The Android Platform Security Model

Abstract

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 address these threats. There are some exceptions in applying the security model, and we discuss both deliberate deviations as well as gaps for future work.

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 [1] 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 (IoT), automobiles, and more special-use categories) there is a vast – and still growing – range of use cases from communication, media consumption, and entertainment to finance, health, and physical sensors/actuators. Many of these applications are increasingly security and privacy critical, and Android as an OS needs to provide sufficient and appropriate assurances to users as well as developers.

To balance the different (and sometimes conflicting) needs and wishes of users, application developers, content producers, service providers, and employers, Android is fundamentally based on a multi-party consent model: *an action should only happen if all involved parties consent to it.* If any party does not consent, the action is 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 model 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 systematically analyze 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 three-party consent model described and analyzed in this paper predates the authors' own direct involvement with Android security, 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 exceptions.

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.

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. There are complementary security services 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 [2–5]. 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 [6].

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 Pie (9.0) at the time of its initial release unless mentioned otherwise. 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). Finally, we discuss exceptions to the model (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. 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 difficult security or privacy decisions to non-expert users who are not sufficiently skilled or informed to make them [7].

The Android ecosystem is immense. Different statistics show that in the last few years, the majority of the global, intensely diverse user base already used mobile devices to access Internet resources (e.g. 63% in the US [8], 56% globally [9], 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 [10] (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 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 API 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 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 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.

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 guards not only 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. [11] 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.

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 [12]. 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.

Untrusted network communication: 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 [13] with additional relay threats for short-range radios (e.g. NFC or BLE wormhole attacks). 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.

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. [11]):

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 [14–17].

T9 Abusing APIs supported by other apps installed on the device [18].

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 [19,20]), e.g. to input PIN/password into a malicious app [21].

T12 Reading content from system or other app user interfaces, e.g. to screen-scrape confidential data from another app [22,23].

T13 Injecting input events into system or other app user interfaces [24].

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 [25]. 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 location data).

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 4 rules:

① Three party consent. No action should be executed unless all three main parties – i.e. *user*, *developer* (implicitly representing stake holders such as content producers and service providers), and *platform* – agree. Any one party can veto the action. This three-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. [26]). 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 three party consent, even if one party primarily controls a data item, it may only act on it if the other two parties consent. Control over data also does not imply ownership (which is a legal, not technical, concept, and therefore outside the scope of an OS security model). Part of the consent may be a legal agreement that one party must make the data available to another (e.g. for exporting data from an app or backend service to the user for data portability [27, Article 20]). Failure to fulfill this agreement may count as abuse on the part of the app. Note that there are corner cases in which only two parties may need to consent (for actions in which the user only uses platform/OS services without involvement of additional apps) or a fourth party may be introduced (e.g. on devices or profiles controlled by a mobile device management (MDM), this policy is also considered for consenting to an action).

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) [28] 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[1], 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.

[1]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.

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.

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 [29, 30] (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 [31], history or other tracking data, cryptocurrency wallet keys, etc.).

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 address, 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.

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 steal data from other applications on a user's device.

Consent for developers, unlike the user, is enshrined via the code they sign and the system executes. 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 [32]. 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).

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.

The Platform While the platform, like the developer, consents via code signing the goals are quite different, with the platform acting to ensure that the system functions as intended. This includes enforcing regulatory or contractual requirements as well as taking an opinionated stance on what kinds of behaviors are acceptable. Platform consent is enforced via Verified Boot (see below for details) protecting the system images from modification as well as platform applications using the platform signing key and associated permissions, much like applications.

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. [33]). A naive approach to ensuring user consent is to prompt the user with a yes/no prompt for every action. This however 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 technical or understand nuanced security questions. If the user cannot understand a question being asked then, even if the user says yes, it is impossible for the user to provide meaningful consent. 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 be opinionated and not just offload hard problem onto the user.** Android regularly takes an opinionated stance on what behaviors are too risky to be allowed. We may avoid adding functionality that may be useful to a power user but too risky 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 they are being tired or distracted. To mitigate against such mistakes or the user simply changing their mind, whenever possible it should be easy for the user to undo a previous decision. 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:
  - user consent through the user selecting a target app in the share dialog;
  - developer consent of the source app by initiating the share with the data (e.g. image) they want to allow out of their app;
  - developer consent of the target app by accepting the shared data; and

    - platform consent by arbitrating the data access and ensuring that the target app cannot access any other data than the explicitly shared item through the same link, which forms a temporary trust relationship between two apps.

