Uraniborg's Device Preloaded App Risks Scoring Metrics

Billy Lau[1], Jiexin Zhang[2], Alastair R. Bereford[2], Daniel Thomas[3], and René Mayrhofer[4,1]

[1]Android Security and Privacy, Google Inc.
[2]University of Cambridge
[3]University of Strathclyde
[4]Johannes Kepler University Linz
billylau@google.com, {jz448,arb33}@cl.cam.ac.uk, d.thomas@strath.ac.uk, rm@ins.jku.at

August 2020

1 Introduction

The security of Android devices depends on a wide range of factors. In this paper we focus on quantifying the risks associated with one important factor: the security and privacy posture of *preloaded apps*. Such applications deserve particular attention since they are installed by the manufacturer on all devices of a particular make and model, individual apps may have elevated privileges beyond those available to apps installed via the Google Play Store, and typically cannot be removed by the user. In order to measure the risk presented by preloaded apps in a quantifiable way, we adopt a numerical approach and derive a single overall score for a given handset and therefore support the relative comparison of risks posed by different handsets.

Due to the difficulty in computing the security and privacy risk, we *approximate* the actual risk by estimating the attack surface[1] presented by this layer of software. We therefore present an extensible mathematical software framework that allows us to define, compute, and analyze various aspects of security and privacy risks of preloaded Android apps in a systematic manner.

This work fits into a larger effort called Uraniborg [12]. Uraniborg is designed to be an open-access observatory or database of preloaded app metadata that is envisioned to cover all Android devices in the ecosystem, for the benefit of end users. An instance of Uraniborg is hosted on the Android Device Security Database website [8].

Specifically, this work attempts to quantitatively measure the *change* in the risk introduced by preloaded apps on a given Android device by comparing it against a baseline risk measurement: Uraniborg's Device Preloaded App Risk (DPAR). It specifically does *not* consider the exact utility or features provided by the app. We assume that a perfect baseline implementation exists, where security risks have been considered and tailored to produce a *minimally functional*[2] Android device.

While we acknowledge that a number of apps are preloaded in order to provide additional value and features, every line of additional code comes with an inherent security risk due to the potential bugs that can turn into security vulnerabilities or be abused by developers themselves. In other words, configuration and installation of every app on a device contributes to overall device security. Thus, it is important to note that the scores produced do not provide an absolute mandate of what is good or bad, but instead provide a quantitative comparison relative to a minimalistic build and measure the additional risk introduced by the manufacturer through the introduction of preloaded apps on a specific device.

While we consider the likelihood and damage potential of security risk, we currently do not consider the cost of such risk. This is mainly because the cost is usually associated with the attempt to measure the difficulty of launching a certain attack. Based on the information that we are working on (metadata of packages and other device configurations but explicitly without a model of potential adversaries), we are not able to factor the cost into consideration for this framework. Instead, we assume equal cost for all preloaded apps.

[1]Note that the attack surface of any app – preloaded or user-installed, system-privileged or not – can only be measured precisely when related to specific attack vectors. In the general case, we are unaware of frameworks for accurately quantifying the attack surface w.r.t. potential future threats. Therefore, attack surface is loosely estimated by the set of (privileged) permissions and entrance points an app has.

[2]While this may be subjective, roughly, the device should be able to boot, make and receive calls and SMSes, and be able to connect to the Internet.

2 Risk Computation Framework

After considering whether to adopt a risk score represented by a bounded range (e.g. a score in the range from 1 to 10) or an unbound range (e.g. the natural numbers), we have settled on the bounded scoring system. This is largely driven by the motivation to have an output in a format that is more easily understood by the general public. Empirically and anecdotally, it is much simpler for consumers to decide on a rating of a product on e-commerce websites if the rating system is in the form of 5 stars. We envisage regular reviews of the scoring method in order to ensure we continue to evaluate risk objectively; this will require us to annotate scores with a version number or date (e.g. current year).

