TRUSTORE: IMPLEMENTING A TRUSTED STORE FOR ANDROID

Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo

May 2014

Technical Report # DISI-14-010
Version 1.0

Published also: Y. Zhauniarovich, O. Gadyatskaya, B. Crispo "Enabling trusted stores for Android" in ACM CCS 2013

TruStore: Implementing a Trusted Store for Android

Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo

University of Trento, Trento, 38100, Italy
`zhauniarovich, gadyatskaya, crispo@disi.unitn.it`

Abstract. In the Android ecosystem, the process of verifying the integrity of downloaded apps is left to the user. Different from other systems, e.g., Apple App Store, Google does not provide any certified vetting process for the Android apps. This choice has a lot of advantages but it is also the open door to possible attacks as the recent one shown by Bluebox [35]. To address this issue, we present how to enable the deployment of application certification service, we called TruStores, for the Android platform. In our approach, the TruStore client enabled on the end-user device ensures that only the applications, which have been certified by the TruStore server, are installed on the user smartphone. We envisage trusted markets (TruStore servers, which can be, e.g., corporate application markets) that guarantee security by enabling an application vetting process. The TruStore infrastructure maintains the open nature of the Android ecosystem and requires minor modifications to Android stack. Moreover, it is backward-compatible and transparent for developers, and does not change the application management process on a device. In the paper we present the TruStore architecture and report the implementation details of the client part.

1 Introduction

During the last several years we observed an unprecedented growth of the Android ecosystem. Google tried to make the Android platform as open as possible: anybody can develop applications (or apps, for short) for it. Moreover, it does not tie developers to a specific application market, providing them a possibility to freely publish their apps on third-party stores. This all leads to the sustainable growth of the number of third-party applications; moreover, there exist a number of third-party markets such as Amazon, Yandex and so on.

At the same time this popularity of the Android platform also attracts adversaries. E.g., Kaspersky Lab recently reported on 20.000 new Android malware samples detected in the beginning of 2013 [28]. Even if a user shops for apps only on the official Google Play market, she cannot be totally sure in malware absence on her device, as reported by security companies[1], as well as by security researchers [46]. Secondly, there is also a lot of examples of apps, which

[1] `http://countermeasures.trendmicro.eu/android-malware-believe-the-hype/`

are not classified as malware, but at the same time they collect a lot of private information about the users [14], or are simply of poor quality.

One of the recent attacks on Android, which is discovered by BlueBox, demonstrates that an attacker might be able to inject malicious code into a legitimate application without damaging the digital signature of the original developer [35]. This type of attacks is quite dangerous considering the open nature of the Android platform, which allows a user to download applications not only from the official Google Play market, but virtually from everywhere. Thus, users shopping at a third-party market cannot be reassured by just checking that the desired app comes from a trusted developer and has good reviews. And while the biggest phone manufactures claim to have fixed this vulnerability in their currently supported phones, minor providers might simply ignore it. Moreover, unsupported devices of major players are still the target for this attack. Anyway, the issue of repackaging Android apps is more general and still open [42, 44].

Security companies have started to propose various solutions for securing mobile platforms: anti-virus applications, approval by certificate authorities [38], and reputation services integrated with markets [39]. Yet, these are not enough. Anti-virus software for smartphones in its current format is very limited (not only it drains the battery faster, but it will detect only the threats recognized by a security company, while ignoring, e.g., privacy leaks or application collisions); certificate authorities are known to be susceptible to failures [32]; and reputation services by nature cannot react quickly to new threats. Especially in the context of an enterprise that accepts the so-called Bring-Your-Own-Device (BYOD) paradigm these solutions are of limited applicability, as they do not allow a fine-grained control over security of apps installed on the platform.