- Changing mobile network operator (MNO) configuration option requires:
    - user consent by selecting the options in a settings dialog;
    - (MNO app) developer consent by implementing options to change these configuration items, potentially querying policy on backend systems; and
    - platform consent by verifying e.g. policies based on country regulations and ensuring that settings do not impact platform or network stability.

4.2 Authentication

Authentication is a gatekeeper function for ensuring that a system interacts with its owner or legitimate user. On mobile devices the primary means of authentication is via the lockscreen. Note that a lockscreen is an obvious trade-off between security and usability: On the one hand, users unlock phones for short (10-250 seconds) interactions about 50 times per day on average and even up to 200 times in exceptional cases [34, 35], and the lockscreen is obviously an immediate hindrance to frictionless interaction with a device. On the other hand, devices without a lockscreen are immediately open to being abused by unauthorized users ([T1]-[T3]), and the OS cannot reliably enforce user consent without authentication.

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

Towards this end, recent Android releases use a tiered authentication model where a secure knowledge-factor based authentication mechanism can be backed by convenience modalities that are functionally constrained based on the level of security they provide. The added convenience afforded by such a model helps drive lockscreen adoption and allows more users to benefit both from the immediate security benefits of a lockscreen, as well as from features such as file-based encryption that rely on the presence of an underlying user-supplied credential. As an example of how this helps drive lockscreen adoption, starting with Android 6.x we see that 74% of devices with fingerprint sensors have a secure lockscreen enabled, while only 50% of devices without fingerprints have a secure lockscreen [36].

As of Android 9.0, the tiered authentication model splits modalities into three tiers.

- *Primary Authentication* modalities are restricted to knowledge-factors, and by default include PIN, pattern, and password. Primary authentication provides access to all functions on the phone.

- *Secondary Authentication* modalities are required to be 'strong' biometrics as defined by their spoof and imposter acceptance rates [37]. Accounting for an explicit attacker in the threat model helps reduce the potential for insecure unlock methods [38]. Secondary modalities are also prevented from performing some actions - for example they do not decrypt file-based or full-disk encrypted userdata partitions (such as on first boot), and are required to fallback to primary authentication once every 72 hours.

- *Tertiary Authentication* modalities are those that are either weak biometrics that do not meet the spoofability bar or 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, but are additionally 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 will, in the current architecture, be able to bypass the lockscreen.

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).
- Attack surface reduction: reducing the number of entry points i.e. principle of least privilege.
- Containment: isolating and de-privileging components, particularly ones which handle untrusted content.
- Architectural decomposition: breaking privileged processes into less privileged components and applying attack surface reduction.
- 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.

Permissions Android uses three distinct permission mechanisms to perform access control:

- **Discretionary Access Control (DAC):** Processes may grant/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 which 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 owns.
- **Mandatory Access Control (MAC):** The system has a security policy which dictates what actions are allowed. Only actions explicitly granted by policy are allowed. On Android this is implemented using SELinux [39] and primarily enforced by the kernel. Android makes extensive use of SELinux to protect system components and assert security model requirements during compatibility testing.
- **Android permissions** gate access to sensitive data and services. Enforcement is primarily done in userspace by the data/service provider (with notable exceptions such as `INTERNET`). Permissions are defined statically in an app's `AndroidManifest.xml` [40], though not all permissions requested may be granted. 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 dangerous permissions that the user must approve as part of a runtime prompt dialog. These permissions are guarding commonly used sensitive user data.
  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 preinstalled 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 [41] 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 the dangerous `protectionLevel` are runtime permissions, and apps must both declare them in their manifest as well as request users grant them during use. Dangerous 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 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.

The 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 racy per-process ID (PID) checks. 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`. 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 addressed 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 in Appendix A).

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`. Other processes that warranted UID isolation include the telephony stack, WiFi, and Bluetooth (cf. Table 2 in Appendix A).

