# Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android

David Barrera Jeremy Clark Daniel McCarney
Paul C. van Oorschot

School of Computer Science
Carleton University

## ABSTRACT

We provide a detailed analysis of two largely unexplored aspects of the security decisions made by the Android operating system during the app installation process: update integrity and UID assignment. To inform our analysis, we collect a dataset of Android application metadata and extract features from these binaries to gain a better understanding of how developers interact with the security mechanisms invoked during installation. Using the dataset, we find empirical evidence that Android's current signing architecture does not encourage best security practices. We also find that limitations of Android's UID sharing method force developers to write custom code rather than rely on OS-level mechanisms for secure data transfer between apps. As a result of our analysis, we recommend incrementally deployable improvements, including a novel UID sharing mechanism with applicability to signature-level permissions. We additionally discuss mitigation options for a security bug in Google's Play store, which allows apps to transparently obtain more privileges than those requested in the manifest.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls, Access controls*

## Keywords

Software installation, digital signatures, Android

## 1. INTRODUCTION

The wide-spread adoption of smartphones and tablets has popularized third-party app development and consumption. This new app-centric culture has encouraged users to install an assortment of task-specific apps on their devices. The average Google Android user has 32 installed apps, while the average Apple iOS user has 44.[1] In this paper, we consider the security decisions made by the Android operating system (OS) during the app installation process. We perform an in-depth study of the Android OS due to its popularity (at the time of writing, the smartphone OS with the largest market share) and open architecture.

We present a detailed description of the Android installation process, breaking it into three stages: *Update Integrity* (whether to treat the installation as a new app or as an update, overwriting the previous version), *UID Assignment* (whether to assign the app a new Linux user ID (UID) or allow the app to run under an existing UID), and *Permission Assignment* (which set of permissions are granted to, or inherited by the newly installed app). To our knowledge, the first two stages have been largely unexplored in the literature.

We believe it is important for research to be grounded in real-world practices, especially in the case of Android's widespread adoption. Any recommended OS changes which are compatible with current practices are more likely to be adopted and to have higher impact than blank-sheet theoretical approaches. To better understand how developers package and sign applications, we design and explore a customized repository of meta-data and app components extracted from a representative set of Android apps. Our dataset currently includes metadata, packaging and code signing information for app packages collected from 7 different sources, including app markets, filesharing networks and malware repositories.

Our repository allows us to use these components to infer relationships *between* apps. For example, starting with a single app, we can query information about the app, all versions of this app both within and across marketplaces and other app sources, all other apps signed with the same key (or key set), and all other apps that share compiled code, resources or similar manifest entries.

Some notable findings from our dataset are: 18% of developers signed more than one app with the same key; 4% of apps obtained from file-sharing networks have direct links to known malware; 2 infected apps were found live on the Aproov app store; a publicly available test key was used to sign 291 apps in our dataset, including 51 malicious apps and 15 apps on the Google Play Store, posing a risk to update integrity; and due to a peculiarity in the current Android implementation, apps sharing a UID can display no requested permissions and still perform privileged operations — in one

[1]The Mobile Media Report, *Nielsen*, Q3 2011


*SPSM'12,* October 19, 2012, Raleigh, North Carolina, USA.


`http://blog.nielsen.com/nielsenwire/online_mobile/the-state-of-mobile-apps/`

case observed from the dataset, obtaining full internet access, location data and personal data.

We plan to release our full database of extracted app components (and provide a web-interface which we call the Android Observatory[2] that allows a number of pre-constructed queries to be made against the database). Our dataset currently contains information from 11,000 app packages and is the first dataset to include apps from the Amazon Appstore, the MiKandi adult app market, the Aproov app store, and the F-Droid open source repository. This modestly-sized dataset allows representative observations to be made about developer practices, which informs our analysis and suggestions. Gathering a larger dataset, which is not our current focus, may nonetheless be of interest to the community to allow complementary ongoing research.

Outside of the installation process, security mechanisms for Android [5, 12] are also applied before and after installation (*e.g.,* app store vetting or run-time analysis respectively). Security mechanisms that are applied pre-installation [3, 14, 17, 7, 31] and post-installation [26, 13, 19, 9, 6] are important to the overall Android security architecture, but beyond our present scope. Permission assignment has been extensively studied in the literature [15, 17, 24, 23] but the other portions of the installation process—update integrity and UID assignment—remain largely unexplored in the research literature prior to our work. See the related work in Section 6 for additional details.

Our primary contributions are as follows.

1. We conduct an empirically-informed systematic analysis of the security mechanisms employed during Android app installation. These mechanisms include signature verification and UID assignment, and how they relate to granting (and possible inheritance) of permissions.

2. We propose a mechanism for allowing distinct developers to share an Android UID while maintaining authorship and control over updates to their respective applications. Our proposal is incrementally deployable and congruent with the developer practices seen in our data. The proposed mechanism can also augment the flexibility of signature-permissions.

3. We identify and describe a flaw in the Google Play app that enables apps sharing UIDs to obtain higher privileges than those declared in their manifest. We call these attacks "permission inheritance" and "retroactive permission inheritance" and we explore possible prevention mechanisms.

Section 2 reviews the Android security model and app installation process. We provide an empirically-based analysis of update integrity on Android in Section 3. In Section 4, we discuss the limitations of Android's current UID sharing mechanism, and propose improvements. Section 5 discusses permissions as they relate to UID sharing and signature-level protections. We present related work in Section 6 and conclude in Section 7. In Appendix A we provide more detail on our dataset and web-interface.

[2]The name is inspired by the SSL Observatory [10], which is in the same spirit as our own Observatory.

# 2. ANDROID PRELIMINARIES

For context, we first review background details [16, 29]. Android libraries and middleware run on top of a Linux kernel. Thus, many OS and security properties are inherited from the underlying Linux architecture. We first briefly review the Android security model. We then detail the app installation process, noting that it differs significantly from standard Linux software installation.

### *App Packages.*

An Android app package (apk file) is an archive. It contains a Dalvik executable (dex file), which is a compiled Android program that runs on a Dalvik virtual machine, and a set of resources (non-executable files like graphics, media files, user interface components, *etc.*). App packages also contain a manifest (`AndroidManifest.xml`), which on Android contains meta-information about the app like package name, version, supported Android versions, and other attributes. These components are digitally signed with the developer's signing key. The developer's signing certificate can be self-signed and is included in the application package.

### *Android Security Model.*

The Android security model is based on the Application Sandbox.[3] Android enforces isolation between sandboxes, preventing apps from modifying or otherwise interfering with each other. Android assigns a unique Linux user ID (UID) to each sandbox, which may contain one or more apps (see Section 4). This mechanism allows apps to have private file storage as well as private memory and process space, all enforced by the kernel.

If necessary, developers request special privileges (*e.g.,* use of the camera, GPS sensor, microphone) for their app by including a list of permissions in the app's manifest. The manifest is read at installation time, and the user is prompted to accept or reject the permission set. On stock Android builds, permissions cannot be individually rejected. Thus, developers assume that once their app has been installed they can make use of all the features protected by the permissions requested.

