in key attestation certificates (provided by Keymaster/Strongbox) in the `deviceLocked` and `verifiedBootState` fields, which can be verified by apps as well as passed onto backend services to remotely verify boot integrity [36] to support rule ③ (compatibility).

Starting with Android 10 on some devices supporting the latest Android Verified Boot (AVB, the recommended default implementation for verifying the integrity of read-only partitions [38]) version 2, the VBMeta struct digest (a top-level hash over all parts) is included in these key attestation certificates to support firmware transparency by verifying that digest match released firmware images [109**?**]. In combination with server side validation, this can be used as a form of remote system integrity attestation akin to PCR verification with trusted platform modules (TPMs). Integrity of firmware for other CPUs (including, but not limited to, the various radio chipsets, the GPU, touch screen controllers, etc.) is out of scope of AVB at the time of this writing, and is typically handled by OEM-specific boot loaders.

4.7.1 Verification key hierarchy and updating

While the details for early boot stages are highly dependent on the respective chipset hardware and low-level boot loaders, Android devices generally use at least the following keys for verifying system integrity:

(1) The first (and potentially multiple intermediate) boot loader(s) is/are signed by a key $\mathbf{K_A}$ held by the hardware manufacturer and verified through a public key embedded in the chipset ROM mask. This key cannot be changed.
(2) The (final) bootloader responsible for loading the Android Linux kernel is verified through a key $\mathbf{K_B}$ embedded in a previous bootloader. Updating this signing key is chipset specific, but may be possible in the field by updating a previous, intermediate bootloader block. Android 10 strongly recommends that this bootloader use the reference implementation

of Android Verified Boot [**?**] and VBMeta structs for verifying all read-only (e.g. `system`, `vendor`, etc.) partitions.

(3) A VBMeta signing key $\mathbf{K}_C$ is either directly embedded in the final bootloader or retrieved from a separate TRH to verify flash partitions before loading the kernel. AVB implementations may also allow a user-defined VBMeta signing key $\mathbf{K}'_C$ to be set (typically in a TEE or TRH) — in this case, the Verified Boot state will be set to YELLOW to indicate that non-manufacturer keys were use to sign the partitions, but that verification with the user-defined keys has still been performed correctly (see Figure 2).
Updating this key $\mathbf{K}_C$ used to sign any partitions protected through AVB is supported through the use of chained partitions in the VBMeta struct (resulting in partition-specific signing keys $\mathbf{K}^i_D$ for partition i that are in turn signed by $\mathbf{K}_C/\mathbf{K}'_C$), by updating the key used to sign the VBMeta struct itself (through flashing a new version of the final bootloader in an over-the-air update), or – in the case of user-defined keys – using direct physical access[18].

(4) The digest(s) embedded in VMBeta struct(s) are used by the Android Linux kernel to verify blocks within persistent, read-only partitions on-access using `dm-verity` (or for small partitions, direct verification before loading them atomically into memory). Inside the `system` partition, multiple public signing keys are used for different purposes, e.g. the platform signing key mentioned in section 4.3.1 or keys used to verify the download of over-the-air (OTA) update packages before applying them. Updating those keys is trivial through simply flashing a new `system` partition.

(5) All APKs are individually signed by the respective developer key $\mathbf{K}^j_E$ for APK j (some may be signed by the platform signing key to be granted `signature` permissions for those components), which in turn are stored on the `system` or `data` partition. Integrity of updateable (system or user installed) apps is enforced via APK signing [39] and is checked by Android's `PackageManager` during installation and update. Every app is signed and an update can only be installed if the new APK is signed with the same identity or by an identity that was delegated by the original signer.
For run-time updateable apps, the APK Signature Scheme version 3 was introduced with Android 9.0 to support rotation of these individual signing keys [39].

4.8 Patching

Orthogonal to all the previous defense mechanisms, vulnerable code should be fixed to close discovered holes in any of the layers. Regular patching can be seen as another layer of defense. However, shipping updated code to the huge and diverse Android ecosystem is a challenge [131] (which is one of the reasons for applying the defense in depth strategy).

Starting in August 2015, Android has publicly released a monthly security bulletin and patches for security vulnerabilities reported to Google. To address ecosystem diversity, project *Treble* [147] launched with Android 8.0, with a goal of reducing the time/cost of updating Android devices [107, 111] and implemented through decoupling of the main system image from hardware-dependent chipset vendor/OEM customization. This modularization introduced a set of security-relevant changes:

- The SELinux policy is no longer monolithic, but assembled at boot time from different partitions (currently `system` and `vendor`). Updating the policy for platform or hardware components can therefore be done independently through changes within the relevant partition [40, 58].

[18] E.g. Pixel devices support this through `fastboot flash avb_custom_key` as documented online at https://source.android.com/security/verifiedboot/device-state.