Alternative solutions available could be summarized as follows: (a) app rewriting (e.g., [24,40]), (b) off-device app verification either by the user or on a market (for instance, [9,13,19]), (c) platform hardening (see for example [8,10,34,37,43]); or (d) a combination of those. However, they are not fully satisfactory for a variety of reasons. App rewriting is not acceptable from the legal perspective, as the rewriters (e.g. an enterprise information security staff or an end-user herself) are not the digital rights holders, and, therefore, rewriting is as legal as repackaging (a known source of app plagiarism and a malware distribution vector [18, 45]). Off-device verification and certification are quite efficient if done by a market. This approach works for the Apple's walled garden ecosystem, but for Android apps this step is essentially missing. Google claims to check the apps submitted to Google Play, but the community has reported severe limitations of their app validation process, see, e.g., [29]. In the same time, as a consequence of the Android ecosystem openness, even if a third-party market will enable app verification process, there is no assurance that the installed app is the one that was checked on the market, i.e., there is no difference for a user between checked and unckecked applications.

We aim to close this gap of lack of trust on a particular app market by enabling TruStore: a concept of a trusted market for Android that can guarantee security of apps provisioned to end-users' devices. The idea of the TruStore implementation is based on the Android application signing process described in Section 2. TruStore adds an additional market signature to the vetted applications before provisioning. Thus, we do not break the integrity of the signed package but add an additional feature that helps to ensure that the application has been checked. Then the end-user's device is responsible for checking whether the loaded application is signed by a trusted market signature. The demo of our system is presented in [41].

In this report we show an architecture of a trusted market and describe a proof-of-concept implementation for a regular Android device. We discuss how a trusted market may affect the most significant steps in the app lifecycle: installation, update, deletion and interactions with other apps and with the device infrastructure, and overview the potential of a trusted market for end-users, developers and enterprises with the BYOD policy enabled.

The rest of this paper is structured as follows. Section 2 describes the purposes of code signing and why and how it is used in Android. In Section 3 we overview TruStore, while in Section 4 we provide the implementation details. Section 5 discusses the implications of TruStore on application management process and on the Android platform as a whole. Section 6 reports on application signing and distribution approaches of other mobile platforms, while in Section 7 we discuss the existing approaches of securing mobile platforms and how TruStore fits them. We conclude with Section 8.

2 Application Signing Internals

Android applications are spread across the devices in the form of Android *Application Package Files* (`apk` files). As programs for this platform are mainly written in Java, not surprisingly this format has a lot in common with the Java packaging format – `jar` (Java ARchive), which is used to combine code, resource and metadata (from an optional `META-INF` directory) files into one file using the `zip` archiving algorithm. The META-INF directory stores package and extension configuration data, including security, versioning, extension and services [4]. Basically, in the case of Android the *apkbuilder* tool zips together built project files [1] and then this archive is signed with the standard Java utility *jarsigner* [5]. During the application signing process `jarsigner` creates the `META-INF` directory that usually contains the following files in case of Android: *manifest file* (`MANIFEST.MF`), *signature files* (with `.SF` extension) and *Signature Block Files* (`.RSA` or `.DSA`).

The *manifest file* (`MANIFEST.MF`) consists of the main attributes section and per-entry attributes, one entry for each file contained in the unsigned apk. These per-entry attributes store information about the file name and a digest of the file contents encoded using the `base64` format. On Android, the `SHA1` algorithm is used to compute the digest. An excerpt from a manifest file is presented in Protocol 1.

The content of the *Signature File* (`.SF`), which contains data to be singed, is similar to the one of `MANIFEST.MF`. An example of this file is presented in Proto-

Protocol 1 An excerpt from a manifest file.

```
Manifest-Version: 1.0
Created-By: 1.6.0_41 (Sun Microsystems Inc.)

Name: res/layout/main.xml
SHA1-Digest: NJ1YLN3mBEKTPibVXbFO8eRCAr8=

Name: AndroidManifest.xml
SHA1-Digest: wBoSXxhOQ2LR/pJY7Bczu1sWLy4=
```

col 2. Main section contains a digest (`SHA1-Digest-Manifest-Main-Attributes`) of the main attributes and a digest (`SHA1-Digest-Manifest`) of the content of the manifest file. Per-entry section contains digests of entries in the manifest file with the corresponding file names.