Codesigning is used in conjunction with the app sandbox to provide several fundamental aspects of the Android security model. The developer's certificate is used to restrict who can issue software updates to the app (described in detail in Section 3), the app's inter-process communication (IPC) abilities, and whether certain permissions can be obtained. As an example of the latter, the `MASTER_CLEAR` permission allows an app to delete all user data and restore the device to its factory state. While the `MASTER_CLEAR` permission can be requested by any app, it can only be granted to apps signed with the same key as the system image. These permissions are called `signature` or `signatureOrSystem` permissions. Third-party developers may additionally write public APIs and protect them with signature-level permissions.

## 2.1 Deconstructing App Installation

Android allows any developer (even those who have not registered with Google) to create and distribute apps. Apps can be distributed through the official Google Play Store, through third-party markets (*e.g.,* Amazon Appstore) or

[3]`http://developer.android.com/guide/topics/security/security.html`

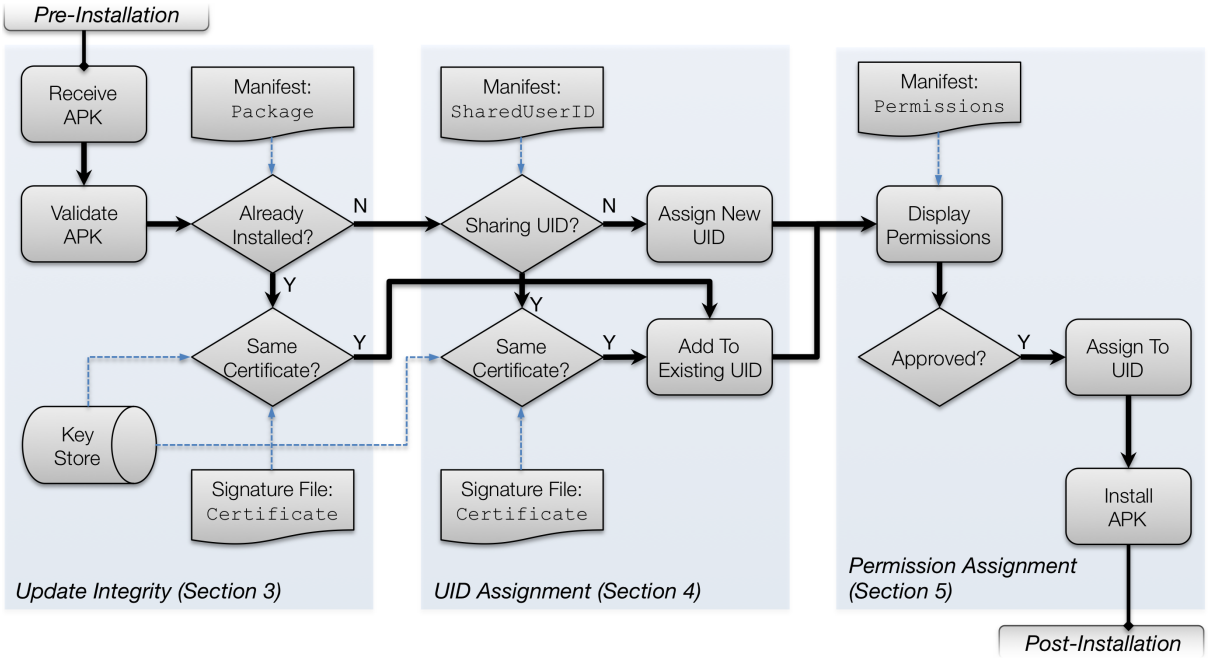



**Figure 1: Our abstract model of the Android installation process for an app package (apk). Unless otherwise specified, an answer of no to any conditional aborts the installation.**

through developer websites (side-loading). The lack of control over the app distribution process raises the importance of enforcing security within the Android OS.

When a new app is loaded for installation, either through a third-party market or side-loading, the sequence of events is as we depict in Figure 1. When apps are installed through the Google Play Store, permissions are approved prior to installation (see Section 5.1), but the rest of the process remains the same.

First, the app package validity is verified: the system ensures that the Android app package has not been modified or corrupted since being signed, and that it contains a valid certificate for the signing key. Next, Android decides whether the app is a new installation or should overwrite an existing app. If the app being installed has the same `package` attribute in the manifest (*e.g.,* `com.google.android.music`) as another currently installed app, then Android will treat the installation as an update. In this case the certificate (or set of certificates if signed by multiple keys) is compared to the certificate(s) of the already installed app. If both apps were signed with the same key(s), then the currently installed binary is removed (preserving any user data) and the new app is installed in its place. Otherwise, the new app is installed as an initial installation.

Next Android must assign a UID to the app. In this case, the previous app's UID is used. In the case of an initial installation, Android checks if the app's manifest contains the `sharedUserId` directive. If so, Android looks for any other installed apps that are signed with the same key(s) and also have `sharedUserId` specified in their manifest. If such apps are found, the app is assigned the same UID; otherwise a new UID is created.

Finally permissions must be assigned to the UID. The user is prompted to review and approve the permission assignments before the app is installed. When UID sharing is not used, permissions listed in the app's manifest are assigned to the UID. When UID sharing is used, the UID is assigned the union of all permissions in the manifests of apps sharing the UID. If the app is updating an already installed app, the permissions listed in the updated app's manifest are assigned to the UID.

## 2.2 Empirical Dataset

To inform our research, we constructed a dataset intended to roughly represent what typical Android users would be exposed to when searching for and installing apps. To obtain this diversity, we crawled, downloaded and parsed apps from three types of repositories. See Appendix A.2 for details on the types of apps contained in each repository, crawling methodology and rationale for inclusion in our dataset.

1. **Official and alternative app markets (8612 binaries)**: Google Play Store (free apps, 6079 binaries), Amazon Appstore (paid apps, 39 binaries), F-Droid (open-source apps, 647 binaries), MiKandi (adult free apps, 104 binaries), Aproov (free apps, 1743 binaries).
2. **File sharing networks (2283 binaries)**: 3 application packs found on the Bittorrent network.
3. **Malware (209 binaries)**: Infected apps contributed by security researchers and anti-virus firms

Our data collection process involves writing a custom crawler or downloader for each application repository. We note that in contrast to other researchers (particularly those focused on permission analysis), we must obtain the full Android application package instead of screen-scraping. Our analysis depends heavily on code-signing details which are not displayed on web-facing app repository front-ends. Downloading the full apk files allows us to peer into the

archive and extract signatures and certificates, metadata, compiled code and resources.

# 3. APP UPDATE INTEGRITY

App signing is the primary security mechanism that protects the integrity of the app after it is released by the developer, ensuring that only the developer can issue an update to an already installed app. For an app to be installable on Android, the app package must be digitally signed with at least one developer's signing key. Packages generally contain a self-signed developer certificate binding the public signing key to developer-supplied information. If an update contains the same package name but is signed with a different key, the Android OS will not allow it to install. Users can only install such an update by first removing the current version of the app, which also removes all user data associated with it.

In this section, we analyze the specifics of app signing in Android. With insight from our dataset, we show how the current model has limitations regarding certificate expiration, revocation and key evolution. We discuss alternatives and enhancements, and offer some cautious recommendations.

