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

ACM Reference Format:
René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2020. The Android Platform Security Model. *ACM Trans. Priv. Sec.* 1, 1 (May 2020), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Android is, at the time of this writing, the most widely deployed end-user operating system. With more than 2 billion monthly active devices [8] 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,

*Last updated in March 2020 based on Android 10 as released and some pre-release public sources of Android 11.

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.

2471-2566/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 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.

[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.
[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

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 [17, 43, 67, 115]. 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 [54].

Structure. In the following, we will first introduce Android security principles, and 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 10 at the time of its initial release unless mentioned otherwise (cf. [39] for changes in Android 10 and [101] 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 10 (inclusive), spanning about 9 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 platform security principles.

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 [16].

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 [1], 56% globally [2], 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 [102] (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 Android security principles

From the start, Android has assumed a few basic security and privacy principles that can be seen as an implicit contract between many parties in this open ecosystem:

Actors control access to the data they create. Any actor that creates a data item is implicitly granted control over this particular instance of data representation. Note that this refers to the technical act of protecting data, either on the filesystem or in memory — but does not automatically imply ownership over data in the legal sense. While this model has long been the default for filesystem access control (DAC, cf. section 4.3.1 below), we consider it a wider design principle. One intention of making this principle explicit is that exceptions such as device backup (cf. section 5) can be argued about within the scope of the security model.

Consent is informed and meaningful. 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.

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.

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 below 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*, *exploit mitigation*, *integrity*, and *patching/updates*. Their implementation will be discussed in more detail in section 4.

2.3 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 [95] previously introduced a layered threat model for mobile devices which we adopt for discussing the Android security model within the scope of this paper:

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 access:

T1 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.

T2 Screen locked devices under complete physical control of an adversary, e.g. thieves trying to exfiltrate data for additional identity theft.

T3 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.

T4 (Screen locked or unlocked) devices in physical proximity to 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 [5, 52]. 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.

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 [57] with additional relay threats for short-range radios (e.g. NFC or BLE wormhole attacks [105]). For practical purposes, we mainly consider two network-level threats:

T5 Passive eavesdropping and traffic analysis, including tracking devices within or across networks, e.g. based on MAC address or other device network identifiers.

T6 Active manipulation of network traffic, e.g. MITM on TLS connections or relaying.

These two threats are different from [T4] (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 code from arbitrary sources, and does not enforce vetting of apps by a central instance. This implies attack vectors on multiple levels (cf. [95]):

T7 Abusing APIs supported by the OS with malicious intent, e.g. spyware.

T8 Exploiting bugs in the OS, e.g. kernel, drivers, or system services [6, 9, 10, 12].

T9 Abusing APIs supported by other apps installed on the device [11].

T10 Untrusted code from the web (i.e. JavaScript) is executed without explicit consent.

T11 Mimicking system or other app user interfaces to confuse users (based on the knowledge that standard in-band security indicators are not effective [56, 103]), e.g. to input PIN/password into a malicious app [66].

T12 Reading content from system or other app user interfaces, e.g. to screen-scrape confidential data from another app [78, 84].

T13 Injecting input events into system or other app user interfaces [69].

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 and metadata:

T14 Exploiting code that processes untrusted content in the OS or apps, e.g. in media libraries [4]. This can be both a local as well as a remote attack surface, depending on where input data is taken from.

T15 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.

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 general Android principles 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. [113]). 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).

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).

There are 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). 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) [20] 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 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 [80, 107] (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 [92], 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.

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'.

4.1 Consent

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

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.

[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 will forcibly factory reset the device (wiping all data) to make sure that security guarantees are not retroactively violated for data on the 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 [21]. 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.

In order to ensure that it is the app developer and not another party that is consenting, applications are signed by the developer (or when using key rotation functionality, a key that was previously granted this ability by the app). 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 installation, so replacing or modifying apps in transit (e.g. when side-loading apps) is currently out of scope of the platform security model and may violate developer consent.

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 that entity – 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 considered 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 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). 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.

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. [18]). 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. [65]). 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 ([T1]-[T3]), which directly relies on the next component in the form of the Android lock screen. Implementing model rule ① is cross-cutting on all system layers.