In the first version of our risk score, we consider the following three categories of security and privacy risks to calculate the bounded risk score:

- Platform Signature Risk
- Pregranted Permissions Risk
- Cleartext Traffic Risk

We acknowledge that the three categories of risks are not the *only* types of risks that are presented by preloaded apps today. However, they represent the major directly measurable risk types — in contrast to other important but difficult-to-measure categories such as code security (susceptibility to direct vulnerabilities in the app code) or entrance points (for other code to call into such apps). As we find more signals, we will update the model.

2.1 Baseline

A measurement is fundamentally a comparison of a subject to a reference point. Therefore, in our work, we calculate the increase in attack surface and infer risk from it. For this purpose, we compare every Android device against a baseline implementation. This baseline implementation must have the following properties:

- It must satisfy basic criteria that allows it to be called an Android device [2], including a complete implementation of the Android platform security model [13].
- It must be loadable **and** bootable on a *physical* device (not just in emulator or virtual machine).
- It must be able to perform basic operations such as:
  - make and receive phone calls,
  - send and receive SMSes,
  - connect to or browse the Internet, and
  - install other Android compatible applications.
- Its source code is publicly accessible.

The Android Open Source Project (AOSP) [1] builds fulfill all the criteria listed above. And for builds running Android 8.1.0 and above, Android defines a generic system image (GSI) [6] that fulfills all the criteria above as well. These builds have the following properties:

- They fulfill requirements in Android Compatibility Definition Document (CDD) [2], passing Compatibility Test Suite (CTS) [4], allowing it to be called Android.
- They are loadable and bootable on physical devices.
- When booted, they are usable.[3]
- They are built from source code that is open source [5, 3].

Thus, we have identified and set GSI and AOSP builds as the baseline for all derivative measurements in our framework. In our implementation, we have made use of GSI builds whenever available as the baseline for comparison, and AOSP builds when GSI builds are not available.

For a fairer comparison, we compare each target build/device with a baseline of the *same* API level. This is because we know that new changes are introduced with each release. It would be unfair to compare a device with the latest OS version with a baseline that was released three years ago.

Therefore, any scores that are computed in any risk category should be interpreted as a measurement of the *additional* risk introduced due to either the addition of APKs or modification/customization of similar APKs existing in the baseline. For this purpose, a score in each category is computed for a GSI build. And there are different baselines according to which API level or OS release we are comparing.

[3]We are able to make phone calls and send/receive SMSes using these builds.

3 General Formula

In order to quantify risks for the purposes of analysis and comparison, we need a numerical approach of representing such risks. To do this, we utilize the following formula [14] to standardize the computation of risks across all devices and all categories of risks:

$$\beta = \lambda \times \phi \tag{1}$$

where λ is the likelihood of the risk being measured, and ϕ is the damage potential of the risk category being measured. We explain each term in more detail in the following sections.

3.1 Likelihood

For each category of risk that we consider, we assume security risk increases as the volume of code increases, and therefore the likelihood of a security failure is estimated as the proportion of additional attack surface introduced. Due to the way that we mathematically compute likelihood (Equation 2), its value is bounded between 0 and 1, much like probability. However, we intentionally do not use the term "probability" as the construct of the likelihood value differs from the classical definitions of probability.

Across all categories, we use the following formula to define likelihood:

$$\lambda_m = \frac{|A_{t,m} \setminus A_{0,m}|}{|A_{t,m}|} \tag{2}$$

where:

m is the risk category that we are measuring, as defined in Section 4;
$A_{t,m}$ is the set of apps on a target device that fulfills m's risk definition;
$A_{0,m}$ is the set of apps on a baseline build that meets the same criteria; and
$\setminus$ is the set complement operator.

The likelihood is the number of apps with a particular risk on the target that are not in the baseline image divided by the number of apps with that risk on the target. Using this formula, if a device does not preinstall any additional app other than those found on a GSI (or AOSP, respectively) build, the target device would score a 0, as the numerator would evaluate to 0.

Note that when the manufacturer does not introduce any new packages, the denominator value would never be zero. The only scenario when this would occur is when manufacturer does not include any of the apps that are bundled in a corresponding baseline build. If this happens, the build, by definition, is no longer a valid Android build, as it would no longer fulfill the CDD [2].

3.2 Gaming the score

We do not consider OEMs as adversaries in the traditional sense, however OEMs wish to obtain a good score, so it is important that the scoring method encourages changes which result in positive security improvements, and discourages changes which reduce device security.