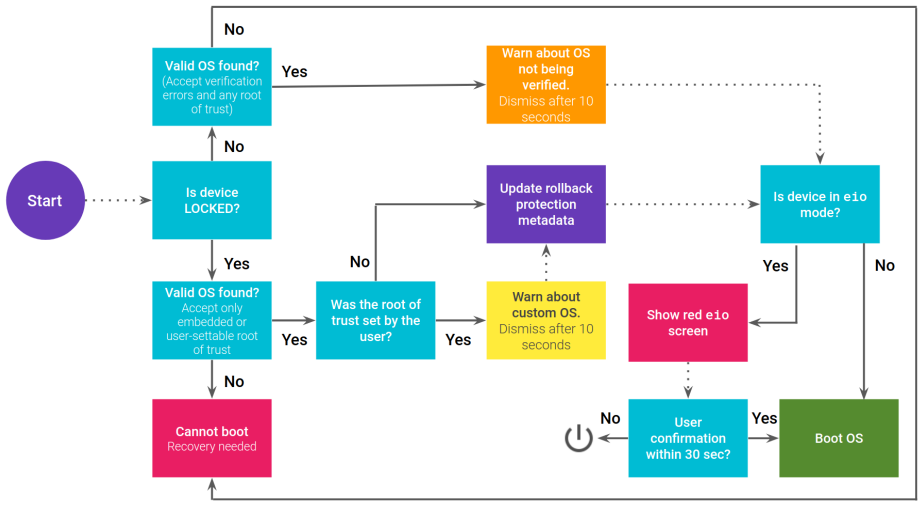

Fig. 2. Verified Boot flow and different states: (YELLOW): warning screen for LOCKED devices with custom root of trust set; (ORANGE): warning screen for UNLOCKED devices; (RED): warning screen for dm-verity corruption or no valid OS found [22].

- Each HAL component (mainly native daemons) runs in its own sandbox and is permitted access to only the hardware driver it controls; higher-level system processes accessing this hardware component are now limited to accessing this HAL instead of directly interacting with the hardware driver [139].

As part of project Treble, approximately 20 HALs were moved out of system server, including the HALs for sensors, GPS, fingerprint, WiFi, and more. Previously, a compromise in any of those HALs would gain privileged system permissions, but in Android 8.0, permissions are restricted to the subset needed by the specific HAL. Similarly, HALs for audio, camera, and DRM have been moved out of audioserver, cameraserver, and drmserver respectively.

In 2018, the Android Enterprise Recommended program as well as general agreements with OEMs added the requirement of 90-day guaranteed security updates [20].

Starting with Android 10, some core system components can be updated through Google Play Store as standard APK files or – if required early in the boot process or involving native system libraries/services – as an APEX loopback filesystems in turn protected through `dm-verity` [82].

5 SPECIAL CASES

There are some special cases that require intentional deviations from the abstract security model to balance specific needs of various parties. This section describes some of these but is not intended to be a comprehensive list. One goal of defining the Android security model publicly is to enable researchers to discover potential additional gaps by comparing the implementation in AOSP with the model we describe, and to engage in conversation on those special cases.

- **Listing packages:** The ability for one app to discover what other apps are installed on the device can be considered a potential information leak and violation of user consent (rule ①). However, app discovery is necessary for some direct app-to-app interaction which is derived from the open ecosystem principle (rule ②). As querying the list of all installed apps is potentially privacy sensitive and has been abused by malware, Android 11 supports

more specific app-to-app interaction using platform components and limits general package visibility for apps targeting this API version. While this special case is still supported at the time of this writing, it will require the new `QUERY_ALL_PACKAGES` and may be limited further in the future.

- **VPN apps may monitor/block network traffic for other apps:** This is generally a deviation from the application sandbox model since one app may see and impact traffic from another app (*developer* consent). VPN apps are granted an exemption because of the value they offer users, such as improved privacy and data usage controls, and because *user* consent is clear. For applications which use end-to-end encryption, clear-text traffic is not available to the VPN application, partially restoring the confidentiality of the application sandbox.
- **Backup:** Data from the private app directory is backed up by default. Android 9 added support for end-to-end encryption of backups to the Google cloud by entangling backup session keys with the user lockscreen knowledge factor (LSKF) [91]. Apps may opt out by setting fields in their manifest.
- **Enterprise:** Android allows so-called Device Owner (DO) or Profile Owner (PO) policies to be enforced by a Device Policy Controller (DPC) app. A DO is installed on the primary/main user account, while a PO is installed on a secondary user that acts as a work profile. Work profiles allow separation of personal from enterprise data on a single device and are based on Android multi-user support. This separation is enforced by the same isolation and containment methods that protect apps from each other but implement a significantly stricter divide between the profiles [6].
  A DPC introduces a fourth party to the consent model: only if the policy allows an action (e.g. within the work profile controlled by a PO) in addition to consent by all other parties can it be executed. The distinction of personal and work profile is enhanced by the recent support of different user knowledge factors (handled by the lockscreen as explained above in subsection 4.2), which lead to different encryption keys for FBE. Note that on devices with a work profile managed by PO but no full-device control (i.e. no DO), privacy guarantees for the personal profile still need to hold under this security model.
- **Factory Reset Protection (FRP):** is an exception to not storing any persistent data across factory reset (rule ④), but is a deliberate deviation from this part of the model to mitigate the threat of theft and factory reset ([T.P2][T.P3]).

6 RELATED WORK

Classical operating system security models are primarily concerned with defining access control (read/write/execute or more finely granular) by subjects (but most often single users, groups, or roles) to objects (typically files and other resources controlled by the OS, in combination with permissions sometimes also called protection domains [128]). The most common data structures for efficiently implementing these relations (which, conceptually, are sparse matrices) are Access Control Lists (ACLs) [120] and capability lists (e.g. [142]). One of the first well-known and well-defined models was the Bell-LaPadula multi-level security model [48], which defined properties for assigning permissions and can be considered the abstract basis for Mandatory Access Control and Type Enforcement schemes like SELinux. Consequently, the Android platform security model implicitly builds upon these general models and their principle of least privilege.

One fundamental difference is that, while classical models assume processes started by a user to be a proxy for their actions and therefore execute directly with user privileges, more contemporary models explicitly acknowledge the threat of malware started by a user and therefore aim to compartmentalize their actions. Many mobile OS (including Symbian as an earlier example) assign permissions to processes (i.e. applications) instead of users, and Android uses a comparable approach.

A more detailed comparison to other mobile OS is out of scope in this paper, and we refer to other surveys [68, 98, 112] as well as previous analysis of Android security mechanisms and how malware exploited weaknesses [14, 70, 74, 101–103, 149].