We use two examples to better describe the consent parties:

- Sharing data from one app to another requires:

491 – user consent through the user selecting a target app in the share dialog;
492 – developer consent of the source app by initiating the share with the data (e.g. image) they
493 want to allow out of their app;
494 – developer consent of the target app by accepting the shared data; and
495 – platform consent by arbitrating the data access between different components and ensuring
496 that the target app cannot access any other data than the explicitly shared item through
497 the same link, which forms a temporary trust relationship between two apps.
498 • Changing mobile network operator (MNO) configuration option requires:
499 – user consent by selecting the options in a settings dialog;
500 – (MNO app) developer consent by implementing options to change these configuration
501 items, potentially querying policy on backend systems; and
502 – platform consent by verifying e.g. policies based on country regulations and ensuring that
503 settings do not impact platform or network stability.
504
505 ### 4.2 Authentication
506 Authentication is a gatekeeper function for ensuring that a system interacts with its owner or
507 legitimate user. On mobile devices the primary means of authentication is via the lockscreen. Note
508 that a lockscreen is an obvious trade-off between security and usability: On the one hand, users
509 unlock phones for short (10-250 seconds) interactions about 50 times per day on average and
510 even up to 200 times in exceptional cases [62, 75], and the lockscreen is obviously an immediate
511 hindrance to frictionless interaction with a device [73, 74]. On the other hand, devices without a
512 lockscreen are immediately open to being abused by unauthorized users ([T1]-[T3]), and the OS
513 cannot reliably enforce user consent without authentication.
514 In their current form, lockscreens on mobile devices largely enforce a binary model — either the
515 whole phone is accessible, or the majority of functions (especially all security or privacy sensitive
516 ones) are locked. Neither long, semi-random alphanumeric passwords (which would be highly
517 secure but not usable for mobile devices) nor swipe-only lockscreens (usable, but not offering any
518 security) are advisable. Therefore, it is critically important for the lockscreen to strike a reasonable
519 balance between security and usability.
520 Towards this end, recent Android releases use a tiered authentication model where a secure
521 knowledge-factor based authentication mechanism can be backed by convenience modalities that
522 are functionally constrained based on the level of security they provide. The added convenience
523 afforded by such a model helps drive lockscreen adoption and allows more users to benefit both
524 from the immediate security benefits of a lockscreen and from features such as file-based encryption
525 that rely on the presence of an underlying user-supplied credential. As an example of how this helps
526 drive lockscreen adoption, starting with Android 7.x we see that 77% of devices with fingerprint
527 sensors have a secure lockscreen enabled, while only 50% of devices without fingerprints have a
528 secure lockscreen[9].
529 As of Android 10, the tiered authentication model splits modalities into three tiers.
530
531 • *Primary Authentication* modalities are restricted to knowledge-factors and by default include
532 PIN, pattern, and password. Primary authentication provides access to all functions on the
533 phone.
534 • *Secondary Authentication* modalities are biometrics — which offer easier, but potentially less
535 secure (than Primary Authentication), access into a user's device. Secondary modalities are
536 themselves split into sub-tiers based on how secure they are, as measured along two axes:
537
538 [9]These numbers are from internal analysis that has not yet been formally published.
539

(1) *Spoofability* as measured by the Spoof Acceptance Rate (SAR) of the modality [100]. Accounting for an explicit attacker in the threat model on the level of [T1-T2] helps reduce the potential for insecure unlock methods [97].
(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.

- *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.

As of April 2019, lockscreen authentication on Android 7+ can now be used for FIDO2/WebAuthn [13, 126] authentication to web pages, additionally making Android phones second authentication factors for desktop browsers through implementing the Client to Authenticator Protocol (CTAP) [111]. While this support is currently implemented in Google Play Services [68], 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.

4.2.1 *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[10], apps can make use of a new "Identity Credential" subsystem to support privacy-first identification [77] (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 [14]. The first version of this subsystem targets in-person presentation of credentials, and identification to automated verification systems is subject to future work.