An advantage of our approach to scoring is that malicious players can try to improve their Uraniborg score by removing or merging GSI apps from their builds, but the risk score calculated via this formula will stay the same, due to the property of the set complement operator (as compared to utilizing $|A_t| - |A_0|$ as the numerator). That is, we intentionally do not measure the impact of apps removed from the baseline as this may allow the risk score to be gamed.

Shrewd OEMs may try to lower their build's risk score by merging additional apps that they preload into a single preloaded app. Unfortunately, such an approach to lowering of the risk score may actually come with an increase in real risk as merged apps will require the superset of all permissions of the previously separate apps and therefore do not benefit from Android app sandboxing to isolate privileged code from each other. In future iterations we may capture such behaviour but for now we note that merging multiple apps into a single app likely represents significant additional work (thereby acting as a disincentive). In any case, OEMs cannot easily merge third-party apps they bundle into their builds for various reasons including different signing keys or build systems, which is another argument against expecting significant gaming of this metric by merging previously separate apps.

As noted previously, our approach to calculating likelihood is subject to change in future iterations based on how OEMs in this space evolve. We therefore have the opportunity to take corrective action if our scoring method is gamed.

3.3 Damage Potential

Damage potential (denoted by ϕ in Formula 1) is essentially a statement (or again, approximation) of how severe a category of risk is. We currently assign a numerical value per category based on our domain knowledge and expertise in analyzing preloaded apps.

In this iteration, we assign scores for damage potential for each risk type introduced in Section 2 and further defined in Section 4 as shown in Table 1.

Category (Risk metric)	Damage Potential Value (ϕ)
Platform Signature Risk	6
Pregranted Permissions Risk	3
Cleartext Traffic Risk	1
Total	**10**

Table 1: Mapping of risk categories to damage potential values

We currently assign the heaviest weight to the platform signature category because platform signed apps have access to the most privileged resources on the system and represents over half of the overall risk.

When more risk categories are identified in the future, the distribution of damage potential may change to reflect the proportion of the individual risk category with respect to the others. However, the total score will continue to add up to 10.

3.4 Uraniborg's Device Preloaded App Risk (DPAR)

Uraniborg's Device Preloaded App Risk (DPAR) score is a composite score of the risk categories that we introduced in Section 2. In fact, it is the sum of individual scores from each metric.

We have picked a theoretical limit of 10 as the maximum risk score (B_{total}) any device can attain in this framework iteration. The DPAR score offers easy interpretation and comparison: *the higher the score, the more risk there is.*

The formula is as follows:

$$\begin{aligned} B_{total} &= \beta_{ps} + \beta_{pp} + \beta_{ct} && (3)\\ &= \lambda_{ps} \cdot \phi_{ps} + \lambda_{pp} \cdot \phi_{pp} + \lambda_{ct} \cdot \phi_{ct} && (4)\\ &= \lambda_{ps} \cdot 6 + \lambda_{pp} \cdot 3 + \lambda_{ct} && (5) \end{aligned}$$

where:

ps represents the platform signature category;
pp represents the pregranted permissions category;
ct represents the cleartext traffic category;
$\beta_{category}$ is the risk score for individual categories;
$\lambda_{category}$ is the likelihood value of the measured category;
$\phi_{category}$ is the damage potential value of the measured category.

4 Risk Categories

In this section, we discuss each risk type or category in more depth.

4.1 Platform Signature Risks

In the Android execution environment, not every preloaded app has equal power or capability. In particular, apps that are signed using the platform signature enjoy an array of additional capabilities; these apps are able to make use of certain signature level permissions and/or access certain data that other privileged apps do not have access to. Therefore, a preloaded app may have different capabilities based on whether it is platform signed. We highlight this difference and try to quantify the added risks introduced by platform-signed apps.

In particular, we try to account for privilege escalation risks that are embedded and carried by apps signed using platform keys, as they become the attack vector for other (unprivileged) apps. The score we compute also embodies the possibility of an insider attack where a rogue app that is already equipped with malicious code is (re-)signed using the platform key.