7 CONCLUSION

In this paper, we described the Android platform security model and the complex threat model and ecosystem it needs to operate in. One of the abstract rules is a multi-party consent model that is different to most standard OS security models in the sense that it implicitly considers applications to have equal veto rights over actions in the same sense that the platform implementation and, obviously, users have. While this may seem restricting from a user point of view, it effectively limits the potential abuse a malicious app can do on data controlled by other apps; by avoiding an all-powerful user account with unfiltered access to all data (as is the default with most current desktop/server OS), whole classes of threats such as file encrypting ransomware or direct data exfiltration become impractical.

AOSP implements the Android platform security model as well as the general security principles of 'defense in depth' and 'safe by default'. Different security mechanisms combine as multiple layers of defense, and an important aspect is that even if security relevant bugs exist, they should not necessarily lead to exploits reachable from standard user space code. While the current model and its implementation already cover most of the threat model that is currently in scope of Android security and privacy considerations, there are some deliberate special cases to the conceptually simple security model, and there is room for future work:

- Keystore already supports API flags/methods to request hardware- or authentication-bound keys. However, apps need to use these methods explicitly to benefit from improvements like Strongbox. Making encryption of app files or directories more transparent by supporting declarative use similar to network security config for TLS connections would make it easier for app developers to securely use these features.
- It is common for malware to dynamically load its second stage depending on the respective device it is being installed on, to both try to exploit specific detected vulnerabilities and hide its payload from scanning in the app store. One potential mitigation is to require all executable code to: a) be signed by a key that is trusted by the respective Android instance (e.g. with public keys that are pre-shipped in the firmware and/or can be added by end-users) or b) have a special permission to dynamically load/create code during runtime that is not contained in the application bundle itself (the APK file). This could give better control over code integrity, but would still not limit languages or platforms used to create these apps. It is recognized that this mitigation is limited to executable code. Interpreted code or server based configuration would bypass this mitigation.
- Advanced attackers may gain access to OEM or vendor code signing keys. Even under such circumstance, it is beneficial to still retain some security and privacy assurances to users. One recent example is the specification and implementation of *Insider Attack Resistance* (IAR) for updateable code in TRH [145], and extending similar defenses to higher-level software is desirable [109]. Potential approaches could be reproducible firmware builds or logs of released firmware hashes comparable to e.g. Certificate Transparency [100].
- Hardware level attacks are becoming more popular, and therefore additional (software and hardware) defense against e.g. RAM related attacks would add another layer of defense, although, most probably with a trade-off in performance overhead.

However, all such future work needs to be done considering its impact on the wider ecosystem and should be kept in line with fundamental Android security rules and principles.

ACKNOWLEDGMENTS

We would like to thank Dianne Hackborn for her influential work over a large part of the Android platform security history and insightful remarks on earlier drafts of this paper. Additionally, we thank Joel Galenson, Ivan Lozano, Paul Crowley, Shawn Willden, Jeff Sharkey, Billy Lau, Haining Chen, and Xiaowen Xin for input on various parts, and particularly Vishwath Mohan for direct contributions to the Authentication section. We also thank the enormous number of security researchers (https://source.android.com/security/overview/acknowledgements) who have improved Android over the years and anonymous reviewers who have contributed highly helpful feedback to earlier drafts of this paper.

REFERENCES

[1] 2015. Stagefright Vulnerability Report. https://www.kb.cert.org/vuls/id/924951
[2] 2017. BlueBorne. https://go.armis.com/hubfs/BlueBorne%20-%20Android%20Exploit%20(20171130).pdf?t=1529364695784
[3] 2017. CVE-2017-13177. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13177
[4] 2018. https://www.stonetemple.com/mobile-vs-desktop-usage-study/
[5] 2018. http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet
[6] 2018. Android Enterprise Security White Paper. https://source.android.com/security/reports/Google_Android_Enterprise_Security_Whitepaper_2018.pdf
[7] 2018. Android Security 2017 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf
[8] 2018. CVE-2017-17558: Remote code execution in media frameworks. https://source.android.com/security/bulletin/2018-06-01#kernel-components
[9] 2018. CVE-2018-9341: Remote code execution in media frameworks. https://source.android.com/security/bulletin/2018-06-01#media-framework
[10] 2018. SVE-2018-11599: Theft of arbitrary files leading to emails and email accounts takeover. https://security.samsungmobile.com/securityUpdate.smsb
[11] 2018. SVE-2018-11633: Buffer Overflow in Trustlet. https://security.samsungmobile.com/securityUpdate.smsb
[12] 2019. Android Now FIDO2 Certified. https://fidoalliance.org/android-now-fido2-certified-accelerating-global-migration-beyond-passwords/
[13] 2020. Personal identification – ISO-compliant driving licence – Part 5: Mobile driving licence (mDL) application. Draft International Standard: ISO/IEC DIS 18013-5.
[14] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. 2016. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *2016 IEEE Symposium on Security and Privacy (SP)*. 433–451. https://doi.org/10.1109/SP.2016.33
[15] Anne Adams and Martina Angela Sasse. 1999. Users Are Not the Enemy. *Commun. ACM* 42, 12 (Dec. 1999), 40–46. https://doi.org/10.1145/322796.322806
[16] Andrew Ahn. 2018. How we fought bad apps and malicious developers in 2017. https://android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-and-malicious.html
[17] Bonnie Brinton Anderson, Anthony Vance, C. Brock Kirwan, Jeffrey L. Jenkins, and David Eargle. 2016. From Warning to Wallpaper: Why the Brain Habituates to Security Warnings and What Can Be Done About It. *Journal of Management Information Systems* 33, 3 (2016), 713–743. https://doi.org/10.1080/07421222.2016.1243947
[18] Anil Kumar Reddy, P. Paramasivam, and Prakash Babu Vemula. 2015. Mobile secure data protection using eMMC RPMB partition. In *2015 International Conference on Computing and Network Communications (CoCoNet)*. 946–950. https://doi.org/10.1109/CoCoNet.2015.7411305
[19] AOSP. [n. d.]. Android Compatibility Definition Document. https://source.android.com/compatibility/cdd
[20] AOSP. [n. d.]. Android Enterprise Recommended requirements. https://www.android.com/enterprise/recommended/requirements/
[21] AOSP. [n. d.]. Android platform permissions requesting guidance. https://material.io/design/platform-guidance/android-permissions.html#request-types
[22] AOSP. [n. d.]. Android Verified Boot Flow. https://source.android.com/security/verifiedboot/boot-flow
[23] AOSP. [n. d.]. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro
[24] AOSP. [n. d.]. App Manifest permission element. https://developer.android.com/guide/topics/manifest/permission-element