## 3.1 Signing Details

We were not able to locate documentation defining the precise process used to sign all the resources in an Android app. We consequently reconstruct it for the current (as of writing[4]) version of Android. App signing is handled by the `jarsigner` tool supplied with Java development tools. The developer generates a self-signed certificate, which can include standard X.509 certificate attributes like a common name, organization, location and validity period. The signature algorithm can be either RSA or DSA. Out of 4,141 distinct signing certificates observed in our data set, 96% used RSA and 4% used DSA. RSA is the default for the Android Development Tools. The certificate is stored in the `META-INF/NAME.RSA` file (or `.DSA`).

A manifest of every file in the app package (each resource, the binary, and the app manifest) is created by `jarsigner` in `META-INF/MANIFEST.MF`. This manifest file should not be confused with the app manifest, `AndroidManifest.xml`, which specifies developer-supplied meta-information about the package (*e.g.,* app name, version, permissions, *etc.*). For each file, `MANIFEST.MF` includes an entry with its path and a SHA1 hash.

Next, a signature file `META-INF/NAME.SF` is generated by `jarsigner`. The signature file may include a hash of `META-INF/MANIFEST.MF` and must include an individual hash of each entry in `MANIFEST.MF`[5] along with the path to the file the entry concerns. This level of indirection (between `MANIFEST.MF` and `NAME.MF`) is redundant in Android apps as the manifest entries themselves are only a file path and hash, however this may not be the case for general JAR archives. Finally, the file `NAME.SF` is hashed and signed. This signature is appended to the certificate found in `META-INF/NAME.RSA` (which itself is already signed). When apps are installed, the OS verification process checks the signature on `NAME.SF` using the public key in `NAME.RSA`, and the correctness of the hashes in `NAME.SF` and `MANIFEST.MF` against the files in the package.

[4]Android 4.0.4 (Ice Cream Sandwich).

[5]`MANIFEST.MF` entries are delimited with `0x0D0A0D0A`.

### *Authentication Model.*

Android takes a trust-on-first-use approach to app signing (aka leap-of-faith authentication [2]). The identity of app developers is not authenticated but if an app is first installed from a legitimate developer, subsequent updates can be recognized as being from the same developer. By contrast, in a full authentication model, a developer's public key is bound to her identity by an authoritative party. Android's recognition-based approach is lightweight,[6] requires no certificate authorities, or centralized PKI of any form (*cf.* [1, 25, 22]). It is sufficient for preserving update integrity given a correct initial installation. Its drawback is its inability to provide trust on an initial installation.

Android will install apps with CA-issued certificates. In this case, the CA certificate must also be included in `META-INF/NAME.RSA` with the developer certificate. Of the over 4000 certificates in our dataset, only 5 instances in our dataset had a CA-issued certificate: three were issued by Sprint Vision, one by Thawte Premium, and one by Thawte Personal Email. Android does not reference a list of trusted root CAs during installation, and it will accept a self-signed certificate that authorizes itself as a CA (*i.e.,* sets `basicConstraints=CA:true`).

### *Signature Stripping.*

A modification to a signed app will cause verification of the original signature to fail, making the signature useful for preventing passive file corruption. However the original signature can be removed and replaced with a new signature and self-signed certificate. For new installations, Android will accept any signature accompanied by a self-signed certificate (by contrast, in standard iOS installation, apps must be signed by Apple [11]).

As one apparent example of how signature stripping can be useful to an adversary, our dataset contains multiple versions of the app Baseball Superstars (`com.gamevil.bs2010`) from the Google Play Store, all signed with the same key. We also observed another instance of this app in the Contagio malware dump. It contained the same package name and resources but had a different Dex file (one that contained the malware `Geinimi`[7]), a larger permission set, and a different signature. While we cannot conclusively classify this as an example of signature stripping, it is consistent with a malicious developer stripping the signature, weaponizing the Dex file, resigning and redistributing the app.

### *Maintaining a Reputation.*

The recognition-based approach used by Android can allow a developer to build an unforgeable reputation across apps. Of 4141 certificates used to sign apps, 18% were used to sign more than one unique app. Google signed all of their apps with one of two keys.[8] With a tool like the Android Observatory (the web-interface to our dataset described in Appendix A) to illustrate the set of apps signed with the same key, this reputation can result in positive security as-

[6]In the categorization of Goldberg *et al.*, it is a stateless non-interactive message recognition protocol which is shown to be one-to-one with a digital signature scheme [22].

[7]`http://blog.mylookout.com/blog/2010/12/29/geinimi_trojan/`

[8]SHA1 certificate fingerprints:
`38918A453D07199354F8B19AF05EC6562CED5788`
`24BB24C05E47E0AEFA68A58A766179D9B613A600`

sertions such as: 'the entity which signed this relatively unknown app is the same entity signing another trusted app;' or negative ones: 'the entity which signed this app is also the signer of another app with known malware.'

As an example of the latter, we found in our dataset instances of the same signature applied to both apps available on the Aproov marketplace and a different app with known malware from the Contagio malware repository. Using Virus Total, we scanned the Aproov apps (`com.zft` and `com.droiddream.musicbox`) and detected malware (`TrojanSpy:AndroidOS/DroidDream.A`). We notified the Aproov marketplace of the malware and the apps have since been removed from the market. Generally with a dataset like ours (and the web-interface we have built for it), it is easy to see apps signed with the same key. We anticipate that developers of malicious apps will, if they do not do so already, use a new unlinkable certificate for each app to prevent detection via this method.

While we see no long-term benefit of code-signing for detecting malicious apps, it is very useful for building a positive reputation as a developer. One drawback of this approach is that it may lead development teams to use a single reputable key as often as possible, *e.g.,* by sharing the key and setting unreasonably long certificate expiration periods, which could increase its exposure to internal compromise by a single rogue developer or external compromise by having it replicated on multiple platforms.

## 3.2 Alternative Signing Key Management

In this section, we analyze the Android app signing model relative to other PKIs. We find theft-resistant measures and deterrents from traditional PKIs like certificate expiration and revocation are not available in Android, nor is any mechanism for changing signing keys. Allowed measures like signing apps with multiple keys are rarely used. We also consider various ways the current model could be augmented. As is common in PKI design, each augmentation presents a trade-off as opposed to an unequivocal improvement.

### *Full Authentication.*

One alternative to the entire reputation-based model is to use a full-fledged PKI where developers prove their identity to a certificate authority (CA) and are issued a certificate. In this model, the certificate information can link developers together as being the same person or from the same team without needing to use the exact same key to sign every app. While this is beneficial, the added complexity makes the model a poor tradeoff in our opinion. It would require developers to obtain CA certificates and Android to decide on the set of trustworthy CAs as well as make security decisions (or require the user to) based on certificate attributes. Being a heavy-weight change for limited benefit, and one that does not work particularly well in other domains (e.g., SSL/TLS), we do not recommend it.