To determine if an app is signed using the platform key, we compare its signature with the package named `android` from the same build. If it matches, the app is platform signed.

4.1.1 Formula

Following the general formula (defined in Section 3), the formula to compute the platform signature risk for a given build is as follows:

$$\beta_{ps} = \lambda_{ps} \times \phi_{ps} \tag{6}$$

$$= \frac{|A_{t,ps} \setminus A_{0,ps}|}{|A_{t,ps}|} \times 6 \tag{7}$$

where:
$A_{t,ps}$ is the set of apps on a target device that are signed/verified using platform key/certificate,
$A_{0,ps}$ is the set of apps on a corresponding baseline build that are signed/verified using platform key/certificate.

4.2 Pregranted Permissions Risks

For preloaded apps, Android provides a mechanism for OEMs to grant certain privileges that are not available for ordinary apps that are installed during normal runtime, called privileged permissions whitelisting [7]. Permissions granted via the whitelisting process are called pregranted permissions. This unfortunately bypasses the normal user consent or runtime permission step, and therefore allows privileged permissions to be granted, potentially without user knowledge or consent.

Therefore, we attempt to quantify the risk posed by the pregranting of risky or privileged[4] permissions with the following method. We first classify Android permissions [11] into severity categories. Table 2 shows this classification. The classification is made using our expertise with Android's permission system and the risks of the capabilities that the individual permissions are guarding.

We then assign risk scores according to how severe each category of permission would be if they were abused, based on the type of data and/or capability that they guard against.

4.2.1 Formula

The formula for this category differs slightly from the general formula, although the larger part fits into the general formula.

We begin with some building blocks. Instead of counting the number of apps, we compute scores based on the number and type of pregranted permissions we found among non-platform-signed apps. For this, we introduce an intermediary term called the raw pregranted privileged permissions score (τ), defined as follows:

$$\tau_{cat} = \gamma_{cat} \sum_{i=app_0}^{app_k} \varepsilon_{cat,i} \tag{8}$$

where:
cat is the permission risk category, as defined in Section 4.2.2;
γ_{cat} is weight or score of the permission category, where $\gamma_{cat} \in \{100, 10, 7.5, 5, 2.5, 0\}$ (also refer to Table 2);
i iterates through every preloaded app on the target build that is *not* signed using platform signature;
$\varepsilon_{cat,i}$ is the count of pregranted permissions in app i that map to the current category being evaluated, *cat*.

With this, we are now able to define a total score (ξ) for pregranted permission risks, which is a composite score of all the raw scores (τ) for each permission category; formula as below:

$$\xi_{build} = \sum_{cat=Low}^{Astronomical} \tau_{cat} \tag{9}$$

where:
cat is the permission risk category (refer to Table 2);
τ_{cat} is the raw pregranted permissions score, as defined in Equation 8.

After obtaining the total score, we apply a variation of the general formula (Equation 1) by tweaking the definition of likelihood (λ) in Equation 2 to compute the score for pregranted permissions, as below:

$$\beta_{pp} = \lambda_{pp} \times \phi_{pp} \tag{10}$$

$$= \frac{\xi_t - \xi_0}{\xi_t} \times 3 \tag{11}$$

[4]This is to be understood semantically, and not directly mapped onto Android permission framework's `privileged` protection level.

where:
  ξ_t is the total score computed on a target device following Equation 9;
  ξ_0 is the total score computed on a baseline build following the same equation.

4.2.2 Permission Risk Categories

In Table 2, we enumerate and categorize those permissions we consider risky as pregranted permissions. It is important to highlight that the effort to complete this categorization for the entire Android permission space is currently a work-in-progress.

The permission category weight values are assigned based on their relative value to the other categories. For instance, while the number may look large, we assign 100 for the **ASTRONOMICAL** category to give it a sense of about 10x the magnitude of risk as compared to its next lower category (**CRITICAL**), which had an assignment of 10. We created the ASTRONOMICAL category to adjust for the abuse phenomena that we observe in the wild. In this case, we categorize the `INSTALL_PACKAGES` permission in this category because it allows for silent installation of arbitrary APKs, has had high visibility impact in terms of abuse and exploitation in the past, and provides the most direct method of compromising the security of users and their data. We may "upgrade" other permissions to this category in future revisions should we find more evidence of other permissions that display such a profile.