[10]Note to reviewers: The library is expected to be released in a production version before the final version of this paper is submitted. A preview draft is available as open source at https://android.googlesource.com/platform/hardware/interfaces/+/refs/heads/master/identity/aidl/android/hardware/identity/IIdentityCredentialStore.aidl.

Android 11 will include the Identity Credential subsystem in the form of a new HAL, a new system daemon, and API support in AOSP [22]. 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"[11] 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 ②: 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 [T7-T15].
- Attack surface reduction: reducing the number of entry points, particularly [T7-T9], i.e. the principle of least privilege.
- Containment: isolating and de-privileging components, particularly ones that handle untrusted content as in [T10] and [T14].
- Architectural decomposition: breaking privileged processes into less privileged components and applying attack surface reduction for [T8-T14].
- 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 Permissions

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 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 [110] 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`).

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

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. [64, 65, 104, 128]).

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 potentially 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 [50].

At a high level Android permissions fall into one of five classes in increasing order of severity:

(1) Audit-only Permissions: These are install time permissions with the 'normal' protection level.
(2) Runtime Permissions: These are permissions that the user must approve as part of a runtime prompt dialog. 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 [38].
(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. 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 privileged applications only and allow privileged actions such as carrier billing.
(5) Signature Permissions: These permissions are only available to components signed with the same key as the component which declares the permission e.g. the platform signing key. They are intended to guard internal or highly privileged actions, e.g. configuring the network interfaces.

Permission availability is defined by their `protectionLevel` attribute [24] with two parts (the level itself and a number of optional flags) which may broaden which applications may be granted a permission as well as how they may request it. The protection levels are:

- `normal`: Normal permissions are those that do not pose much privacy or security risk and are granted automatically at install time. These permissions are primarily used for auditability of app behavior.
- `dangerous`: Permissions with this `protectionLevel` are runtime permissions, and apps must both declare them in their manifest as well as request users grant them during use. These permissions, which are fairly fine-grained to support auditing and enforcement, are grouped into logical permissions using the `permissionGroup` attribute. When requesting runtime permissions, the group appears as a single permission to avoid over-prompting.
- `signature`: Applications can only be granted such permission if they are signed with the same key as the application that defines the permission, which is the platform signing key for platform permission. These permissions 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.

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 `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.

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, partially through the addition of MAC policies but also including many other mechanisms such as runtime permissions and attack surface reduction (cf. Table 1).

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 [76], 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.

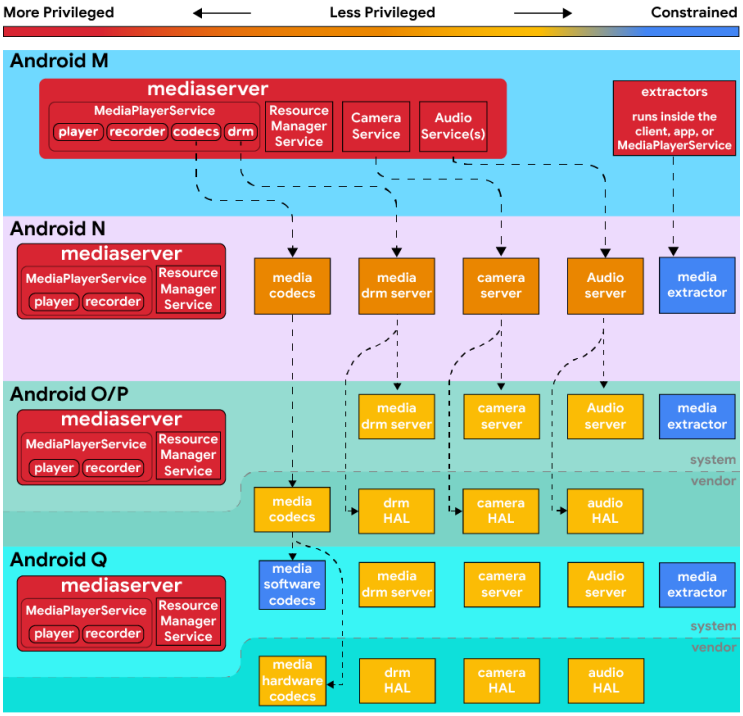

Fig. 1. Changes to mediaserver and codec sandboxing from Android 6 to Android 10

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 [112]. 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).