Protocol 2 An excerpt from a signature file.

```
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: nl/DtR972nRpjey6ocvNKvmjvw8=
Created-By: 1.6.0_41 (Sun Microsystems Inc.)
SHA1-Digest-Manifest: Ej5guqx3DYaOLOm3Kh89ddgEJW4=

Name: res/layout/main.xml
SHA1-Digest: Z871jZHrhRKHDaGf2K4p4fKgztk=

Name: AndroidManifest.xml
SHA1-Digest: hQtlGk+tKFLSXufjNaTwd9qd4Cw=
...
```

The last part in the chain is the *Signature Block File* (`.DSA` or `.RSA`). This binary file contains a signed version of the signature file; it has the same name as the corresponding `.SF` file. Depending on the used algorithm (RSA or DSA) it has different extensions.

It is possible to sign the same apk file with several different certificates. In this case in the `META-INF` directory there will be several `.SF` and `.DSA` or `.RSA` files (their number will be equal to the number of times the application was signed). This multisigning feature is the cornerstone of the trusted store concept.

Signing Usage on Android Mobile code signing is used to assure strong authentication and integrity. After the code is downloaded the user wants to be sure that it was not modified and comes from a particular developer (actually, from the entity who has singed the code with a corresponding private key). Still, the user cannot be sure if she can trust the developer. Usually, this problem is solved by finding a common root of trust, which builds the trust relationship between the user's platform and the developer's certificate used to sign the code (e.g., on iOS this root of trust is implemented as the Apple market signature). If this root is not found, the developer's certificate cannot be trusted and thus, the code signed with it cannot be trusted either. Although there is a lot of different certificate authorities, it quite difficult to find a root which is trusted by all platforms.

In particular, on Android an app with a trusted certificate does not mean a trusted app. For example, Symantec promotes its trusted certificate services to be used to sign apps on Android (they already perform this trust certificates provisioning for Windows Phone) [38]. Yet, even a certificate from Symantec does not guarantee safety, as the code is not reviewed at all: it can still be malicious or buggy.

Most of Android apps are sealed with a developer-signed certificate (notice that for Android "certificate" and "signature" can be used interchangeably). This certificate is used for assurance that the code of the original application and its update come from the same place, and to establish trust relationships between applications of the same developer. To perform this check Android simply compares binary representations of certificates, which were used to sign an application and its update (in the first case) and collaborating applications (in the second).

This check of certificates is implemented in `PackageManagerService` by the method `int compareSignatures(Signature[] s1, Signature[] s2)`. In the previous section we noted that in Android it is possible to sign the same application with several different certificates. This explains why the method takes two arrays of signatures as parameters. Despite the fact that this method takes the central place in the Android security provision, its behaviour strongly depends on the version of the platform. In the newer versions (starting from Android 2.2) this method compares two arrays of `Signature`, and if both arrays are not equal to `null` returns `SIGNATURE_MATCH` value if all `s2` signatures are contained in `s1`, and `SIGNATURE_NO_MATCH` otherwise. Before the version 2.2, this method checked if array `s1` is contained in `s2`. That behaviour allowed the system to install upgrades even if they had been signed only with a subset of certificates of the original application [3].

Today there are no benefits of signing an application with multiple certificates and this functionality can be easily suppressed. However, we believe that for compatibility reasons Google will not change it in the near future. We use this functionality to develop an architecture of TruStore.

3 TruStore Overview

Let us overview our approach, which we have called TruStore. TruStore (stands for Trusted Store) provides an infrastructure for app distribution and management on an Android smartphone that can be trusted by a user.

The TruStore architecture consists of two main parts: a *server* and a *client*. The server part, besides offering the standard app provisioning functionality provided by app markets, is responsible for the application vetting process.

A possible vetting workflow can be the following. A developer uploads his application to the server. Along with the app package TruStore could also require the source code to ease the vetting process, like it is done for the Apple AppStore market. The server then checks the application for compliance with certain standards of secure applications. This process can include static analysis of application executables and source code (if provided) and dynamic analysis. During this process the server can also provide a short report about the functionality the application uses. This report can be later requested by users if they would like to understand better the application functionality and its usage of sensitive device features.

If the app has passed the vetting process, the server signs the application with its private key and places it in its market to be accessible by users. It should be mentioned that it is possible to sign an Android package without violating the integrity of the original developer's signature. This can be done in case the source code was not provided by the developer for some reasons. Notice that the server part implementation (including the app provisioning and vetting functionalities) is out of scope of this chapter work. The interested reader can refer to, e.g., [9,13,19,46] for the examples of app verification frameworks.