### *Certificate Tree.*

The self-sign model can be combined with a constrained certificate hierarchy we refer to as a one-level certificate tree. A development team (or single developer) would establish a self-signed long-term CA certificate as the root of a one-level tree. The signing key would be well-protected and used only infrequently to issue shorter term certificates for actually signing apps. The update process would still use the leaf certificate in the tree to decide on allowing an update to an existing app but the root certificate could be used to establish a shared reputation amongst all certificates it issues. In a team environment, this could be a shared reputation across distinct developer certificates. With a single developer, it could be a shared reputation amongst distinct app certificates.

A certificate tree can reduce the need to replicate copies of the same key. It can also be useful if a developer wants to transfer ownership of a signed and widely installed app to another developer. If the signing key for an app is transferred to the buyer, the buyer can issue updates to all apps signed with the key. A certificate tree can allow individual signing keys for distinct apps while still allowing the developer to maintain a reputable link to their other apps. It is not a fully satisfactory solution as transferring a key does not sever the link to the CA certificate.

Certificate trees require no changes to Android but redefine the purpose of a CA. Typically, if you hold a CA-issued certificate, this is interpreted as the CA validated your identity. Here, CA-issued certificates are used to collect several signing keys under one shared reputation. For the latter type of CA to distinguish itself from the former, they could use a naming convention that would be obvious (or, more formally, use an X.509 extended attribute).

Using our dataset, we found one example of a complementary approach to a certificate tree used (ostensibly[9]) by Sprint. Sprint established a CA certificate for *Sprint Vision Root CA*, which itself issued a CA certificate for *Sprint App CA*. The app CA issued a shorter lived certificate used to sign two different apps in our dataset. We did not see more than one certificate issued by the App CA but this is a similar hierarchy to the tree approach.

### *Certificate Expiration.*

Android requires the app signing certificate to specify a validity period but Android does not currently enforce expiration through preventing app installation. We found 2% of the 4,141 observed signing certificates in our dataset were expired. If expiration were enforced, the reputation model would still encourage developers to specify a long validity period for their certificate. The Android documentation encourages this (it is still written as if expiration were enforced) and claims that the Google Play Store explicitly requires certificates to be valid until at least 2033[10] (there are examples to the contrary in our data from the Android Marketplace but we did not verify if this requirement is presently enforced). Long expiration periods are a poor security practice, and having no enforced expiration means a compromised or stolen signing key can be used indefinitely (unless a revocation mechanism exists, like kill switches discussed below).

### *Signature Key Updates.*

An unintended consequence of trust-on-first-use is that the first app signed with a new key appears suspicious even if the key update is legitimate. To ensure a developer can continue to update existing apps after switching signing keys

[9]The issue with self-signed certificates is, of course, that you can never be certain who issued them.

[10]The Developer's Guide: Signing Your Applications `http://developer.android.com/guide/publishing/app-signing.html`

(due to certificate expiration or other reasons) without involving the user, the developer could use their existing key to sign their newly issued certificate. Establishing certificate continuity in this way will also allow a clean transfer of ownership of an app (*cf.* key evolution in Bin-locking [30]).

In our dataset, we observed one high-profile example of a key update where Google attempted to switch the signing key for its two-step authentication app called Google Authenticator.[11] As mentioned, Google typically signs their apps with one of two keys. In the case of Google Authenticator, Google released an update that switched from one of their commonly used signing keys to their other key. Since Android has no mechanism for updating keys, users cannot install such an update directly. To cope with this shortcoming, developers (in this case Google) must ask users to uninstall the current version before installing the update. This effectively turns the "update" into a new installation with the same package name, causing all user data to be lost.

If data preservation is necessary, developers need to write a custom data backup mechanism. In the case of Authenticator (`com.google.android.apps.authenticator`), Google opted to release the update as an entirely new app with a new package name (`com.google.android.apps.authenticator2`). Users were instructed to install `authenticator2` alongside the existing `authenticator`, and `authenticator2` would copy the data from `authenticator` using a custom API built into the last release of `authenticator` to do authenticated inter-process communication (IPC). The user was then prompted to uninstall `authenticator`.

This case is interesting because it illustrates two drawbacks in different parts of the Android installation process: (1) the inability to update an existing key, which we have discussed in this section on update integrity; and (2) the inability for two apps, which in this case must necessarily be assigned different UIDs, to securely share data using OS services, which we discuss in Section 4 on UID assignment.

### *Revocation of Signing Keys.*

There is no documented mechanism for key or certificate revocation. Compounded with certificate expiration not being enforced, the only way to limit the use of a stolen signing key in issuing malicious upgrades is through marketplace removal of apps signed with the key and using a kill switch [16, 4] which is also implemented at the marketplace level. For side-loaded apps, neither of these techniques will work. An OS update could blacklist keys that pose a significant threat outside of maintained app markets.

### *Distributed & Threshold Signing.*

An approach to protecting against the theft of signing keys is to use distributed signing, where $n$ participants individually sign the app and all $n$ participants are required to sign each update (*cf.* TUF update framework [27]). Android currently supports distributed signing. For each signature, an individual signature file (*e.g.,* `KEY1.SF`) is created and signed, appending the signature to individual certificate files (*e.g.,* `KEY1.RSA`). Android will allow updates only if the same exact set of signatures are used with each update. Of all the apps in our dataset, only 42 had more than one signature. For 30 of these, the additional certificate was a temporary (and expired) debug certificate created by the Android app development tool for testing purposes. For 4 of them, a publicly available key pair was used—see Section 3.3 for discussion of this case. This leaves a small set of apps that appear to be using distributed signing (in each case, a 2-out-of-2 access structure) for security purposes. One example is the Mint app (`com.mint`), a financial management service.

In the current model, there is no cross-certification between the multiple signatures. This means signatures can be selectively stripped in distributed signing instances [30]. We verify that this is the case even if the first signatures applied are included in the signature file for subsequent signatures (*e.g.,* `KEY2.SF` includes `KEY1.SF` and `KEY1.RSA`). In one app from Sprint, (ostensibly) HTC added a signature that updated `MANIFEST.MF` with Sprint's `SF` and `RSA` files.[12] We stripped Sprint's signature and verified that the validation tool ignores files recorded in `MANIFEST.MF` that are located in the `META-INF/` subdirectory.

Android could be modified to allow apps to specify in `AndroidManifest.xml` (a file that is explicitly signed) the intention to use multiple signatures signing and pin the set of public keys required to update the app (*cf.* certificate pinning[13]). A simple extensions is to also permit developers to specify a $t$-out-of-$n$ access structure that requires the signatures of any $t \leq n$ of a set of $n$ signatures. This maintains the security of distributed signing while adding robustness against lost or unavailable keys [21].

For completeness, we note that threshold (and distributed) signing can be conducted externally to `jarsigner`. Threshold signing protocols are known for RSA signatures, both with a trusted dealer that generates the key shares [28] and with distributed key generation [8]. To the best of our knowledge, no tool is available that implements a threshold signature scheme and easily allows developers to perform such a protocol.

### *Summary.*

We cannot make any strong recommendations for modifying how Android provides update integrity through app signing that would preserve backwards compatibility with the majority of existing applications in our dataset. However, we hope that our discussion of the limitations and alternatives to update integrity will be informative for the design of future systems that rely on app signing (*e.g.,* browser extensions or Mozilla's Firefox Mobile OS[14]).