Sandboxing the kernel Security hardening efforts in Android's userspace have increasingly made the kernel a more attractive target for privilege escalation attacks [42]. Hardware drivers provided by System on a Chip (SoC) vendors account for the vast majority of kernel vulnerabilities on Android [43]. 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 in Appendix A).

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` [44]. 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 [45]), 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 [46]. That is, even with a fully compromised kernel, an attacker cannot read key material stored in Keymaster. 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 [47], 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 addresses [T1] and [T2] against strong adversaries, e.g. against cold boot memory attacks [48] or hardware bugs such as Spectre/Meltdown [49, 50], Rowhammer [51,52], or Clkscrew [53] 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 [54]. **Weaver** implements the same functionality in TRH and communicates with Strongbox. Specified for Android 9.0 and initially implemented on the Google Pixel 2 phone, 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 [55]. 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 [56], partially addresses [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 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 userdata 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 and multi-user support introduced in Android 6.0 still required the password of the primary user before disk access.

These shortcomings were addressed 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.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.

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 encrypted. 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 [57]. Table 4 lists recent improvements in terms of making network connections safe by default.

4.6 Exploit mitigation

A robust security system should assume that software vulnerabilities exist and actively defend against them. Historically, about 85% of security vulnerabilities on Android result from unsafe memory access (cf. [58, 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. Much of the Android framework is written in Java, effectively defending large swathes of the OS from entire categories of security bugs.

Android mandates use of a number of mitigations including ASLR [59, 60], RWX memory restrictions (e.g. $W \oplus X$, cf. [61]), and buffer overflow protections such as stack-protector for the stack and allocator protections for the heap. Similar protections are mandated for Android kernels [62].

In addition to the mitigations listed above, Android is actively rolling out additional mitigations, focusing first on code areas which are remotely reachable (e.g. the media frameworks [63]) 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 LLVM Control Flow Integrity (CFI) in the kernel and security sensitive userspace components including media, Bluetooth, WiFi, NFC, parsers, etc [64].

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 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 Android's Trusted Computing Base (TCB) and system components to implement rule ④. Verified Boot [65] has been mandated since Android Nougat (with an exemption granted to devices which cannot perform AES crypto above 50MiB/sec.) and makes modifications to the boot chain detectable by verifying the boot, TEE, and additional vendor/OEM partitions and performing on-access verification of blocks on the system partition [66]. 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 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 [67].

Application integrity is enforced via APK signing [68]. 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.

With Android 9.0, only updateable apps are not covered by Verified Boot. Integrity of updateable apps is checked by Android's `PackageManager` during installation/update. 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 Android Verified Boot at the time of this writing, and is typically handled by OEM-specific boot loaders.

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 (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 [69, 70].

In 2018, the Android Enterprise Recommended program as well as general agreements with OEMs added the requirement of 90-day guaranteed security updates [71].

5 Exceptions and special cases

Android's implementation of the security model has improved over time, providing better enforcement of security boundaries, safer defaults, and clearer mechanisms for consent. However, there are potential weaknesses in the implementation – in the sense that mitigating vulnerabilities in these components would benefit from additional layers of defense – as well as some intentional deviations from the security model. 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 additional gaps by comparing the implementation in AOSP with the model we describe.

5.1 Weaknesses

- **Third party app access to the GPU driver:** An important part of sandboxing apps includes disallowing direct access to device drivers. Access control to device drivers has become progressively more restrictive due to Project Treble and security hardening in the media frameworks. However, GPU access has retained its exemption and is still directly accessible to untrusted apps. This weakens the security of Android by granting untrusted code access to a large and complex attack surface in the kernel, part of Android's TCB. Moving GPU access out of the app process and into an isolated HAL would significantly improve Android's resilience to vulnerabilities in the GPU driver.

- **Lockscreen outside TEE:** In the current architecture, the lockscreen is implemented within Android system services and therefore subject to being dismissed by attackers who have already reached kernel level privilege. While this is arguably sufficient for most use cases, advanced attackers may use such an attack chain to e.g. access authentication-bound keys that are stored in TEE or TRH but gated by lockscreen, or may eavesdrop on the entry of user knowledge factors (PIN/password/pattern). An obvious improvement would be to handle lockscreen in TEE (implementing secure input as an extension to the secure output currently available in the form of Protected Confirmation), but detailed threat models and software architectures require more work before they are suitable for the diverse Android ecosystem.