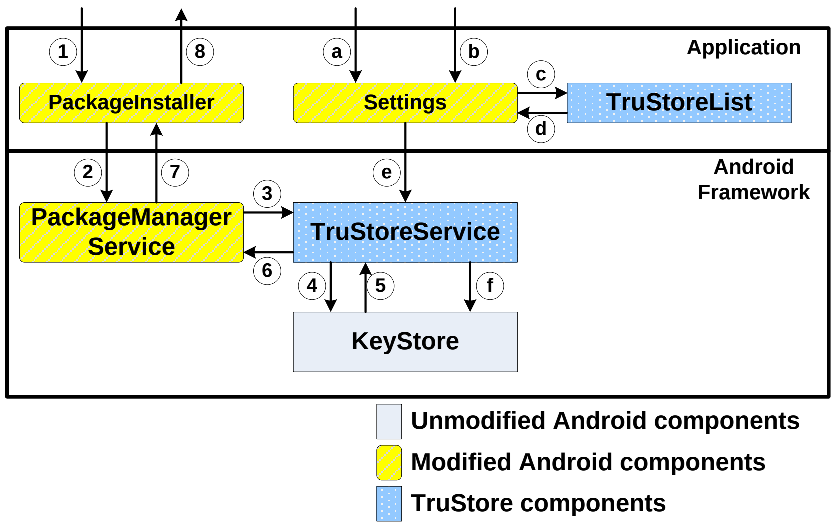


Fig. 1: The TruStore architecture: the steps with letters represent the TruStore management process; those ones with numbers describe the checks during app installation.

The client part is based on a modified Android system; it allows the device holder to make use of TruStore: the user with TruStore protection enabled can install only the packages signed with TruStore certificates. Figure 1 summarises the architecture of the client part of TruStore. The implementation intricacies of our system are presented in Section 4.

The TruStore management process starts with the activation of the TruStore protection. If a user wants to activate the TruStore protection she checks a special checkbox (Step *a* in Figure 1) added in the standard Android *Settings* application. After that she is able to select a special item (Step *b*) that will display the list of TruStore certificates installed in the system. On this screen the user may start the process of adding a new TruStore certificate to the system.

The `TruStoreList` application is responsible for displaying the list of certificates available to install on the external storage (Step *c*). The user can select a certificate and `TruStoreList` will pass the certificate back to the `Settings` application (Step *d*), which will store it in the system credential storage using `TruStoreService` (Steps *e* and *f*).

There are three main ways to install an application on Android:

- Using the *Google Play* application.
- Using the *PackageInstaller* application.
- Using the *adb* interface (`adb install` command).

These components interact with the `PackageManagerService` service responsible for package management in Android. The service functionality related to the installation of new packages is protected with a special permission `INSTALL_PACKAGES`, which permission level is `signatureOrSystem`. This means that only the applications that are placed on the system image or signed with the Android platform certificate can communicate with this service to install a new package. Therefore, if the TruStore server uses a dedicated market application for distributing purposes it cannot directly start app installation using `PackageManagerService`. However, any app using a special intent can call the standard `PackageInstaller` application that can invoke the installation of a package and report about the installation process to the user.

The application installation process with the activated TruStore protection is presented in Figure 1. The user starts the installation using our modified version of `PackageInstaller` and performs the usual sequence of app installation steps (Step 1). The installer notifies `PackageManagerService` that it has to begin the installation of the package into the system. From this point `PackageManagerService` performs the job, while `PackageInstaller` waits for the installation report. In one of the checks of `PackageManagerService` we added a hook that communicates with `TruStoreService` and passes to it the list of certificates extracted from the installed application (Step 3). `TruStoreService` checks if at least one certificate in the obtained list matches the TruStore certificate installed in the system (Step 4). If a match is found then `PackageManagerService` finishes the installation and notifies `PackageInstaller` about the success, otherwise it generates a special error that is displayed to the user by the installer application (Steps 7, 8).