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 [123]. Hardware drivers provided by System on a Chip (SoC) vendors account for the vast majority of kernel vulnerabilities on Android [125]. Reducing app/system access to these drivers was described above, but sandboxing code inside the kernel itself also 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` [40]. 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 [T2]-[T4] or chaining together multiple bugs from user space code to reach kernel surfaces in [T8]), misconfiguration (e.g. with incorrect or overly permissive SELinux policies [51]), or bypass (e.g. by modifying the 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 [26]. That is, even with a fully compromised kernel, an attacker cannot read key material stored in Keymaster[12]. 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 [27], allowing verification of compatibility in terms of rule ③.
- **Strongbox**, specified starting with Android 9.0, implements the Android keystore in separate tamper resistant hardware (TRH) for even better isolation. This mitigates [T1] and [T2] against strong adversaries, e.g. against cold boot memory attacks [72] or hardware bugs such as Spectre/Meltdown [82, 91], Rowhammer [48, 121], or Clkscrew [114] 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 hardware. Adding a separate TRH affords another sandboxing layer of defense in depth.

  > 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 [28]. **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 Pixel 3 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 [96, 129]. 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 [29], partially mitigates [T11] and [T13]. 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

[12]Note: This assumes that hardware itself is still trustworthy. Side-channel attacks such as [86] 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.

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.

4.4 Encryption of data at rest

A second element of enforcing the security model, particularly rules ① and ③, 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 [T1], 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.[13] FBE allows individual files to be tied to the credentials of different users, cryptographically protecting per-user data on shared devices [T3]. 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 [53], 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, 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 [T5] and [T6].

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 MITM attacks [60, 61, 70]. 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

[13] 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.

memory access (cf. [83, slide 54]). While this section primarily describes mitigations against memory unsafety, 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.

Android mandates the use of a number of mitigations including ASLR [44, 109], RWX memory restrictions (e.g. $W \oplus X$, cf. [108]), and buffer overflow protections (such as stack-protector for the stack and allocator protections for the heap). Similar protections are mandated for Android kernels [118].

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 [41]) or have a history of high severity security vulnerabilities (e.g. the kernel). Android has pioneered the use of LLVM undefined behavior sanitizer (UBSAN) 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) [119] in the kernel and security sensitive userspace components including media, Bluetooth, WiFi, NFC, and parsers [93] in a fine-grained variant as implemented by current LLVM [117] that improves upon previous, coarse-grained approaches that have been shown to be ineffective [55]. 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 [112] as the best trade-off between performance overhead and effectiveness [47].

These mitigation methods work in tandem with isolation and containment mechanisms 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 and ③ without placing additional assumptions on which languages apps are written in.

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. Verified Boot [31] 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 [32]. 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 [19]) 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 [33].

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 [34]) version 2, the VMBeta 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 [34, 96]. 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 [34] 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[15].

(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 [35] 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.