- **App code integrity:** Standard (user-installed) APK files are cryptographically verified upon installation, but by default not during run-time. This opens a potential threat when attackers already have a temporary privilege escalation to the point where they can modify code contained in (other) installed apps to persist their access.

5.2 Special Cases

- **Listing packages:** The ability for one app to discover what other apps are installed on the device can be considered an potential information leak and violation of user consent (rule ①). However, app discovery is currently necessary for direct app-to-app interaction which is derived from the open ecosystem principle (rule ②).

- **VPN apps may monitor/block network traffic for other apps:** This is generally a violation of the application sandbox 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.

- **Backup:** Data from the private app directory is backed up by default. Apps which do not consent must opt-out by setting fields in their manifest.

- **Enterprise:** Android Enterprise 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 [72].

  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 exception of this part of the model to address 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 [26]). The most common data structures for efficiently implementing these relations (which, conceptually, are sparse matrices) are Access Control Lists (ACLs) [73] and capability lists (e.g. [74]). One of the first well-known and well-defined models was the Bell-LaPadula multi-level security model [75], which defined properties for assigning permissions and can be considered the abstract basis for Mandatory Access Control and Type Enforcement schemes like SELinux. Consequently, the Android platform security model implicitly builds upon these general models and their principle of least privilege.

One fundamental difference is that, while classical models assume processes started by a user to be a proxy for their actions and therefore execute directly with user privileges, more contemporary models explicitly acknowledge the threat of malware started by a user and therefore aim to compartmentalize their actions. Many mobile OS (including Symbian as an earlier example) assign permissions to processes (i.e. applications) instead of users, and Android uses a comparable approach. A more detailed comparison to other mobile OS is out of scope in this paper, and we refer to other surveys [76–78] as well as previous analysis of

Android security mechanisms and how malware exploited weaknesses [79–85].

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 more limiting from a user point of view, it effectively limits the potential abuse a malicious app can do on data controlled by other apps; by removing an all-powerful user account that has unfiltered access to all data (as is the default with most current desktop and server OS), whole classes of threats such as file encrypting ransomware or direct data exfiltration are no longer practical.

AOSP implements the Android platform security model as well as the general security principles of 'defense in depth' and 'safe by default'. The 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, some gaps remain. Some of these are deliberate special cases to the conceptually simple security model, but there is also 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.

- 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, and extending similar defenses to higher-level software are desirable. Potential approaches could be reproducible firmware builds or logs of released firmware hashes comparable to e.g. Certificate Transparency [86].

- $W \oplus X$ memory is already a standard protection mechanism in most current OS, including Android. However, pages that are executable but not writable are typically still readable, and such read access can leak e.g. the location of ROP gadgets. A potential improvement would be to restrict code page access to only execute but not even read (execute-only pages).

- 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

Acknowledgments removed for anonymous review.

References

[1] "Android security 2017 year in review," Mar. 2018. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf

[2] N. Fischer, "Protecting WebView with Safe Browsing," April 2018. [Online]. Available: https://android-developers.googleblog.com/2018/04/protecting-webview-with-safe-browsing.html

[3] A. Ahn, "How we fought bad apps and malicious developers in 2017," January 2018. [Online]. Available: https://android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-and-malicious.html

[4] J. Bender, "Google Play security metadata and offline app distribution," June 2018. [Online]. Available: https://android-developers.googleblog.com/2018/06/google-play-security-metadata-and.html

[5] S. D. Tetali, “Keeping 2 billion android devices safe with machine learning,” May 2018. [Online]. Available: https://android-developers.googleblog.com/2018/05/keeping-2-billion-android-devices-safe.html

[6] E. Cunningham, “Improving app security and performance on google play for years to come,” December 2017. [Online]. Available: https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html

[7] A. Adams and M. A. Sasse, “Users are not the enemy,” *Commun. ACM*, vol. 42, no. 12, pp. 40–46, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/322796.322806