4 TruStore Implementation Details

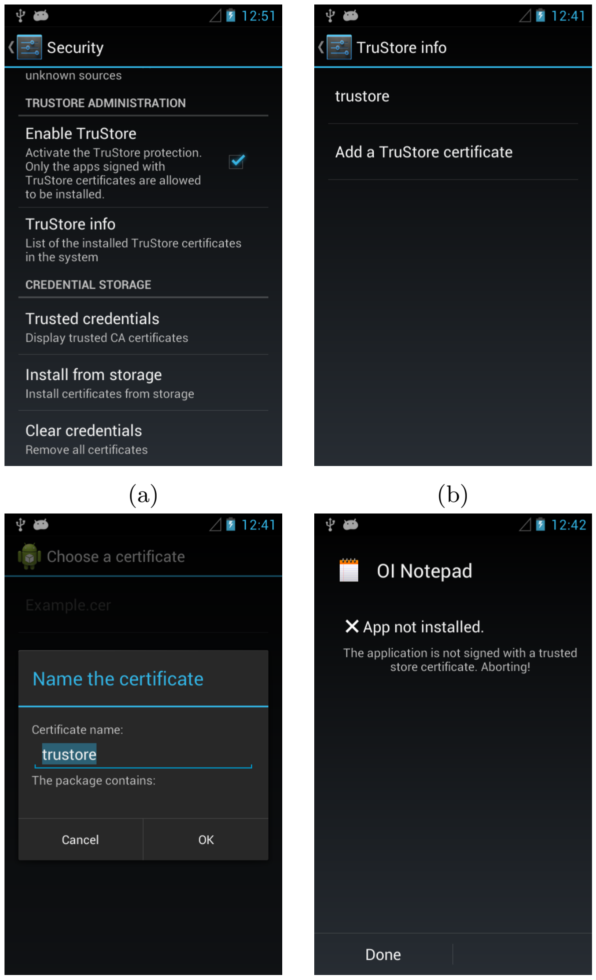
In this section we describe the implementation details of the TruStore architecture considered so far. TruStore is developed on top of Android Open Source Project (AOSP) [2]. The proof of concept has been implemented for the Google Nexus S phone with 4.1.2_r2 version of AOSP. The implementation of TruStore touches two levels of the Android software stack: the *Application* and the *Android Framework* levels; at both of these levels our proof-of-concept implementation modifies the standard Android components, as well as adds new parts. During TruStore implementation we followed the objective to make as less intrusive modification of the platform as possible, using standard components where possible. The proposed modifications will not change the process of Android application development, but will require changes in the Android system code, which can be easily incorporated by Google in the future releases of Android.

The process of the TruStore management begins from the modified `Settings` application. In this application we added two preferences that activate the TruStore protection and allow the user to see the list of installed trusted store certificates. The setting is written into `Settings.Secure` content provider and is later used by different components to detect if the TruStore protection is enabled. The screenshot of the modified `Settings` application with these added preferences is presented in Figure 2a, while in Figure 2b the list of currently installed trusted store certificates is shown. From this component it is possible to start the process of adding a new certificate.

As *Settings* application shares its UID with the system server and it is prohibited to read the content of the external storage from the system process, the functionality to search for and demonstrate the available TruStore certificates is extracted to an additionally implemented application *TruStoreList*, which is launched from the *Settings* app using an explicit intent. When the user selects new certificate she can provide additional information with it. Currently *TruStoreList* only provides the functionality to enter the name of the certificate, but other fields can be added to the dialog (shown in Figure 2c) with ease. For instance, along with a certificate it may require to save credentials, which can be used later to prevent unauthorised modifications of the stored trusted certificate. *TruStoreList* passes this information (the certificate and additional data) back to the *Settings* application, which in turn communicates with `TruStoreService` to store them.

`TruStoreService` (implemented at the *Android Framework* level) is responsible for storing and extraction of certificates and data related to them. It is also used to compare the list of obtained certificates with the stored ones (the positive result is returned if even one match is found). We decided to implement this functionality as a separate service in order to be able to add some additional features in the future. For instance, an external app, e.g., a corporate BYOD profile management app, may in the future be responsible for managing TruStore certificates having the interface provided by `TruStoreService`.