## 3.3 Publicly Available Key Pairs

In our dataset, we noticed one set of 291 apps, including 58 with known malware, were signed with the same key.[15] Upon further investigation, we found that this particular key belongs to a publicly available key pair that is distributed

[11] `http://support.google.com/a/bin/answer.py?hl=en&answer=1037451`

[12] This changes the hash of `MANIFEST.MF` from the one reported in Sprint's `SF` file, which the verification tool ignored. It appears to only verify the `MANIFEST` entries themselves.

[13] IETF Internet-Draft: HTTP Strict Transport Security (HSTS)

[14] `http://blog.mozilla.org/blog/2012/07/02/firefox-mobile-os/`

[15] SHA1 certificate fingerprint: `61ED377E85D386A8DFEE6B864BD85B0BFAA5AF8`

with the Android source code.[16] This test key is often used to sign third-party Android builds. The danger in releasing an app signed with only this key is that anyone can issue an update for the app that is acceptable to the Android OS. We observed apps signed with this key across multiple marketplaces, including 15 on the Google Play Store. It seems clear that all marketplaces should refuse to host or distribute apps signed with this test key.

# 4. UID ASSIGNMENT

In Android's current security model, new apps are assigned unique UIDs to enforce filesystem and process isolation, as well as grant access to privileged device capabilities (see Figure 1). Indeed, UIDs are an essential component of the Android security model. Android allows developers to write apps that share a UID. To use this feature, developers set the `sharedUserId` directive to a common string value in the `AndroidManifest.xml` file of each app that will share the UID. The apps must be signed with the same key (or set of keys) for Android to allow a shared UID. The common string allows for apps signed with the same key to have multiple, distinct shared UIDs.

While Android already allows developers to write modular apps that integrate with each other via interprocess communication (IPC) [16], UID sharing allows developers to split app functionality into multiple installable pieces, yet share binary resources such as fonts, images and sound clips. Android itself uses shared UIDs for system apps to interact with other system components. For example, the `sharedUserId` "`android.uid.system`" is used by the Settings, VPN, and AccountsAndSync settings in stock Android apps.

In our dataset, we observed 117 apps utilizing UID sharing and 60 distinct shared UID strings. Most of these were apps with modular designs such as browsers and associated extensions with one main component and several smaller plugins. Some examples are language packs for a dictionary (package name `com.socialnmobile.dictapps.notepad.color.note`), extensions for a browser (package name `mobi.mgeek.TunnyBrowser`), and additional fonts for a message app (package name `com.handcent.nextsms`). A small number appear to be sharing UIDs without apparent reason.

## 4.1 Improving Android's UID Sharing

The main limitation of UID sharing as currently implemented by Android is its dependency on app signing, which precludes apps signed with different keys from sharing a UID. First, this conflicts with the reputation-based model implicitly encouraged by Android's app signing protocol (see Section 3). Second, it encourages developers to pursue integration through custom mechanisms, such as IPC which by itself offers no authentication and is more open to developer error. While developer-defined permissions can protect access to an API, either (1) any app can be granted the permission or (2) only apps signed with the same key can be granted the permission (see Section 2). This dichotomy is a direct result of permissions in Android being assigned to UIDs.

One example of UID sharing observed in our dataset was a browser that shared its UID with browser extensions. In this example, it is natural to consider that the extension may be authored by a different developer than the browser, and yet the browser's developer must sign the extension for it to function correctly. This could be problematic for the browser developer as code signing is implicitly endorsing someone else's code, and it also fails to allow the extension developer to transfer any reputation from other apps they have released. The extension developer also loses attributed authorship.

### *Properties for UID Sharing.*

Android's current model for UID sharing achieves a number of important properties which we have distilled into Properties 1 to 6 below (we denote apps that share a UID as members of a group):

1. **Groups are Disjoint:** Processes must run under a single UID so there is no way for an instance of a running app to be a member of two or more groups.
2. **Groups are Consistent:** If app A is in the same group as B, and app A is in the same group as C, then B and C must be in the same group.
3. **Group Size is Arbitrary:** It is possible to specify UID-sharing groups of size 1, 2 or greater than 2.
4. **Membership is Authorized:** An app cannot join a group without being authorized to join.
5. **Adding New Members is Efficient:** If a new member is added to a group, ideally no other group members require an update. In the next best case, only one app requires update (denoted ∘ in Table 1). In the worst case, all members of the group require update.
6. **Different Groups with Same Key:** It is possible for apps signed with the same key to be members of different groups.

Ideally, a UID sharing mechanism would satisfy all 6 properties above as well as allow apps signed with different keys to share a UID (property 7). However, in the course of redesigning the mechanism, we found that in every case, adding property 7 precludes at least one other existing property. We therefore consider the relative merits of trading off various properties in order to achieve property 7.

### *Alternative Mechanisms for UID Sharing.*

We consider three possible alternatives to Android's currently implemented mechanism for UID sharing: mutual approval, pairs and parent-child (see Table 1 for a comparison matrix).

- **Mutual Approval:** Each app specifies in its manifest every other app in its UID sharing group. Apps in the group are specified by package name and fingerprint of their embedded signing certificate. This fulfills the goal of allowing apps signed by different keys to share a UID, but makes it difficult to add new members to the group as every app in the group must be updated. Mutual approval is also prone to group inconsistencies since it is possible that one app (A) approve two apps independently (B and C), without B and C mutually

[16] `platform_build/target/product/security/testkey.pk8`

| | Mutual Approval | Pairs | Parent-child | Signature-based |
|---|---|---|---|---|
| 1. Groups are Disjoint | • | • | • | • |
| 2. Groups are Consistent | | • | • | • |
| 3. Group Size is Arbitrary | • | | • | • |
| 4. Membership is Authorized | • | • | • | • |
| 5. Adding New Member is Efficient† | | ○ | ○ | • |
| 6. Different Groups with Same Key | • | • | • | • |
| 7. Different Keys within Same Group | • | • | • | |
| Viable | | | • | • |

**Table 1: A comparison of proposed UID sharing mechanisms to signature-based sharing (the current model).**
**† For new additions, • means only one member must be updated, ○ means two members must be updated, and empty means all members must be updated.**

approving each other. Android has no clear mechanism to resolve such inconsistencies, so we conclude mutual approval is not a viable replacement to the current model.

- **Pairs:** This alternative constrains groups to a maximum size of two and can be seen a sub-case of the Mutual Approval mechanism above. This restriction helps satisfy group consistency, but has the obvious disadvantage of not allowing groups larger than two apps. In our dataset, nearly all instances of apps sharing a UID were larger than two, so we do not consider pairs to be a viable replacement.

- **Parent-child:** In this alternative, one parent app (*e.g.,* a browser) authorizes in its manifest each other app (child) in the group. This resolves the group inconsistencies of mutual approval as well as our original goal of divorcing UID sharing from application signing. The main drawback of the parent-child mechanism is that that the parent app must be updated every time a new group member is added.