[25] AOSP. [n. d.]. Developer documentation android.security.identity. https://developer.android.com/reference/android/security/identity/package-summary
[26] AOSP. [n. d.]. Developer documentation android.security.keystore.KeyGenParameterSpec. https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec
[27] AOSP. [n. d.]. Gatekeeper. https://source.android.com/security/authentication/gatekeeper
[28] AOSP. [n. d.]. Hardware-backed Keystore. https://source.android.com/security/keystore/
[29] AOSP. [n. d.]. Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters
[30] AOSP. [n. d.]. Network security configuration. https://developer.android.com/training/articles/security-config
[31] AOSP. [n. d.]. Privacy: MAC Randomization. https://source.android.com/devices/tech/connect/wifi-mac-randomization
[32] AOSP. [n. d.]. Privileged Permission Allowlisting. https://source.android.com/devices/tech/config/perms-whitelist
[33] AOSP. [n. d.]. Restrictions on non-SDK interfaces. https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces
[34] AOSP. [n. d.]. Security Updates and Resources - Process Types. https://source.android.com/security/overview/updates-resources#process_types
[35] AOSP. [n. d.]. Verifying Boot. https://source.android.com/security/verifiedboot/verified-boot
[36] AOSP. [n. d.]. Verifying hardware-backed key pairs with Key Attestation. https://developer.android.com/training/articles/security-key-attestation
[37] AOSP. 2018. Android Protected Confirmation. https://developer.android.com/preview/features/security#android-protected-confirmation
[38] AOSP. 2018. Android Verified Boot 2.0. https://android.googlesource.com/platform/external/avb/+/pie-release/README.md
[39] AOSP. 2018. APK Signature Scheme v3. https://source.android.com/security/apksigning/v3
[40] AOSP. 2018. SELinux for Android 8.0: Changes & Customizations. https://source.android.com/security/selinux/images/SELinux_Treble.pdf
[41] AOSP. 2019. Restrictions on starting activities from the background. https://developer.android.com/guide/components/activities/background-starts
[42] AOSP. 2020. Android 11 biometric authentication. https://developer.android.com/about/versions/11/features#biometric-auth
[43] AOSP. 2020. Security and Privacy Enhancements in Android 10. https://source.android.com/security/enhancements/enhancements10
[44] Dan Austin and Jeff Vander Stoep. 2016. Hardening the media stack. https://android-developers.googleblog.com/2016/05/hardening-media-stack.html
[45] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. 2010. Smudge Attacks on Smartphone Touch Screens. In *Proceedings of the 4th USENIX Conference on Offensive Technologies (WOOT'10)*. USENIX Association, USA, 1–7.
[46] Steve Barker. 2009. The next 700 Access Control Models or a Unifying Meta-Model?. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies (SACMAT '09)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/1542207.1542238
[47] David Barrera, Daniel McCarney, Jeremy Clark, and Paul C. van Oorschot. 2014. Baton: Certificate Agility for Android's Decentralized Signing Infrastructure. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2627393.2627397
[48] D. Bell and L. LaPadula. 1975. *Secure Computer System Unified Exposition and Multics Interpretation.* Technical Report MTR-2997. MITRE Corp., Bedford, MA.
[49] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. 1998. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology*. Springer, 26–45.
[50] M. Benantar. 2005. *Access Control Systems: Security, Identity Management and Trust Models.*
[51] James Bender. 2018. Google Play security metadata and offline app distribution. https://android-developers.googleblog.com/2018/06/google-play-security-metadata-and.html
[52] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. 2003. A Logical Framework for Reasoning about Access Control Models. *ACM Trans. Inf. Syst. Secur.* 6, 1 (Feb. 2003), 71–127. https://doi.org/10.1145/605434.605437
[53] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proc. USENIX Security Symposium - Volume 12*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=1251353.1251361
[54] Chad Brubaker. 2014. Introducing nogotofail — a network traffic security testing tool. https://security.googleblog.com/2014/11/introducing-nogotofaila-network-traffic.html

[55] Chad Brubaker. 2018. Protecting users with TLS by default in Android P. https://android-developers.googleblog.com/2018/04/protecting-users-with-tls-by-default-in.html

[56] N. Burow, X. Zhang, and M. Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. 985–999. https://doi.org/10.1109/SP.2019.00076

[57] Pierre Carru. 2017. Attack TrustZone with Rowhammer. http://www.eshard.com/wp-content/plugins/email-before-download/download.php?dl=9465aa084ff0f070a3acedb56bcb34f5

[58] Dan Cashman. 2017. SELinux in Android O: Separating Policy to Allow for Independent Updates. https://events.static.linuxfound.org/sites/events/files/slides/LSS%20-%20Treble%20%27n%27%20SELinux.pdf Linux Security Summit.