To preserve securely a certificate with additional data, the system *KeyStore* component is used. This is a standard way to store credentials on Android. This



12:51

Security

unknown sources

TRUSTORE ADMINISTRATION

Enable TruStore
Activate the TruStore protection. Only the apps signed with TruStore certificates are allowed to be installed. ✓

TruStore info
List of the installed TruStore certificates in the system

CREDENTIAL STORAGE

Trusted credentials
Display trusted CA certificates

Install from storage
Install certificates from storage

Clear credentials
Remove all certificates

(a)

12:41

TruStore info

trustore

Add a TruStore certificate

(b)

12:41

Choose a certificate

Example.cer

Name the certificate

Certificate name:
trustore

The package contains:

Cancel | OK

(c)

12:42

OI Notepad

✕ App not installed.

The application is not signed with a trusted store certificate. Aborting!

Done

(d)

Fig. 2: Screenshots of TruStore: (a) Settings to enable TruStore, (b) The certificate list of trusted stores, (c) TruStoreList application, (d) PackageInstaller error when a package is not signed by TruStore certificate

component automatically encrypts the stored information and grants access to it (based on UID) only to the component that originally initiated the preservation of the data. As `TruStoreService` is a part of the system server, only the Android components with the system UID equal to 1000 can read and modify these data. To distinguish the TruStore information from other credentials stored in the credential storage we add a special prefix (TRUSTORE). Thus, `TruStoreService` selects from the storage only the appropriate data. Additionally, `TruStoreService` may act as a cache for stored certificates.

To invoke a TruStore check we embedded a hook into the method `installPackageLI`. This hooks extracts the certificates of the installed package and compares them against the list of trusted store certificates using `TruStoreService`. If a match is found the installation process proceeds, otherwise `PackageManagerService` finishes the installation with a special TruStore error. The hook is embedded in the place when Android `PackageManagerService` has finished all verification steps. In this way, we are sure that the private keys corresponding to the certificates extracted from the package to be installed have been used to sign the package (thus, verifying the integrity of the signatures).

The last piece of the puzzle, `PackageInstaller` was modified to correctly display the explanation of the TruStore certification match error. An example of such fail is shown in Figure 2d. All other functionality of this component is left untouched. However, the correct handling of this error via, e.g., Google Play, cannot be guaranteed in our proof-of-concept implementation: as the Google Play application is not a part of AOSP, we cannot modify its sources. Yet, if the certificate check has failed, the user will still receive an error and the installation will fail.

To the best of our knowledge, the existing third-party Android application markets (e.g., Amazon) do not sign the deployed applications; therefore they do not take the responsibility for security of the deployed applications, while the TruStore does so.

5 Android Application Management with TruStore

As we mentioned before, the TruStore modifications have changed only the process of application installation. At the same time, due to the Android platform security architecture these changes also influence other aspects of application management. In this section, we consider how the process of application management has been changed with the TruStore modifications.

There are three main points in the lifecycle of each application: install, update and delete. The process of application deletion from the user or the developer point of view is not changed by TruStore, so it is not considered in this section. The installation process of an application is considered in details in the previous sections; it is not modified from the developer's perspective, but changes from the user perspective (installation may fail now).

Here it is worth mentioning that Android will prohibit the installation of the same application from different trusted stores, because these two applications have the same package name, but are signed with different sets of signatures. The same situation is for updates: an application cannot be updated by a store that has not installed it. A potential problem can arise if application updates in one market are vetted more quickly than in another. Thus, the user may want to install an update from another market, but will not be allowed to do this by the Android system.

At the same time, the TruStore approach does not reduce the app update capabilities of the developer, as the scheme of app distribution via a marketplace already assumes that the updates can be executed only through the market. In our approach the developer has to submit a new version of his app to the TruStore server, where it will be analysed, signed and distributed to users. The "kill switch" functionality available, e.g., on the Apple market and Google Play, which has proved very useful, can also be supported via TruStore.

The TruStore modifications influence the interactions with app components protected with `signature` and `signatureOrSystem` type of permissions, and the `sharedUserId` functionality for app interactions. The TruStore modifications will prohibit such kind of interactions between applications that are installed from different trusted markets, although they have been implemented by the same developer. This limitation comes from the fact that the applications installed from different markets will have different set of signatures (although developer signature may be the same). However, this restriction enables more security: TruStore will ensure safety of the apps loaded via itself, while safety of apps on other third-party markets cannot be guaranteed, and the interactions between trusted and untrusted apps may lead to information leaks and privilege escalation attacks.