### *Implementing Parent-child UID Sharing.*

To implement the Parent-child mechanism, the core Android package installer framework[17] must be updated and a new set of XML attributes must be added to the official API. This means that our mechanism would need to be adopted by Google (or a popular third-party Android build) for wide-scale deployment. Google routinely updates their OS and manifest specification. For example, in API level 3, Google introduced the `sharedUserLabel` directive.[18]

We suggest adding the following XML attributes to the Android Manifest: `sharedUserIdMode, parentOf`, and `childOf`. `sharedUserIdMode` specifies if the application in question is intended to be a parent, a child, or to use the traditional signature-based UID scheme. Not specifying a `sharedUserIdMode` defaults to the traditional scheme for backwards compatibility with apps that are no longer updated. The `parentOf` attribute specifies (for the parent app) each of the child apps with which it should share a UID. Child apps are specified via package name and certificate fingerprint (see Figure 2). Finally the `childOf` attribute specifies the package name and certificate fingerprint of the parent app with which to share a UID (see Figure 3).

We note that some restrictions must be enforced by Android at install-time to guarantee integrity and consistency.

- An application must not declare both `parentOf` and `childOf`.

- Applications must only specify one of "parent", "child" or "signature" for the `sharedUserIdMode` attribute.

- Shared UID Labels are not required as in the traditional signature-based scheme, since both child apps and parent apps specify other apps in their manifest.

- At install-time, after verifying restrictions above, the installer looks for apps specified as `childOf` or `parentOf` in the newly installed app's manifest. If matching apps are found and manifests are consistent (*i.e.,* the parent and the child specify each other), child apps are updated to use the parent's UID.

```
AndroidManifest.xml (Browser)
<manifest xmlns:android="http://sche...com/apk/res/android"
. . .
 android:sharedUserIdMode="parent"
 android:parentOf:"com.cool.extension1, 67709F9D63BFD6A0"
 android:parentOf:"com.cool.extension2, 774A0E232918EE8B" >
. . .
</manifest>
```

**Figure 2: The Android Manifest of a browser using our improved UID scheme. The developer specifies that the browser can be the parent of two extensions.**

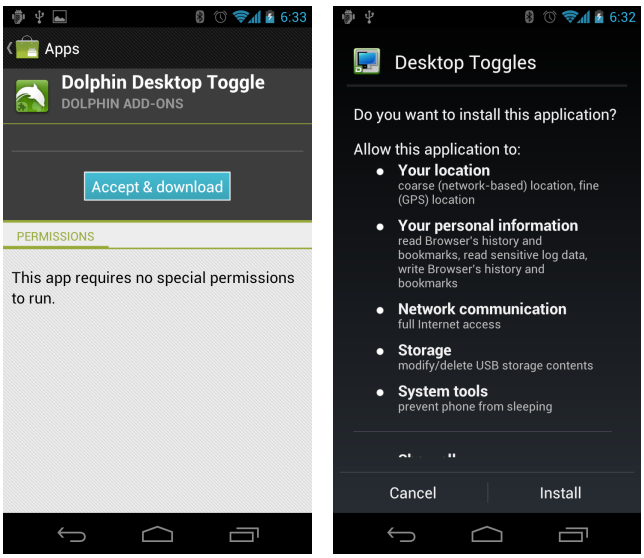
```
AndroidManifest.xml (Extension)
<manifest xmlns:android="http://sche../apk/res/android"
. . .
 android:sharedUserIdMode="child"
 android:childOf:"com.cool.browser, 0DCDD39F12A07A25" >
. . .
</manifest>
```

**Figure 3: The Android Manifest of an extension using our improved UID scheme. The developer specifies that the extension is a child of a browser, and certificate fingerprint of the browser.**

## 5. PERMISSION ASSIGNMENT

Android permissions have been extensively discussed in the literature [15, 17, 24, 23]. In this Section, we focus narrowly on permissions as they relate to UID sharing and signature-level permissions protecting IPC calls.

[17] `PackageManager.java` and `PackageManagerService.java`
[18] `http://developer.android.com/guide/topics/manifest/manifest-element.html`




(a) Market (b) Sideloaded

**Figure 4: Screenshots of permissions requested by installing the Desktop Toggle browser extension via the Google Play Store and when side-loaded. In both cases, the Dolphin Browser had been previously installed.**

## 5.1 Inheritance through UID sharing

Through its user interface, Android implicitly treats permissions as if they were assigned to apps. In reality, permissions are assigned to UIDs. This distinction is generally inconsequential, except in the case when apps share a UID. Upon installation of a new app which will share a UID with an existing app, the new app will be granted all the permissions that have been granted to the existing app in addition to its own requested permissions. We call this effect permission inheritance. Likewise, the existing app will be granted all the permissions associated with the new app. We call this retroactive permission inheritance.

An example of permission inheritance is shown in Figure 4. Here, we show two device screenshots taken during the installation of a browser extension which requests no permissions in its manifest, but shares a UID with a highly privileged browser. When the extension is installed from the official Google Play Store[19] (Figure 4a), no permissions are displayed. However, due to permission inheritance, the extension is granted the same permissions set as the browser. The correct permission set is only displayed when the extension is side-loaded (Figure 4b).

Permission inheritance inconsistencies occur when the application installer does not have an accurate, updated view of all installed apps on a device. Google keeps track of applications installed through the Google Play store, but does not know about side-loaded apps. If Google knew about all apps (installed and side-loaded) installed, the Google Play app should be able to accurately display the effective resulting permission set at install-time. The package installer invoked when side-loading does have an accurate list of all installed apps, and can therefore correctly display the permission list.

One possible solution to this problem is to always use the OS installer when installing apps that have the `sharedUserId` attribute set in their manifest. The user experience in these cases would suffer, since users would see an unfamiliar installation screen. The OS installer could additionally be augmented to display which permissions are requested by each app in the group, rather than only display the union of all permissions.

## 5.2 Signature permissions

While UID sharing allows for tight integration of separate installable apps, developers may want simpler IPC for offloading tasks to other apps (*e.g.,* send a tweet via any installed Twitter client), or to offer some functionality to other apps (*e.g.,* download the latest prices of a stock).

IPC on Android is generally slower than sharing a UID and is optimized for small data messages. We argue that while technically possible, moving large amounts of data over IPC is often impractical, and developers tend to rely on other mechanisms such as shared UIDs or making files world-readable on the device.

When writing Android apps, developers can define public interfaces, and protect calls to those interfaces with signature-level permissions. As expected, signature-level permissions can only be granted to the caller app if it is signed with the same key as the callee. Our parent-child manifest extensions for UID sharing could be used similarly for permission-based IPC calls by allowing apps to specify the package name and certificate fingerprint of apps that are allowed to invoke the declared methods. This change could also be deployed incrementally by defaulting to the traditional same-key configuration if the apps are not written to target the latest Android API level.

The Google Authenticator example of Section 3.2 serves as a case study for showing limitations of the currently deployed signature permission scheme. The primary failing in this case was that changing the Google Authenticator signing key requires a new installation, which if done in an improper order could lead to data loss. If both versions of Authenticator had used our signature-permission scheme, the developers could have used OS-level support for moving the sensitive user data from `authenticator` to the new version `authenticator2`. As it stands, Google developers were forced to do this over IPC and handle authentication between apps through custom code. To ensure that no app other than `authenticator2` could obtain this private data, Google implemented a custom API (under Java class name `dataexport.ExportServiceV2`) in `authenticator`, which manually checked that the signature on the requesting app matched `authenticator2` before granting access.