[59] Jen Chai. 2019. Giving users more control over their location data. https://android-developers.googleblog.com/2019/03/giving-users-more-control-over-their.html

[60] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. 2017. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, New York, NY, USA, 553–565. https://doi.org/10.1145/3134600.3134638

[61] Jiska Classen and Matthias Hollick. 2019. Inside job: diagnosing bluetooth lower layers using off-the-shelf devices. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2019, Miami, Florida, USA, May 15-17, 2019*. ACM, 186–191. https://doi.org/10.1145/3317549.3319727

[62] Paul Crowley and Eric Biggers. 2018. Adiantum: length-preserving encryption for entry-level processors. *IACR Transactions on Symmetric Cryptology* 2018, 4 (Dec. 2018), 39–61. https://doi.org/10.13154/tosc.v2018.i4.39-61

[63] Edward Cunningham. 2017. Improving app security and performance on Google Play for years to come. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html

[64] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 401–416. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

[65] Sabrina De Capitani di Vimercati. 2011. *Access Matrix*. Springer US, Boston, MA, 14–17. https://doi.org/10.1007/978-1-4419-5906-5_807

[66] Rachna Dhamija, J. D. Tygar, and Marti Hearst. 2006. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. ACM, New York, NY, USA, 581–590. https://doi.org/10.1145/1124772.1124861

[67] Danny Dolev and Andrew Chi chih Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29 (1983), 198–208. Issue 2. https://doi.org/10.1109/TIT.1983.1056650

[68] Andre Egners, Björn Marschollek, and Ulrike Meyer. 2012. *Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms*. Technical Report 2012,7. RWTH Aachen University. https://itsec.rwth-aachen.de/publications/ae_hacker_in_your_pocket.pdf

[69] Malin Eiband, Mohamed Khamis, Emanuel von Zezschwitz, Heinrich Hussmann, and Florian Alt. 2017. Understanding Shoulder Surfing in the Wild: Stories from Users and Observers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 4254–4265. https://doi.org/10.1145/3025453.3025636

[70] W. Enck, M. Ongtang, and P. McDaniel. 2009. Understanding Android Security. *IEEE Security Privacy* 7, 1 (Jan 2009), 50–57. https://doi.org/10.1109/MSP.2009.26

[71] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[72] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2508859.2516655

[73] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in Smartphone Usage. In *Proc. 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, NY, USA, 179–194. https://doi.org/10.1145/1814433.1814453

[74] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. 2015. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys Tutorials* 17, 2 (2015), 998–1022. https://doi.org/10.1109/COMST.2014.2386139

[75] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David A. Wagner. 2012. How to Ask for Permission. In *HotSec*.

[76] 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). ACM, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/2335356.2335360

[77] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Financial Cryptography and Data Security (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 41–59. https://doi.org/10.1007/978-3-662-54970-4_3

[78] Nate Fischer. 2018. Protecting WebView with Safe Browsing. https://android-developers.googleblog.com/2018/04/protecting-webview-with-safe-browsing.html

[79] Google APIs for Android. [n. d.]. https://developers.google.com/android/reference/com/google/android/gms/fido/Fido

[80] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[81] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*. 38–49.

[82] Anwar Ghuloum. 2019. Fresher OS with Projects Treble and Mainline. https://android-developers.googleblog.com/2019/05/fresher-os-with-projects-treble-and-mainline.html

[83] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Commun. ACM* 52, 5 (May 2009), 91–98. https://doi.org/10.1145/1506409.1506429

[84] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. 2020. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 271–287. https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

[85] Daniel Hintze, Rainhard D. Findling, Muhammad Muaaz, Sebastian Scholz, and René Mayrhofer. 2014. Diversity in Locked and Unlocked Mobile Device Usage. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UbiComp 2014)*. ACM Press, 379–384. https://doi.org/10.1145/2638728.2641697

[86] Daniel Hintze, Rainhard D. Findling, Sebastian Scholz, and René Mayrhofer. 2014. Mobile Device Usage Characteristics: The Effect of Context and Form Factor on Locked and Unlocked Usage. In *Proc. MoMM 2014: 12th International Conference on Advances in Mobile Computing and Multimedia*. ACM Press, New York, NY, USA, 105–114. https://doi.org/10.1145/2684103.2684156

[87] Daniel Hintze, Philipp Hintze, Rainhard Dieter Findling, and René Mayrhofer. 2017. A Large-Scale, Long-Term Analysis of Mobile Device Usage Characteristics. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 2, Article 13 (June 2017), 21 pages. https://doi.org/10.1145/3090078

[88] Sebastian Höbarth and René Mayrhofer. 2011. A framework for on-device privilege escalation exploit execution on Android. In *Proc. IWSSI/SPMU 2011: 3rd International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use, colocated with Pervasive 2011*.

[89] Michael Hölzl, Michael Roland, and René Mayrhofer. 2017. Real-world Identification for an Extensible and Privacy-preserving Mobile eID. In *Privacy and Identity Management. The Smart Revolution. Privacy and Identity 2017*. IFIP AICT, Vol. 526/2018. Springer, Ispra, Italy, 354–370. https://doi.org/10.1007/978-3-319-92925-5_24

[90] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. 2014. A11Y Attacks: Exploiting Accessibility in Operating Systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 103–115. https://doi.org/10.1145/2660267.2660295

[91] Troy Kensinger. 2018. Google and Android have your back by protecting your backups. https://security.googleblog.com/2018/10/google-and-android-have-your-back-by.html

[92] Hassan Khan, Urs Hengartner, and Daniel Vogel. 2018. Evaluating Attack and Defense Strategies for Smartphone PIN Shoulder Surfing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3173574.3173738

[93] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.). Springer International Publishing, Cham, 3–24.