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

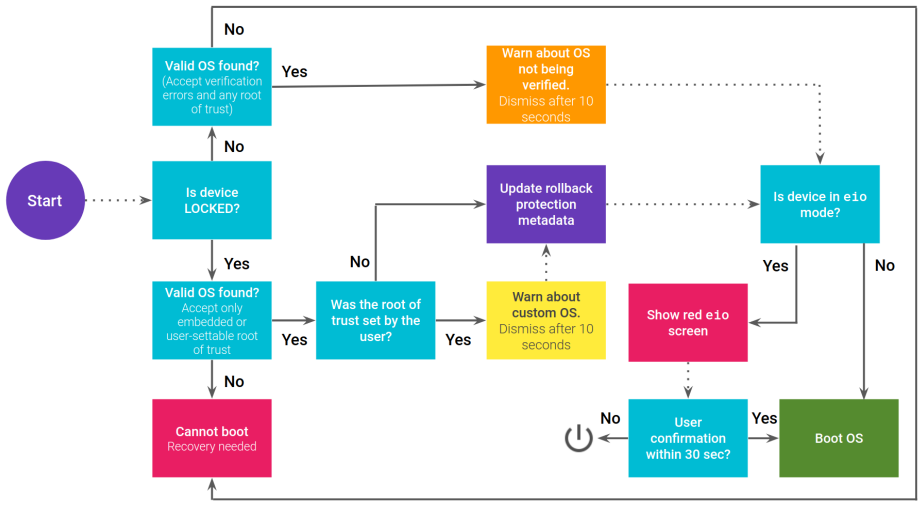

Fig. 2. Verified Boot flow and different states: (YELLOW): warning screen for LOCKED devices with custom root of trust set; (ORANGE): warning screen for UNLOCKED devices; (RED): warning screen for dm-verity corruption or no valid OS found [36].

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 [35].

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 [116] (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 launched with Android 8.0, with a goal of reducing the time/cost of updating Android devices [94, 98].

In 2018, the Android Enterprise Recommended program as well as general agreements with OEMs added the requirement of 90-day guaranteed security updates [37].

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` [71].

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[16]. 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) [79]. 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 [7].
  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 ([T1][T2]).

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 [113]). The most common data structures for efficiently implementing these relations (which, conceptually, are sparse matrices) are Access Control Lists (ACLs) [106] and capability lists (e.g. [127]). One of the first well-known and well-defined models was the Bell-LaPadula multi-level security model [42], 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.

[16]Preview release changes are described at https://developer.android.com/preview/privacy/package-visibility and are expected to become final (potentially in a slightly changed form) before the final version of this paper is published.

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 [58, 85, 99] as well as previous analysis of Android security mechanisms and how malware exploited weaknesses [15, 59, 63, 88–90, 130].

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 [129], and extending similar defenses to higher-level software is desirable [96]. Potential approaches could be reproducible firmware builds or logs of released firmware hashes comparable to e.g. Certificate Transparency [87].

- 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 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, 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] [n. d.]. https://www.stonetemple.com/mobile-vs-desktop-usage-study/
[2] [n. d.]. http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet
[3] [n. d.]. https://kernsec.org/wiki/index.php/Exploit_Methods/Userspace_execution
[4] 2015. Stagefright Vulnerability Report. https://www.kb.cert.org/vuls/id/924951
[5] 2017. BlueBorne. https://go.armis.com/hubfs/BlueBorne%20-%20Android%20Exploit%20(20171130).pdf?t=1529364695784
[6] 2017. CVE-2017-13177. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13177
[7] 2018. Android Enterprise Security White Paper. https://source.android.com/security/reports/Google_Android_Enterprise_Security_Whitepaper_2018.pdf
[8] 2018. Android Security 2017 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf
[9] 2018. CVE-2017-17558: Remote code execution in media frameworks. https://source.android.com/security/bulletin/2018-06-01#kernel-components
[10] 2018. CVE-2018-9341: Remote code execution in media frameworks. https://source.android.com/security/bulletin/2018-06-01#media-framework
[11] 2018. SVE-2018-11599: Theft of arbitrary files leading to emails and email accounts takeover. https://security.samsungmobile.com/securityUpdate.smsb
[12] 2018. SVE-2018-11633: Buffer Overflow in Trustlet. https://security.samsungmobile.com/securityUpdate.smsb
[13] 2019. Android Now FIDO2 Certified. https://fidoalliance.org/android-now-fido2-certified-accelerating-global-migration-beyond-pas
[14] 2020. Personal identification – ISO-compliant driving licence – Part 5: Mobile driving licence (mDL) application. Draft International Standard: ISO/IEC DIS 18013-5.
[15] 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
[16] 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
[17] 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
[18] 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
[19] 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
[20] AOSP. [n. d.]. https://source.android.com/compatibility/cdd
[21] AOSP. [n. d.]. https://developer.android.com/guide/components/intents-filters
[22] AOSP. [n. d.]. https://developer.android.com/reference/android/security/identity/package-summary