# 6. RELATED WORK

The smartphone security literature has matured considerably in recent years, including an introduction to Android security [16], general surveys of smartphone security [5, 12], and a reference book on Android application security [29].

Several authors have performed pre-installation static analysis of applications outside of the Android environment. Enck *et al.* [14] decompile 1,100 popular Android applications from the Android Market and analyze their security.

[19]Google Play Store version 3.5.16 on Android 4.0.4, but this behaviour is also reproducible on earlier versions of Android.

Felt *et al.* [17] build a tool to decompile applications and identify over- and under-declaration of permissions. Barrera *et al.* [3] explore the permissions of popular Android applications and find no strong correlation between application categories and requested permissions. Chia *et al.* [7] analyze the correlation between application ratings, reviews and permissions requested by both new and popular apps. Jiang *et al.* [31] analyze approximately 200,000 Android apps to identify instances of known and previously unknown malware.

Relatively little work has analyzed the installation process on Android, other than a large focus on permissions. Enck *et al.* [15] present an enhancement to the Android installer that can specify dangerous permission combinations in policy files. Apps that request these permissions can be denied installation. Similarly, Hornyack *et al.* [24] describe a modified Android app installer that allows the user to reduce privileges of new apps by selectively disabling individual permissions, and limiting access to private information. Grace *et al.* [23] briefly describe "implicit capability leaks" on stock Android builds through UID sharing. In contrast to our analysis, these leaks are attributed to poor developer practices, rather than to limitations of the UID sharing model itself. Van Oorschot and Wurster [30] provide a brief analysis of Android code signing practices, comparing them to the properties of their key-locking tool.

An app, once installed, can attempt to escalate its privileges by either colluding with other installed apps, or abusing unprotected interfaces on installed privileged apps (known as the confused deputy problem). Ontang *et al.* [26] propose a flexible policy language to define restrictions that developers can apply on inbound IPC requests. To minimize the impact of confused deputy attacks, Felt *et al.* [19] propose IPC inspection which reduces the privileges of a calling application when it interacts with a higher privileged application. Dietz *et al.* [9] use a form of call stack analysis to track provenance on sensitive information, and prevent this information from reaching unintended applications. Enck *et al.* [13] also focus on sensitive information leakage by exploring the use of taint analysis on Android. Malicious application collusion is addressed by Bugiel *et al.* [6] who present a broad security framework which establishes semantic links between application calls, uses a reference monitor, and provides a kernel-level mandatory access control mechanism.

# 7. CONCLUDING REMARKS

In this paper, we have used a large collection of Android application data to infer security-relevant relationships between apps. Most of these relationships were non-apparent prior to our exploration. Using this dataset, we conducted a detailed analysis, the first of its kind, of the Android app installation process. We plan to make this dataset available for download (with a web-interface we call Android Observatory), which will enable other researchers to replicate our results and conduct investigations beyond the scope of this paper.

We observed that test keys are widely used to sign applications. Clearly, markets should refuse the inclusion of applications signed with these keys, as they pose a threat to software updates. It also seems clear that inconsistencies in the display of permissions during app installation involving shared UIDs are important to address. Finally, we have have made a case that the Android development community would benefit if UID sharing and signature permissions were allowed between apps signed with different keys. Our suggested UID mechanism could be incrementally deployed to provide this flexibility.

*Acknowledgements.*

We thank the F-Droid maintainers for additional information on their code signing policy. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)—the first author through a Canada Graduate Scholarship; the second through a Postdoctoral Fellowship; and the fourth through a Discovery Grant and as Canada Research Chair in Authentication and Computer Security. We also acknowledge support from NSERC ISSNet.

# 8. REFERENCES

[1] Anderson, R., Bergadano, F., Crispo, B., Lee, J.-H., Manifavas, C., and Needham, R. A new family of authentication protocols. *ACM SIGOPS Operating Systems Review 32*, 4 (1998).

[2] Arkko, J., and Nikander, P. Weak authentication: How to authenticate unknown principals without trusted parties. In *Security Protocols* (2002).

[3] Barrera, D., Kayacik, G., van Oorschot, P., and Somayaji, A. A Methodology for Empirical Analysis of Permission-based Security Models and its Application to Android. In *CCS* (2010).

[4] Barrera, D., and van Oorschot, P. Secure software installation on smartphones. *IEEE S&P Magazine 9*, 3 (2011).

[5] Becher, M., Freiling, F. C., Hoffmann, J., Holz, T., Uellenbeck, S., and Wolf, C. Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices. In *IEEE S&P Symposium* (2011).

[6] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A., and Shastry, B. Towards taming privilege-escalation attacks on Android. In *NDSS* (2012).

[7] Chia, P. H., Yamamoto, Y., and Asokan, N. Is this app safe? A large scale study on application permissions and risk signals. In *WWW* (2012).

[8] Damgard, I., and Koprowski, M. Practical threshold RSA signatures without a trusted dealer. In *EUROCRYPT* (2001).

[9] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security* (2011).

[10] Eckersley, P., and Burns, J. Is the SSLiverse a safe place? In *Chaos Communication Congress* (2010).

[11] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: detecting privacy leaks in iOS applications. In *NDSS* (2011).

[12] Enck, W. Defending users against smartphone apps: Techniques and future directions. In *ICISS* (2011).

[13] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).

[14] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of Android application security. In *USENIX Security* (2011).

[15] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *CCS* (2009).

[16] Enck, W., Ongtang, M., and McDaniel, P. Understanding Android security. *IEEE S&P Magazine* (Jan/Feb 2009).

[17] Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *CCS* (2011).

[18] Felt, A., Finifter, M., Chin, E., Hanna, S., and Wagner, D. A survey of mobile malware in the wild. In *SPSM* (2011).

[19] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *USENIX Security* (2011).

[20] Gamma, E., Helm, R., Johnson, R., and Vlisides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] Geer Jr., D. E., and Yung, M. Split-and-delegate: Threshold cryptography for the masses. In *FC* (2002).

[22] Goldberg, I., Mashatan, A., and Stinson, D. On message recognition protocols: Recoverability and explicit confirmation. *International Journal of Applied Cryptography 2*, 2 (2010).

[23] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS* (2012).

[24] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *CCS* (2011).

[25] Lucks, S., Zenner, E., Weimerskirch, A., and Westhoff, D. Concrete security for entity recognition: The Jane Doe protocol. In *INDOCRYPT* (2008).

[26] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in android. In *ACSAC* (2009).

[27] Samuel, J., Mathewson, N., Cappos, J., and Dingledine, R. Survivable key compromise in software update systems. In *CCS* (2010).

[28] Shoup, V. Practical threshold signatures. In *EUROCRYPT* (2000).

[29] Six, J. *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. O'Reilly Media, 2011.