[8] [Online]. Available: https://www.stonetemple.com/mobile-vs-desktop-usage-study/

[9] [Online]. Available: http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[10] S. Pichai, “Android has created more choice, not less,” July 2018. [Online]. Available: https://blog.google/around-the-globe/google-europe/android-has-created-more-choice-not-less/

[11] R. Mayrhofer, “An architecture for secure mobile devices,” *Security and Communication Networks*, 2014. [Online]. Available: http://onlinelibrary.wiley.com/journal/10.1002/%28ISSN%291939-0122

[12] “Blueborne,” 2017. [Online]. Available: https://go.armis.com/hubfs/BlueBorne%20-%20Android%20Exploit%20(20171130).pdf?t=1529364695784

[13] D. Dolev and A. C. chih Yao, “On the security of public key protocols,” *IEEE Transactions on Information Theory*, vol. 29, pp. 198–208, 1983.

[14] “CVE-2017-13177,” Aug. 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13177

[15] “SVE-2018-11633: Buffer overflow in trustlet,” May. 2018. [Online]. Available: https://security.samsungmobile.com/securityUpdate.smsb

[16] “CVE-2018-9341: Remote code execution in media frameworks,” June. 2018. [Online]. Available: https://source.android.com/security/bulletin/2018-06-01#media-framework

[17] “CVE-2017-17558: Remote code execution in media frameworks,” June. 2018. [Online]. Available: https://source.android.com/security/bulletin/2018-06-01#kernel-components

[18] “SVE-2018-11599: Theft of arbitrary files leading to emails and email accounts takeover,” May. 2018. [Online]. Available: https://security.samsungmobile.com/securityUpdate.smsb

[19] R. Dhamija, J. D. Tygar, and M. Hearst, “Why phishing works,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’06. New York, NY, USA: ACM, 2006, pp. 581–590. [Online]. Available: http://doi.acm.org/10.1145/1124772.1124861

[20] P. Riedl, R. Mayrhofer, A. Möller, M. Kranz, F. Lettner, C. Holzmann, and M. Koelle, “Only play in your comfort zone: interaction methods for improving security awareness on mobile devices,” *Personal and Ubiquitous Computing*, pp. 1–14, March 2015.

[21] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, “Android UI Deception Revisited: Attacks and Defenses,” in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Feb. 2016, pp. 41–59. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-662-54970-4_3

[22] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee, “A11y attacks: Exploiting accessibility in operating systems,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: ACM, 2014, pp. 103–115. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660295

[23] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao, “On malware leveraging the android accessibility framework,” in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, I. Stojmenovic, Z. Cheng, and S. Guo, Eds. Cham: Springer International Publishing, 2014, pp. 512–523.

[24] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, “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, May 2017.

[25] (2015) Stagefright vulnerability report. https://www.kb.cert.org/vuls/id/924951.

[26] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[27] "General data protection regulation," Regulation of the European Parliament and Council 2016/679, April 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN

[28] AOSP. [Online]. Available: https://source.android.com/compatibility/cdd

[29] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "Cryptolock (and drop it): Stopping ransomware attacks on user data," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, June 2016, pp. 303–312.

[30] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Almgren, V. Gulisano, and F. Maggi, Eds. Cham: Springer International Publishing, 2015, pp. 3–24.

[31] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 threat model and security considerations," January 2013. [Online]. Available: https://www.rfc-editor.org/info/rfc6819

[32] AOSP. [Online]. Available: https://developer.android.com/guide/components/intents-filters

[33] B. B. Anderson, A. Vance, C. B. Kirwan, J. L. Jenkins, and D. Eargle, "From warning to wallpaper: Why the brain habituates to security warnings and what can be done about it," *Journal of Management Information Systems*, vol. 33, no. 3, pp. 713–743, 2016.

[34] D. Hintze, P. Hintze, R. D. Findling, and R. Mayrhofer, "A large-scale, long-term analysis of mobile device usage characteristics," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 1, no. 2, pp. 13:1–13:21, Jun. 2017.

[35] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proc. 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 179–194.