6 App Signing and Distribution on Other Mobile Platforms

In this section we try to observe other mobile platforms and their approaches of security provisioning of app distribution. In particular, we are interested how third-parties are vetted and signed.

Apple iOS. In its standard configuration Apple permits the run of third-party applications only cryptographically signed by Apple. Thus, all third-party applications should be from Apple App Store. However, sometimes it is required to run applications on real devices during the development and testing. Moreover, companies may also want to run their apps on employees' devices without being published in App Store. To support such types of application distribution Apple admitted provisioning profiles. Provisioning profile is a special plist file signed by Apple that accompanies an application. Being installed on a device the provisioning profile specifies additional signatures, which may be used to sign the app, that allows this application to be run on the device. Basically, each provisioning profile specifies who (specified by DeveloperCertificate) can run an application (specified by AppId) on what devices (specified by UDIDs of the devices). So as provisioning profiles are signed by Apple, the company can control who and on what devices may distribute applications.

When a developer submits an application to Apple App Store she signs the package with her signature. After this app is approved Apple resigns this package with its own signature, thus, it can be run on all devices.

Every application is vetted before it appears in Apple App Store. Although Apple claims that this procedure protects the users from access to malicious applications there are examples showing the opposite [11, 21, 31]. Moreover, the inability to install applications from other sources forces the users to jailbreak their iPhones. The jailbroken phones provide an additional attack vector [12, 22, 26]. The process of application vetting is not divulged. At the same time, this is very interesting topic and it is not surprisingly that some details about this process are uncovered by the researches in this field [20, 36] and by Apple during the lawsuit with Google [30]. According to the article [36], each application is reviewed automatically using static analysis and dynamic analysis. Also it seems that each application is vetted for the use of private APIs. Moreover, according to [30] forty reviewers vet all submitted applications and each app is reviewed by two experts. It is calculated [30] that an expert spends about 6 minutes to vet an application. So, we can conclude that during this time it is impossible to detect all possible flaws in an application and, that is why, a number of applications that violate the "iPhone SDK Agreement" [23] is present at the App Market [11,21,31]. Shock results are also described in the article [27]. According to this article the researches from the Technical University of Vienna have discovered that more than a half of explored third-party applications in Apple App Store collect and distribute unique telephone code that can be used to track the user of the phone. Also the researchers have mentioned that apps from App Store more often access the user's data than apps from the unpoliced Cydia repository for jailbroken iPhones.

Thus, comparing with TruStore the idea exploited by Apple is almost the same – only vetted applications are allowed to be installed on a device. At the same time, TruStore's approach is more flexible because it permits a user to select which market services she trusts, while in the case of Apple, the only trusted market is Apple App Store. Given the open ecosystem (the applications can be installed from any third-party market) this approach is unacceptable in case of Android. Similarly, the distribution of enterprise applications also requires the approval from Apple.

Windows Phone. The latest Windows Phone platform version requires only the marketplace signature. The standard application deployment requires the developer to submit it to the Windows Phone Store run by Microsoft, where the code will be checked and signed.

As an exception to this process, organizations may deploy corporate apps via a dedicated Company Hub app. A company that desires to deploy its apps has to register with Microsoft and acquire a corporate certificate from Symantec. After that all apps signed with the corporate certificate can be deployed on devices with the Company Hub app enabled.

Considering Windows Phone's approach, it can be mentioned that it has a lot of common with Apple's one. Thus, this approach also cannot be used in case of Android given the open ecosystem it provides.