[30] van Oorschot, P., and Wurster, G. Reducing unauthorized modification of digital objects. *IEEE Transactions on Software Engineering 38*, 1 (2012).

[31] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS* (2012).

# APPENDIX

# A. DATASET DETAILS

In order to build a dataset representative of the types of applications available to end-users, we acquired applications from several sources. We describe the sources from which we acquired applications and the rationale for selecting the source. We detail the process by which application information was imported into our dataset. Finally, we describe the web-interface we built to explore the dataset called the Android Observatory.

## A.1 Application Sources

### *Google Play Store.*

The Google Play Store[20] is a factory-installed app on most Android devices. Most applications in our dataset were obtained by crawling the top free applications in the Google Play Store for over a year beginning in early 2010 (then known as the Android Market). In total, our dataset contains 6,079 app packages from the Google Play Store, which is the standard location from which the majority of Android users currently acquire third party applications.

### *Alternative Markets.*

Several markets have emerged to offer alternative application delivery channels. These markets differ from the Android Market by their terms of service, pricing and developer payment schemes, restrictions on content, and user experience. For our data we examined applications from four alternative markets.

- The *Amazon Appstore*[21] is a mainstream alternative to Google's Play Store. It is featured prominently on Amazon-branded devices such as the Kindle Fire. One paid application per day is featured by Amazon and offered for free download. This provided a low-cost (free) means of including paid applications in our dataset.
- The *F-Droid market*[22] offers only free and open source Android apps. All apps are GPL and Apache licensed and built from source by the F-Droid maintainers. The authors of the applications themselves are not responsible for signing the F-Droid app package, rather they are signed by the market maintainers.
- The *MiKandi market*[23] offers applications with adult content. We included the top 100 free applications from MiKandi. Adult content is prohibited on the Android market so the inclusion of these apps expands authorship in the Observatory's dataset.
- The *Aproov market*[24] is an alternative market that provides more detailed app categorization, greater profit shares for developers, and a more sophisticated app rating system than Google Play. The top 100 applications from each Aproov category were included as a large non-Google Play data source.

### *Contagio Malware Dump.*

The Contagio website[25] hosts known desktop and mobile malware samples for research purposes. Some of the samples

[20] https://play.google.com/store/apps

[21] http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011

[22] http://f-droid.org/

[23] http://www.mikandi.com/

[24] http://www.aproov.com/

[25] http://contagiominidump.blogspot.com/

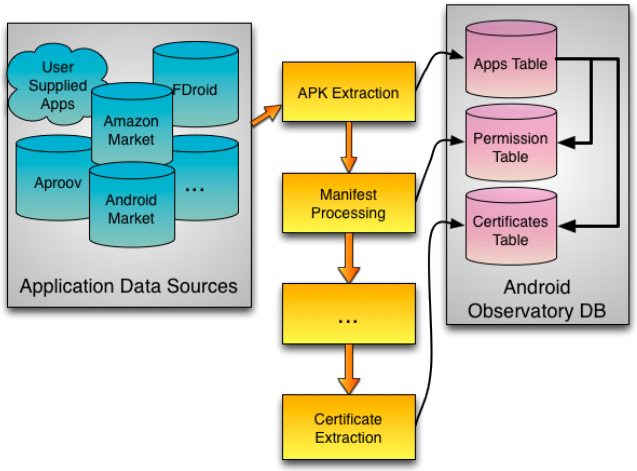



**Figure 5: Block diagram of application import process.**

were once available on the Android or alternative markets. We downloaded 209 malicious Android apps from Contagio. Some malware samples obtained from Contagio have been independently analyzed in the literature [18, 31].

*File-sharing.*

We obtained 4 popular app collections that are available on the Bittorrent network. File-sharing networks are a common source of pirated content and are commonly considered to present a risk of malware. We note that our dataset does not contain the app executables themselves, only meta-data extracted from the application package.

## A.2 Building Our Dataset

The database is populated with information extracted from Android packages. A chain-of-responsibility pipeline [20] allows the import and processing of applications from a variety of sources. Each component of the chain is able to process application information, insert database records, and prepare the application for subsequent components. The modular nature of the pipeline allows for new analysis operations (*e.g.,* static analysis tools) to be integrated in order to augment the database with additional information.

The pipeline extracts two main types of data from each Android app package (see Figure 5): (1) metadata extracted from the package such as app names, package names, version numbers, permissions, and certificate attributes from file contents; and (2) a structural description of the package such as the individual hashes of resources, compiled code, and the application manifest. Distinct variants of the same app (*e.g.,* different versions or same version from difference sources) are indexed separately as unique apks. Relevant pieces of the extracted information are stored in separate database tables for application packages, certificates, and permissions. By using the relations constructed between the tables, it is possible to locate common elements between applications.

In order to obtain attributes from certificates for inclusion in the dataset, we modified Oracle's Java `keytool` utility[26] to extract more detailed information than the base tool provided. We extracted the certificate serial number, public key and public parameters (modulus and exponent for DSA and RSA).

## A.3 The Android Observatory

The ability to query a diverse set of Android applications provides insight into developer behaviour and assists in hypothesis validation. A large empirical dataset also helps researchers to determine the feasibility of deployment of a new technical proposal. The Android Observatory (which we intend to make available at: `androidobservatory.org`) is our first attempt at exploring relationships between apps and their components.

Each field of our dataset is searcheable via the Observatory web interface. Applications can be found by package name, application name, permissions requested, version, or shared user ID strings. Queries may also be posed in terms of SHA1 hashes of the full binary or of its sub-contents, such as the compiled dex code, the resource bundle, or the manifest. Certificates may be searched by fingerprint, X.509 attributes, serial, and key parameters. We constructed a set of SQL queries to obtain the information displayed in the Observatory, however we don't allow for arbitrary queries to be executed. Instead we plan to make available a full copy of the database for custom queries.

It is likely that most end-users will not know how to calculate cryptographic hashes of Android apps or extract X.509 certificate details. With this in mind, the Observatory allows users to upload their own APKs. Uploading APK files is particularly useful for helping users decide whether a downloaded application (perhaps obtained from an untrustworthy source) can be trusted. In the case of an APK upload, the system will extract the app package, process and insert it into the Observatory database, and display a new page for it listing its properties and related applications. The Observatory displays an alert to users if a queried application shares a certificate with an application with known malware.

For each application, links are provided to applications with the same (or similar) attributes: application name, package name, signature, permissions, and identical dex files, manifest, or resource bundles. A direct link to the certificate information is provided from the detailed view of each application. The source of each application (*i.e.,* Android Market, Amazon Marketplace, *etc.*) is displayed in order to aid in the identification of either identical or related applications acquired from several sources. In cases where the application uses a shared UID the effective permissions of the application are shown by taking the union of the permissions of all other applications with a matching shared UID string and signing key.

Moving forward, we see the Android Observatory continuing to be useful to researchers and the Android community. We believe it would be useful to grow the dataset used herein through the inclusion of new and emerging marketplaces, as well as increased coverage of our existing sources. In addition, making the web-interface available will allow Android users to obtain information about apps they have acquired or are interested in installing, as well as helping them understand links between multiple apps.

[26] `http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html`