[36] Statistical data received through private communication from original Google sources for the purpose of this analysis.

[37] V. Mohan, "Better biometrics in Android P," June 2018. [Online]. Available: https://android-developers.googleblog.com/2018/06/better-biometrics-in-android-p.html

[38] R. Mayrhofer, S. Sigg, and V. Mohan, "Adversary models for mobile device authentication," submitted for review.

[39] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Proc. of NDSS 2013*, April 2013, p. 18.

[40] AOSP. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro

[41] ——. [Online]. Available: https://developer.android.com/guide/topics/manifest/permission-element

[42] J. Vander Stoep, "Android: Protecting the kernel," Slides online at https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf, 2016, Linux Security Summit.

[43] J. Vander Stoep and S. Tolvanen, "Year in review: Android kernel security," Slides online at https://events.linuxfoundation.org/wp-content/uploads/2017/11/LSS2018.pdf, 2018, Linux Security Summit.

[44] AOSP. Security updates and resources - process types. [Online]. Available: https://source.android.com/security/overview/updates-resources#process_types

[45] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang, "Analysis of seandroid policies: Combining mac and dac in android," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: ACM, 2017, pp. 553–565. [Online]. Available: http://doi.acm.org/10.1145/3134600.3134638

[46] AOSP. [Online]. Available: https://source.android.com/security/keystore/

[47] ——. [Online]. Available: https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec

[48] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1506409.1506429

[49] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[50] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz,

and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” *ArXiv e-prints*, Jan. 2018.

[51] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms.” ACM Press, 2016, pp. 1675–1689. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976749.2978406

[52] P. Carru, “Attack trustzone with rowhammer,” Slides online at http://www.eshard.com/wp-content/plugins/email-before-download/download.php?dl=9465aa084ff0f070a3acedb56bcb34f5, April 2017.

[53] A. Tang, S. Sethumadhavan, and S. Stolfo, “CLKSCREW: Exposing the perils of security-oblivious energy management,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1057–1074. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[54] AOSP. [Online]. Available: https://source.android.com/security/authentication/gatekeeper

[55] S. Willden, “Insider attack resistance,” May 2018. [Online]. Available: https://android-developers.googleblog.com/2018/05/insider-attack-resistance.html

[56] AOSP. [Online]. Available: https://developer.android.com/preview/features/security#android-protected-confirmation

[57] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, “The most dangerous code in the world: validating ssl certificates in non-browser software,” in *ACM Conference on Computer and Communications Security*, 2012, pp. 38–49.

[58] N. Kralevich, “The art of defense: How vulnerabilities help shape security features and mitigations in Android,” 2016, BlackHat.

[59] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: An efficient approach to combat a board range of memory error exploits,” in *Proc. USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251361

[60] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS ’04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[61] A. Seshadri, M. Luk, N. Qu, and A. Perrig, “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*, ser. SOSP ’07. New York, NY, USA: ACM, 2007, pp. 335–350. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294294

[62] S. Tolvanen, “Hardening the kernel in Android Oreo,” August 2017. [Online]. Available: https://android-developers.googleblog.com/2017/08/hardening-kernel-in-android-oreo.html

[63] D. Austin and J. Vander Stoep, “Hardening the media stack,” May 2016. [Online]. Available: https://android-developers.googleblog.com/2016/05/hardening-media-stack.html

[64] I. Lozano, “Compiler-based security mitigations in Android P,” June 2018. [Online]. Available: https://android-developers.googleblog.com/2018/06/compiler-based-security-mitigations-in.html

[65] AOSP. [Online]. Available: https://source.android.com/security/verifiedboot/verified-boot

[66] ——. [Online]. Available: https://android.googlesource.com/platform/external/avb/+/pie-release/README.md

[67] ——. [Online]. Available: https://developer.android.com/training/articles/security-key-attestation

[68] ——. [Online]. Available: https://source.android.com/security/apksigning/

[69] T. McDonnell, B. Ray, and M. Kim, “An empirical study of API stability and adoption in the Android ecosystem,” in *2013 IEEE International Conference on Software Maintenance*, Sept 2013, pp. 70–79.