[94] Erik Kline and Ben Schwartz. 2018. DNS over TLS support in Android P Developer Preview. https://android-developers.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html

[95] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203 [cs]* (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[96] Nick Kralevich. 2016. The Art of Defense: How vulnerabilities help shape security features and mitigations in Android. https://www.blackhat.com/docs/us-16/materials/us-16-Kralevich-The-Art-Of-Defense-How-Vulnerabilities-Help-Shape-\Security-Features-And-Mitigations-In-Android.pdf BlackHat.

[97] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. 2014. On Malware Leveraging the Android Accessibility Framework. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Ivan Stojmenovic, Zixue Cheng, and Song Guo (Eds.). Springer International Publishing, Cham, 512–523.

[98] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. 2013. A Survey on Security for Mobile Devices. 15 (01 2013), 446–471.

[99] Ben Lapid and Avishai Wool. 2019. Cache-Attacks on the ARM TrustZone Implementations of AES-256 and AES-256-GCM via GPU-Based Analysis. In *Selected Areas in Cryptography – SAC 2018*, Carlos Cid and Michael J. Jacobson Jr. (Eds.). Springer International Publishing, Cham, 235–256.

[100] B. Laurie, A. Langley, and E. Kasper. 2013. Certificate Transparency. https://www.rfc-editor.org/info/rfc6962

[101] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *arXiv:1404.7431 [cs]* (April 2014). http://arxiv.org/abs/1404.7431

[102] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67 – 95. https://doi.org/10.1016/j.infsof.2017.04.001

[103] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer. 2014. ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. 3–17. https://doi.org/10.1109/BADGERS.2014.7

[104] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv:1801.01207 [cs]* (2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[105] T. Lodderstedt, M. McGloin, and P. Hunt. 2013. OAuth 2.0 Threat Model and Security Considerations. https://www.rfc-editor.org/info/rfc6819

[106] Ivan Lozano. 2018. Compiler-based security mitigations in Android P. https://android-developers.googleblog.com/2018/06/compiler-based-security-mitigations-in.html

[107] Iliyan Malchev. 2017. Here comes Treble: A modular base for Android. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html

[108] René Mayrhofer. 2014. An Architecture for Secure Mobile Devices. *Security and Communication Networks* (2014). https://doi.org/10.1002/sec.1028

[109] René Mayrhofer. 2019. Insider Attack Resistance in the Android Ecosystem.

[110] René Mayrhofer, Vishwath Mohan, and Stephan Sigg. 2020. Adversary Models for Mobile Device Authentication. arXiv:cs.CR/2009.10150

[111] T. McDonnell, B. Ray, and M. Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance*. 70–79. https://doi.org/10.1109/ICSM.2013.18

[112] I. Mohamed and D. Patel. 2015. Android vs iOS Security: A Comparative Study. In *2015 12th International Conference on Information Technology - New Generations*. 725–730. https://doi.org/10.1109/ITNG.2015.123

[113] Vishwath Mohan. 2018. Better Biometrics in Android P. https://android-developers.googleblog.com/2018/06/better-biometrics-in-android-p.html

[114] Vikrant Nanda and René Mayrhofer. 2018. Android Pie à la mode: Security & Privacy. https://android-developers.googleblog.com/2018/12/android-pie-la-mode-security-privacy.html

[115] Sundar Pichai. 2018. Android has created more choice, not less. https://blog.google/around-the-globe/google-europe/android-has-created-more-choice-not-less/

[116] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 603–620. https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[117] Peter Riedl, Rene Mayrhofer, Andreas Möller, Matthias Kranz, Florian Lettner, Clemens Holzmann, and Marion Koelle. 2015. Only play in your comfort zone: interaction methods for improving security awareness on mobile devices. *Personal and Ubiquitous Computing* (27 March 2015), 1–14. https://doi.org/10.1007/s00779-015-0840-5

[118] Franziska Roesner, Tadayoshi Kohno, Er Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. 2012. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy, ser. SP'12*. 224–238. https://doi.org/10.1109/SP.2012.24

[119] Michael Roland, Josef Langer, and Josef Scharinger. 2013. Applying Relay Attacks to Google Wallet. In *Proceedings of the Fifth International Workshop on Near Field Communication (NFC 2013)*. IEEE, Zurich, Switzerland. https://doi.org/10.1109/NFC.2013.6482441

[120] R. S. Sandhu and P. Samarati. 1994. Access control: principle and practice. *IEEE Communications Magazine* 32, 9 (Sept 1994), 40–48. https://doi.org/10.1109/35.312842

[121] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. 2016. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 303–312. https://doi.org/10.1109/ICDCS.2016.46

[122] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[123] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 335–350. https://doi.org/10.1145/1294261.1294294

[124] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[125] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. of NDSS 2013*. 18.