[23] AOSP. [n. d.]. https://developer.android.com/guide/topics/manifest/manifest-intro
[24] AOSP. [n. d.]. https://developer.android.com/guide/topics/manifest/permission-element
[25] AOSP. [n. d.]. https://developer.android.com/guide/components/activities/background-starts
[26] AOSP. [n. d.]. https://source.android.com/security/keystore/
[27] AOSP. [n. d.]. https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec
[28] AOSP. [n. d.]. https://source.android.com/security/authentication/gatekeeper
[29] AOSP. [n. d.]. https://developer.android.com/preview/features/security#android-protected-confirmation
[30] AOSP. [n. d.]. https://developer.android.com/training/articles/security-config
[31] AOSP. [n. d.]. https://source.android.com/security/verifiedboot/verified-boot
[32] AOSP. [n. d.]. https://android.googlesource.com/platform/external/avb/+/pie-release/README.md
[33] AOSP. [n. d.]. https://developer.android.com/training/articles/security-key-attestation
[34] AOSP. [n. d.]. https://android.googlesource.com/platform/external/avb/+/master/README.md
[35] AOSP. [n. d.]. https://source.android.com/security/apksigning/v3
[36] AOSP. [n. d.]. https://source.android.com/security/verifiedboot/boot-flow
[37] AOSP. [n. d.]. https://www.android.com/enterprise/recommended/requirements/
[38] AOSP. [n. d.]. Android platform permissions requesting guidance. https://material.io/design/platform-guidance/android-permissions.html#request-types
[39] AOSP. [n. d.]. https://source.android.com/security/enhancements/enhancements10. https://source.android.com/security/enhancements/enhancements10
[40] AOSP. [n. d.]. Security Updates and Resources - Process Types. https://source.android.com/security/overview/updates-resources#process_types
[41] Dan Austin and Jeff Vander Stoep. 2016. Hardening the media stack. https://android-developers.googleblog.com/2016/05/hardening-media-stack.html
[42] D. Bell and L. LaPadula. 1975. *Secure Computer System Unified Exposition and Multics Interpretation.* Technical Report MTR-2997. MITRE Corp., Bedford, MA.
[43] James Bender. 2018. Google Play security metadata and offline app distribution. https://android-developers.googleblog.com/2018/06/google-play-security-metadata-and.html
[44] 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
[45] Chad Brubaker. 2014. Introducing nogotofail — a network traffic security testing tool. https://security.googleblog.com/2014/11/introducing-nogotofaila-network-traffic.html
[46] 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
[47] 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
[48] Pierre Carru. 2017. Attack TrustZone with Rowhammer. http://www.eshard.com/wp-content/plugins/email-before-download/download.php?dl=9465aa084ff0f070a3acedb56bcb34f5
[49] 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.
[50] 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
[51] 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
[52] 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
[53] 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
[54] 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
[55] 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

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

[62] 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

[63] 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

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

[65] 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

[66] 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

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

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

[69] 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.

[70] 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.

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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

[76] 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.*

[77] 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

[78] 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

[79] 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

[80] 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.

[81] 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

[82] 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

[83] 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.

[84] 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.

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

[86] 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.

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

[88] 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

[89] 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

[90] 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

[91] 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

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

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

[94] 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

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

[96] René Mayrhofer. 2019. Insider Attack Resistance in the Android Ecosystem. In *Enigma 2019.* USENIX Association, Burlingame, CA.

[97] René Mayrhofer, Stephan Sigg, and Vishwath Mohan. [n. d.]. Adversary Models for Mobile Device Authentication. ([n. d.]). submitted for review.

[98] 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

[99] 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

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

[101] 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

[102] 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/

[103] 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

[104] 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

[105] 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

[106] 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

[107] 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

[108] 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

[109] 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

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

[111] 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

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

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

[114] 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

[115] 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

[116] 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

[117] 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

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

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

[120] 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

[121] 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

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

[123] 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.

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

[125] 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.