Note that these permissions are classified based on their guarded capabilities in Android 10 onward. With each OS release, potential changes or refactoring may happen that may decrease or increase a permission's current category. A good example of this is that prior to Android 10, the `READ_PHONE_STATE` permission would be classified as **HIGH**, due to the permanent device identifiers (e.g. (IMEI/MEID, IMSI, SIM, and build serial) that it guards. However, starting from Android 10, a bulk of the sensitive information that can be used for tracking has been moved, refactored or rescoped into a new permission called `READ_PRIVILEGED_PHONE_STATE`, putting the new permission in the HIGH category, but resulting in the `READ_PHONE_STATE` permission moving to LOW.

Permission Category	Weight	Permission Name
ASTRONOMICAL	100	`android.permission.INSTALL_PACKAGES`
CRITICAL	10	`android.permission.COPY_PROTECTED_DATA`
		`android.permission.WRITE_SECURE_SETTINGS`
		`android.permission.READ_FRAME_BUFFER`
		`android.permission.MANAGE_CA_CERTIFICATES`
		`android.permission.MANAGE_APP_OPS_MODES`
		`android.permission.GRANT_RUNTIME_PERMISSIONS`
		`android.permission.DUMP`
		`android.permission.CAMERA`
		`android.permission.SYSTEM_CAMERA`
		`android.permission.MANAGE_PROFILE_AND_DEVICE_OWNERS`
		`android.permission.MOUNT_UNMOUNT_FILESYSTEMS`
HIGH	7.5	`android.permission.INSTALL_GRANT_RUNTIME_PERMISSIONS`
		`android.permission.READ_SMS`
		`android.permission.WRITE_SMS`
		`android.permission.RECEIVE_MMS`
		`android.permission.SEND_SMS_NO_CONFIRMATION`
		`android.permission.RECEIVE_SMS`
		`android.permission.READ_LOGS`
		`android.permission.READ_PRIVILEGED_PHONE_STATE`
		`android.permission.LOCATION_HARDWARE`
		`android.permission.ACCESS_FINE_LOCATION`
		`android.permission.ACCESS_BACKGROUND_LOCATION`
		`android.permission.BIND_ACCESSIBILITY_SERVICE`
		`android.permission.ACCESS_WIFI_STATE`
		`com.android.voicemail.permission.READ_VOICEMAIL`
		`android.permission.RECORD_AUDIO`
		`android.permission.CAPTURE_AUDIO_OUTPUT`
		`android.permission.ACCESS_NOTIFICATIONS`
		`android.permission.INTERACT_ACROSS_USERS_FULL`

		android.permission.BLUETOOTH_PRIVILEGED
		android.permission.GET_PASSWORD
		android.permission.INTERNAL_SYSTEM_WINDOW
MEDIUM	5	android.permission.ACCESS_COARSE_LOCATION
		android.permission.CHANGE_COMPONENT_ENABLED_STATE
		android.permission.READ_CONTACTS
		android.permission.WRITE_CONTACTS
		android.permission.CONNECTIVITY_INTERNAL
		android.permission.ACCESS_MEDIA_LOCATION
		android.permission.READ_EXTERNAL_STORAGE
		android.permission.WRITE_EXTERNAL_STORAGE
		android.permission.SYSTEM_ALERT_WINDOW
		android.permission.READ_CALL_LOG
		android.permission.WRITE_CALL_LOG
		android.permission.INTERACT_ACROSS_USERS
		android.permission.MANAGE_USERS
		android.permission.READ_CALENDAR
		android.permission.BLUETOOTH_ADMIN
		android.permission.BODY_SENSORS
LOW	2.5	android.permission.DOWNLOAD_WITHOUT_NOTIFICATION
		android.permission.PACKAGE_USAGE_STATS
		android.permission.MASTER_CLEAR
		android.permission.DELETE_PACKAGES
		android.permission.GET_PACKAGE_SIZE
		android.permission.BLUETOOTH
		android.permission.DEVICE_POWER
NONE	0	android.permission.ACCESS_NETWORK_STATE
		android.permission.RECEIVE_BOOT_COMPLETED
		android.permission.WAKE_LOCK
		android.permission.FLASHLIGHT
		android.permission.VIBRATE

Table 2: Permission Mapping to Category and Weight

4.3 Cleartext Traffic Risks

Apps that use cleartext traffic in their communication with the outside world add risk to the user using them. This signal can be easily determined from the app's manifest, and thus scoring is possible.