[126] Sampath Srinivas and Karthik Lakshminarayanan. 2019. Simplifying identity and access management of your employees, partners, and customers. https://cloud.google.com/blog/products/identity-security/simplifying-identity-and-access-management-of-your-employees-partners-and-customers

[127] Jeff Vander Stoep and Chong Zhang. 2019. Queue the Hardening Enhancements. https://android-developers.googleblog.com/2019/05/queue-hardening-enhancements.html

[128] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[129] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[130] Sai Deep Tetali. 2018. Keeping 2 Billion Android devices safe with machine learning. https://android-developers.googleblog.com/2018/05/keeping-2-billion-android-devices-safe.html

[131] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. 2015. Security Metrics for the Android Ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'15)*. Association for Computing Machinery, New York NY USA, 87–98. https://doi.org/10.1145/2808117.2808118

[132] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 941–955. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[133] Sami Tolvanen. 2017. Hardening the Kernel in Android Oreo. https://android-developers.googleblog.com/2017/08/hardening-kernel-in-android-oreo.html

[134] Sami Tolvanen. 2018. Control Flow Integrity in the Android kernel. https://security.googleblog.com/2018/10/posted-by-sami-tolvanen-staff-software.html

[135] Sami Tolvanen. 2019. Protecting against code reuse in the Linux kernel with Shadow Call Stack. https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html

[136] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. ACM Press, 1675–1689. https://doi.org/10.1145/2976749.2978406

[137] Jeff Vander Stoep. 2015. Ioctl Command Whitelisting in SELinux. http://kernsec.org/files/lss2015/vanderstoep.pdf Linux Security Summit.

[138] Jeff Vander Stoep. 2016. Android: Protecting the Kernel. https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf Linux Security Summit.

[139] Jeff Vander Stoep. 2017. Shut the HAL up. https://android-developers.googleblog.com/2017/07/shut-hal-up.html

[140] Jeff Vander Stoep and Sami Tolvanen. 2018. Year in Review: Android Kernel Security. https://events.linuxfoundation.org/wp-content/uploads/2017/11/LSS2018.pdf Linux Security Summit.

[141] W3C. [n. d.]. Web Authentication: An API for accessing Public Key Credentials. https://webauthn.io/

[142] R. Watson. 2012. *New approaches to operatng system security extensibility.* Technical Report UCAM-CL-TR-818. Cambridge University. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-818.pdf

[143] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android Permissions Remystified: A Field Study on Contextual Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 499–514. https://www.usenix.org/conference/usenixsecurity15/

technical-sessions/presentation/wijesekera

[144] Linux Kernel Security Subsystem Wiki. 2019. Exploit Methods/Userspace execution. https://kernsec.org/wiki/index.php/Exploit_Methods/Userspace_execution

[145] Shawn Willden. 2018. Insider Attack Resistance. https://android-developers.googleblog.com/2018/05/insider-attack-resistance.html

[146] Xiaowen Xin. 2018. Titan M makes Pixel 3 our most secure phone yet. https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/

[147] Keun Soo Yim, Iliyan Malchev, Andrew Hsieh, and Dave Burke. 2019. Treble: Fast Software Updates by Creating an Equilibrium in an Active Software Ecosystem of Globally Distributed Stakeholders. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 104 (Oct. 2019), 23 pages. https://doi.org/10.1145/3358237

[148] David Zeuthen, Shawn Willden, and René Mayrhofer. 2020. Privacy-preserving features in the Mobile Driving License. https://security.googleblog.com/2020/10/privacy-preserving-features-in-mobile.html

[149] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 611–622. https://doi.org/10.1145/2508859.2516689

A TOWARDS A FORMAL NOTATION OF ANDROID SECURITY MODEL RULES

Standard access control models are traditionally based on a matrix notation of (S, O, A) triples with subjects, objects, and a defined set of access permissions $A[s, o]$ (typically read, write, and execute) [65]. While the differences in specific implementations of this conceptual matrix (ACLs vs. capabilities) are superfluous for our discussion, the basic notation is becoming limited [52] and unfortunately not directly applicable to the Android model of multiple stakeholders and combining multiple different types of security controls.

Within the scope of this first draft of a notation of the Android platform security meta model, we define the involved stakeholders as parties $P \in \boldsymbol{P}$ for a set $\boldsymbol{P}$ with pre-defined classes[19]:

- P_U denotes a user of the system. They may or may not be equivalent to the owner of the hardware (client device such as a smart phone), where hardware ownership is defined as out of scope of the security model at this time. However, users are assumed to own their data.
- P_D denotes the developer of an app, which implicitly includes backend services used by that app. That is, P_D is considered owner of the code that is executed by the app as well as potential owner of data used as part of a service (such as video streaming).
- P_P denotes the Android platform or more specifically the set of system components that are neither third-party apps nor representing user data. Examples are cell communication, WiFi or Bluetooth services, standard UI elements, or hardware drivers.
- P_O denotes an optional organization that can place additional restrictions on the use of a device, e.g. because it is owned by the organization or internal services are accessed through the devices that require these security measures. Examples of relevant organizations are employers or schools.

For a specific interaction, e.g. one particular user using one particular app to take a picture and store it on the local filesystem of one particular Android system, the relevant stakeholders will be specific instances of these classes, e.g. $\boldsymbol{P_{photoaction}} = \{P_{U_1}, P_{D_1}, P_{P_1}\}$. This set will usually include 3 or 4 (if an organization policy is in place) specific stakeholders, one from each class. The next interaction may use a different set of specific stakeholders, such as another app, another user (using the same app on the same platform), or continuing the use of the same app by the same user but on a different device. To abstract from those specific use cases, we will use the short form $\forall P$ to refer to all stakeholders of a *current* interaction without loss of generality.

Each stakeholder P has some elements:

- $S(P)$ denotes the internal state of this stakeholder in the form of arbitrary data stored and controlled by this party. This can take different forms, e.g. files, key/value preferences, streams, etc. Note that the Android platform security model is primarily concerned with internal state stored directly within the respective Android system (temporarily in RAM or permanently on non-volatile storage), and data stored outside the physical system (e.g. on cloud services) is considered out of scope. However, internal state of one stakeholder (user account data, an app token, etc.) is often used to directly reference such data, and some rules of the Android platform security model *may* therefore transitively apply to such external data as well.
- $C(P, A) \in \{allow, deny\}$ denotes the run-time consent decision of this party concerning a specific action A, which is generally considered to be the context of a consent query. The specific form of a consent query varies significantly between stakeholder classes (e.g. users will often consent through a pop-up UI dialog while apps will typically consent through policies evaluated automatically in their application code) and between actions.