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

[127] 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

[128] 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

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

[130] 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

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.	[T10] access to [T5] [T8][T9] [T12][T13]
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.	[T8][T14]
5.x	Webview moved to an updatable APK, independent of a full system OTA.	[T10]
6.x	Run time permissions were introduced, which moved the request for dangerous permission from install to first use (cf. above description of permission classes).	[T7]
6.x	Multi-user support: SELinux categories were introduced for a per-physical-user app sandbox.	[T3]
6.x	Safer defaults on private app data: App home directory moved from `0751` UNIX permissions to `0700` (based on `targetSdkVersion`).	[T9]
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 [122, 123].	[T8][T14]
6.x	Removal of app access to `debugfs` (9% of all app-reachable kernel vulnerabilities).	[T8][T14]
7.x	`hidepid=2`: Remove `/proc/<pid>` side channel used to infer when apps were started.	[T11]
7.x	perf-event-hardening (11% of app reachable kernel vulnerabilities were reached via `perf_event_open()`).	[T8]
7.x	Safer defaults on `/proc` filesystem access.	[T7][T11]
7.x	MITM CA certificates are not trusted by default.	[T6]
8.x	Safer defaults on `/sys` filesystem access.	[T7][T11]
8.x	All apps run with a `seccomp` filter intended to reduce kernel attack surface.	[T8][T14]
8.x	Webviews for all apps move into the isolated process.	[T10]
8.x	Apps must opt-in to use cleartext network traffic.	[T5]
9.0	Per-app SELinux sandbox (for apps with `targetSdkVersion=P` or greater).	[T9][T11]
10	Apps can only start a new activity with a visible window, in the foreground activity 'back stack', or if more specific exceptions apply [25].	[T8][T9] [T10][T11]
10	File access on external storage is scoped to app-owned files.	[T7][T9]
10	Reading clipboard data is only possible for the app that currently has input focus or is the default IME app.	[T12]
10	`/proc/net` limitations and other side channel mitigations.	[T7]
11	Legacy access of non-scoped external storage is no longer available.	[T7][T9]

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`.	[T7][T8] [T14]
5.x	SELinux: MAC for all userspace processes.	[T7][T8]
6.x	SELinux: MAC for all processes.	
7.x	Architectural decomposition of mediaserver.	[T7][T8] [T14]
7.x	`ioctl` system call restrictions for system components [122].	[T7][T8] [T14]
8.x	*Treble* Architectural decomposition: Move HALs (Hardware Abstraction Layer components) into separate processes, reduce permissions, restrict access to hardware drivers [49, 124].	[T7][T8] [T14]
10	Software codecs, the source of approximately 80% of the critical/high severity vulnerabilities in media components were moved into a constrained sandbox	[T8][T14]
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.	[T8][T14]
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.	[T8][T14]
10	Scudo is a dynamic heap allocator designed to be resilient against heap related vulnerabilities.	[T8][T14]

Table 3. Kernel sandboxing improvements in Android releases

Release	Improvement	Threats mitigated
5.x	Privileged eXecute Never (PXN) [3]: Disallow the kernel from executing userspace. Prevents 'ret2user' style attacks.	[T8][T14]
6.x	Kernel threads moved into SELinux enforcing mode, limiting kernel access to userspace files.	[T8][T14]
8.x	Privileged Access Never (PAN) and PAN emulation: Prevent the kernel from accessing any userspace memory without going through hardened `copy-*-user()` functions [118].	[T8][T14]
9.0	Control Flow Integrity (CFI): Ensures that front-edge control flow stays within a precomputed graph of allowed function calls [119].	[T8][T14]
10	Shadow Call Stack (SCS): Protects the backwards edge of the call graph by protecting return addresses [120].	[T8][T14]

Table 4. Network sandboxing improvements in Android releases

Release	Improvement	Threats mitigated
6.x	`usesCleartextTraffic` in manifest to prevent unintentional cleartext connections [45].	[T5][T6]
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.	[T5][T6]
9.0	DNS-over-TLS [81] to reduce sensitive data sent over cleartext and made apps opt-in to using cleartext traffic in their network security config.	[T5][T6]
9.0	TLS is the default for all connections [46]	[T5][T6]
10	MAC randomization is enabled by default for client mode, SoftAp, and Wi-Fi Direct[14]	[T5]
10	TLS 1.3 support	[T5][T6]