The criteria we are measuring is if an app that targets API 28 and above declares `android:usesCleartextTraffic="true"` in its manifest or the app targets API 27 and below and does not declare `android:usesCleartextTraffic="false"`. In our implementation, we rely on Android's PackageManager [10] during runtime to tell us about whether the `FLAG_USES_CLEARTEXT_TRAFFIC` [9] is set for the application in question or not. This flag is added in API level 23 which is the minimum device API version supported by Uraniborg.

4.3.1 Formula

Here, we define the risk value to be the proportion of apps that intend to use cleartext traffic in their network communication on any given build as compared to that on a corresponding baseline build.

Again, we instantiate the general formula (as defined in Equation 1), which works quite well directly in our computation for this category:

$$\beta_{ct} = \lambda_{ct} \times \phi_{ct} \tag{12}$$

$$= \frac{|A_{t,ct} \setminus A_{0,ct}|}{|A_{t,ct}|} \times 1 \tag{13}$$

where:

$A_{t,ct}$ is the set of apps on a target device that targets API level 28 and above and declares `android:usesCleartextTraffic="true"` in its manifest,

$A_{0,ct}$ is the set of apps on a corresponding baseline build that fulfil the same criteria.

5 Conclusions & Future Work

In conclusion, we presented a framework to numerically quantify and estimate the risks posed by an Android device or build via the customizing and adding of preloaded apps. Our comparison is relative to a baseline, for which we selected the AOSP or GSI builds.

While this is the first step in our attempt to quantify the risk of preloaded apps on an Android device, we hope that the framework continues to evolve and welcome feedback on the ways we can improve it and make the overall scoring as robust and accurate as possible.

References

[1] AOSP. *About the Android Open Source Project*. Aug. 2020. URL: https://source.android.com/.

[2] AOSP. *Android Compatibility Definition Document*. URL: https://source.android.com/compatibility/cdd (visited on 08/03/2020).

[3] AOSP. *Building GSIs*. Aug. 2020. URL: https://source.android.com/setup/build/gsi#building-gsis.

[4] AOSP. *Compatibility Test Suite*. Aug. 2020. URL: https://source.android.com/compatibility/cts.

[5] AOSP. *Downloading the Source*. Aug. 2020. URL: https://source.android.com/setup/build/downloading.

[6] AOSP. *Generic System Images*. Aug. 2020. URL: https://source.android.com/setup/build/gsi.

[7] AOSP. *Privileged Permission Whitelisting*. Aug. 2020. URL: https://source.android.com/devices/tech/config/perms-whitelist.

[8] Android Device Security University Consortium. *Android Device Security Database*. Aug. 2020. URL: https://www.android-device-security.org/.

[9] Android Developers. *ApplicationInfo#FLAG_USES_CLEARTEXT_TRAFFIC*. Aug. 2020. URL: https://developer.android.com/reference/android/content/pm/ApplicationInfo#FLAG_USES_CLEARTEXT_TRAFFIC.

[10] Android Developers. *PackageManager*. Aug. 2020. URL: https://developer.android.com/reference/android/content/pm/PackageManager.

[11] Android Developers. *Permissions Overview*. Aug. 2020. URL: https://developer.android.com/guide/topics/permissions/overview.

[12] Billy Lau. *Uraniborg*. Aug. 2020. URL: https://github.com/android/security-certification-resources/tree/ioxt/ioXt/uraniborg.

[13] René Mayrhofer et al. *The Android Platform Security Model*. 2019. arXiv: 1904.05572 [cs.CR].

[14] J.D. Meier. "Improving Web Application Security: Threats and Countermeasures". In: Step 6 Rate the Threats, available at: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648644(v=pandp.10)#step-6-rate-the-threats. Microsoft Press, 2003. Chap. 3.