[19] These classes of stakeholders could also be seen as roles in an RBAC notation. However, there is no hierarchical relationship between these stakeholders – they are intentionally considered to be peers – and therefore the RBAC notation seems less useful in this case.

The enforcing agent (e.g. platform components acting as security monitors) may cache and re-apply previous consent decisions without asking again depending on configured policy. For example, user consent expressed through run-time permissions (which is only one way of users to express consent for a specific class of actions for which such permissions are defined by the platform) can currently result in $C(P_U, A) \in \{allow-always, allow-once, allow-in-foreground, deny-once, deny-always\}$ and stored by the permissions controller for future use. A current *allow* can therefore result from multiple different responses such as $\{allow-always, allow-once\}$.

With these preliminaries, we can more formally specify the access control aspects of Android platform security model rules:

Rule ① (multi-party consent). Consent for executing an action A depends on consent of all relevant parties.

$$C(A) = allow \iff \forall P : C(P, A) = allow \tag{1}$$

Consent typically grants (partial) access to the internal state of the stakeholder granting this access.

$$C(P, A) = true \implies S(\forall P) \ni f(S(P), C(A)) \tag{2}$$

where $f(S(P))$ denotes the access control function limiting access to the internal state of P scoped to the context of consent to the current action A. That is, the state accessible to all parties $\forall P$ within the current interaction A includes this additional state of the consenting party P. The type of (partial) access, e.g. read or write, depends on the context of an action A, but may be explicit in the consent query (e.g. read-only or read-write access permission to external storage).

Further, consent of one party may depend on run-time context of another party such as

$$C(P_U, A) = allow-in-foreground \wedge UI-foreground \in S(P_D) \implies C(P_U, A) = allow$$

where P_U consent depends on the UI state of P_D within the current interaction. There is currently no complete set of all sub-instances of consent decisions and their contextual dependencies[20], and the potential existence of a set sufficient for expressing all necessary conditions is doubtful.

Within the lattice notation of mandatory access control (MAC) policies, this multi-party consent rule implies trivial lower (no consent) and upper (all involved stakeholders consent) bounds. While the BLP model is still an underlying principle of SELinux policies and used for Android sandboxing, it is only a part of the higher-level multi-party consent: namely consent expressed by the platform components P_{P_x} is internally derived through through BLP flows. On this level of inter-component permission granting, potential future work could investigate the applicability of the Take-Grant model [50] for reasoning about collusion issues. Comparison of the expressive power under a meta model like [52] or [46] is another potential extension, although a cross-abstraction comparison is at least non-obvious at this point.

Rule ② (open ecosystem). All stakeholder classes P represent unbounded sets, and new specific instances can be created at any time by any party: new users can be created on the system itself, new apps can be published and installed without a central gatekeeping instance, and new platforms (devices) can be created freely as long as they follow these rules (cf. rule ③).

[20] This is especially true for context dependent consent by apps P_D, which can use arbitrarily complex code to implement their own decision process.

Rule ④ (safe reset).

$$\forall P : S(P) := \emptyset \tag{3}$$

For the developer P_D, resetting their state is interpreted as uninstalling the app from a specific platform and clearing all app data in the process. For user P_U and platform P_P resetting state implies removing a user account or invoking reset to factory default. The implication is that this also resets all consent from the resetting party, as consent is defined as (partial) access to internal state.

Rule ⑤ (applications as principals). A developer can have multiple apps, which have distinct internal state. That is, a developer P_D actually manages a set of parties P_A in the form of all apps they sign with their developer key.

$$P_D \supset \{P_{A_1}, ..., P_{A_n}\} \tag{4}$$

Apps within a single developer P_D can explicitly share each other's consent decisions by requesting to be installed with the same shared UID (which implies signature with the same developer key). One of the key elements of rule ⑤, namely that apps P_A do not implicitly represent users P_U is already enforced through keeping their internal state separate (as defined above in preliminaries).

Rule ③ (compatibility). Compatibility is the most difficult to express formally, and we only give an intuitive sketch here. As mentioned in section 3, the Android platform security model by practical necessity spans multiple layers of abstraction. Compatibility therefore requires a rule on a meta level: all potential instances of user P_U, developer P_D, app P_A, and organization P_P operate under the other rules and can continuously update their state and consent decisions, while the platform P_P – specifically as the set of mutually untrusted components enforcing the other rules on different layers – is pinned to a specific version of AOSP to implement the rules of this model. If P_P fails to fully implement this meta rule, all other rules automatically become invalid.

That is, *invalidation of any rule leads to invalidation of all others.* Other parties need to learn of an invalid D_P so that they can revoke their own consent (e.g. users re-installing the system image to revert to a known-good state). This directly complements (and effectively enables) Rule ② because it allows other parties to trust P_P (which often enforces consent decisions by these parties).

On a formal level, enforcement of this rule must necessarily be performed outside the platform security model (hence the elevation to a meta rule) and therefore assumes a trusted third party for platform verification. In the current Android ecosystem, this rule is implemented through a combination of automated test suites (including CTS and VTS, which are available as part of AOSP itself), platform attestation keys provisioned by OEMs and signed by Google for systems verifiably passing those test suites, and APIs to query these attestation results that can be used by the other parties at run-time.

Note that this first formalization only captures the access control implication of the model rules. It is subject to future work to evaluate if these rules could be formulated under a meta model like [46] and be expressed in tandem with access control models of underlying security controls such as MAC policies in the kernel. However, such an endeavour only seems useful if cross-abstraction validation can then be performed using automated tools.