[70] I. Malchev, “Here comes Treble: A modular base for android,” May 2017. [Online]. Available: https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html

[71] AOSP. [Online]. Available: https://www.android.com/enterprise/recommended/requirements/

[72] “Android enterprise security white paper,” Sept 2018. [Online]. Available: https://source.android.com/security/reports/Google_Android_Enterprise_Security_Whitepaper_2018.pdf

[73] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, Sept 1994.

[74] R. Watson, "New approaches to operatng system security extensibility," Cambridge University, Tech. Rep. UCAM-CL-TR-818, April 2012. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-818.pdf

[75] D. Bell and L. LaPadula, "Secure computer system unified exposition and multics interpretation," MITRE Corp., Bedford, MA, Tech. Rep. MTR-2997, July 1975.

[76] A. Egners, B. Marschollek, and U. Meyer, "Hackers in your pocket: A survey of smartphone security across platforms," RWTH Aachen University, Tech. Rep. 2012,7, Jan. 2012. [Online]. Available: https://itsec.rwth-aachen.de/publications/ae_hacker_in_your_pocket.pdf

[77] M. La Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," vol. 15, pp. 446–471, 01 2013.

[78] I. Mohamed and D. Patel, "Android vs iOS security: A comparative study," in *2015 12th International Conference on Information Technology - New Generations*, April 2015, pp. 725–730.

[79] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security Privacy*, vol. 7, no. 1, pp. 50–57, Jan 2009.

[80] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 611–622. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516689

[81] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer, "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)*, Sept 2014, pp. 3–17.

[82] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," *arXiv:1404.7431 [cs]*, Apr. 2014, arXiv: 1404.7431. [Online]. Available: http://arxiv.org/abs/1404.7431

[83] L. Li, T. F. Bissyand, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67 – 95, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917302987

[84] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 433–451.

[85] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Communications Surveys Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

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

[87] J. Vander Stoep, "Ioctl command whitelisting in selinux," Slides online at http://kernsec.org/files/lss2015/vanderstoep.pdf, 2015, Linux Security Summit.

[88] D. Cashman, "Selinux in android o: Separating policy to allow for independent updates," Slides online at https://events.static.linuxfound.org/sites/events/files/slides/LSS%20-%20Treble%20%27n%27%20SELinux.pdf, 2017, Linux Security Summit.

[89] J. Vander Stoep, "Shut the hal up," July 2017. [Online]. Available: https://android-developers.googleblog.com/2017/07/shut-hal-up.html

[90] [Online]. Available: https://kernsec.org/wiki/index.php/Exploit_Methods/Userspace_execution

[91] C. Brubaker, "Introducing nogotofail — a network traffic security testing tool," November 2014. [Online]. Available: https://security.googleblog.com/2014/11/introducing-nogotofaila-network-traffic.html

[92] AOSP. [Online]. Available: https://developer.android.com/training/articles/security-config

[93] E. Kline and B. Schwartz, "DNS over TLS support in Android P developer preview," April 2018. [Online]. Available: https://android-developers.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html

A Appendix: Security improvements in Android releases

The following tables are based on an analysis of security relevant changes to the whole AOSP code base between Android releases 4.x and 9.0, spanning about 7 years of code evolution.

Table 1: Application sandboxing improvements in Android releases

Release	Improvement	Threats addressed
≤ 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]
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 [42, 87].	[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]

Table 2: System sandboxing improvements in Android releases

Release	Improvement	Threats addressed
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 [87].	[T7][T8][T14]
8.x	*Treble* Architectural decomposition: Move HALs into separate processes, reduce permissions, restrict access to hardware drivers [88, 89].	[T7][T8][T14]

Table 3: Kernel sandboxing improvements in Android releases

Release	Improvement	Threats addressed
5.x	Privileged eXecute Never (PXN) [90]: 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 [62].	[T8][T14]

Table 4: Kernel sandboxing improvements in Android releases

Release	Improvement	Threats addressed
6.x	`usesCleartextTraffic` in manifest to prevent unintentional cleartext connections [91].	[T5][T6]
7.x	Network security config [92] 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 [93] to reduce sensitive data sent over cleartext and made apps opt-in to using cleartext traffic in their network security config.	[T5][T6]