The Firefox OS. The Firefox OS is a new mobile platform released by Mozilla; it aims at HTML5 applications. Basically, web apps can be distributed through any website. In this case, they consist only of their manifest and act like web pages rendered in a browser. Thus, they have the same privileges as web pages run in a web browser. However, in some cases an application may also require access to the device features, like sensors indications or telephony stack. These features in Firefox OS are protected with permissions. Basic set of permissions are granted to *web* apps distributed through a website. At the same time, to be granted with other permissions an application must belong to *privileged* or *certified* types. *Certified* are packaged apps (i.e., applications that have all resources, like code and images, bundled) that are pre-installed on a device by a manufacturer. Not surprisingly, this type of applications have access to "dangerous" functionality provided by a phone, e.g., telephony access or app management. To become a *privileged* application, it must be reviewed and signed by the marketplace. A developer implements an application and sends it to the market for review. The marketplace perform a code review process, checks the requested permissions, and vets the code for the absence of malicious actions. After the application is vetted and signed, users can download it from the market and use its full power.

Although Firefox OS has a lot of common with the Android platform [17], the app distribution process is completely different. Basically, from all the app types supported by Firefox OS, only *privileged* applications are cryptographically signed. Additionally, in this case they are signed not by the developers but by the marketplace.

The approach proposed for this operating system is similar to TruStore's in the sense that vetted applications receive more privileges then unvetted ones. Alike Firefox OS, the Android OS with TruStore allows the installation of applications signed by trusted certificates.

Tizen. Tizen is another recent mobile platform. Tizen supports web, native and hybrid applications. The standard web app type on Tizen is a packaged web app. Only packaged apps can access the Tizen device API (the API to access the device capabilities such as telephony or messaging). Tizen also supports hosted web apps, which have an externally hosted document as their starting page.

Unlike other considered mobile systems like iOS or Firefox OS, the Tizen apps can be signed with two signatures: *developer* and *distributor* [17]. The developer signature determines the author of the package. Moreover, it is used to establish trust relationships between applications signed with the same developer certificate. The distributor signature may be applied by a device manufacturer or by the marketplace. This signature controls the maximum level of permissions available for the application. During the installation, this signature is compared with the system trusted certificates and based on this comparison the maximum privileged level is assigned to the application.

In such favour, the idea exploited in Tizen is very similar to the aim of TruStore. To our point of view, the Tizen approach can be considered as TruStore's final aim, when the package is signed with two certificates, one of which regulates the level of trust to the application. However, the implementation of such approach on Android is currently impossible without breaking its backward compatibility with already developed apps.

7 Related Work

As we have mentioned, the existing approaches on securing a mobile platform generally fall into 4 categories: app rewriting, off-device app verification, platform hardening or a combination of these techniques. We now overview these approaches and discuss how a TruStore implementation fits with each of those. For the lack of space we do not discuss in detail the full body of work available for mobile platform security. Rather, we give examples of each technique, while analysing how TruStore relates with each particular approach.

Application Rewriting Application rewriting is a technique of changing the original code of an app e.g. by replacing calls to a sensitive APIs (e.g., the photo stream access) by the calls to a security controller introduced on the platform [7, 24, 40], which would dispatch these calls based on the desired policy. In this way retrofitted apps installed on the platform will not be able to stealthily grab the sensitive data.

By nature app rewriting is not suitable for iOS and other tightly controlled ecosystems, as it invalidates the original market signature. On Android the rewriting mechanism has to replace the developer's signature with a new certificate and enable a control over which app is signed with which certificate, in order not to break the app interactions established through custom developer signatures [40]. In our approach the trust relationships among apps are maintained automatically, as we do not remove the original signature of the developer. As well, as opposed to app rewriting, TruStore does not repackage the original application, therefore it does not violate the rights of the developer.

Application Vetting Application vetting (or off-device app validation) can be used either on a market (as it is done, e.g., for iOS) or by the user/an information security staff of a company before loading the application on device. The existing approaches (e.g., [9, 13, 15, 19, 25]) aim at analysing the app code and retrieving security-relevant aspects of app behavior. For instance, Stowaway [9] analyses Android apps for vulnerable interaction patterns and PiOS [13] aims at detecting privacy violations, when an app illegally distributes sensitive user data. The app analysis can be static [9, 13] or dynamic [19, 25, 29].

The TruStore approach is by nature complimentary to application vetting. The server part of a trusted store is in fact expected to validate the app code submitted by developers.

Platform Hardening Improving the device security by introducing new components to the platform is currently the leading approach (if judged by the number of research papers). Notice that TruStore also falls in this category: we introduce new components of the Android middleware in order to implement the trusted certificates storage and execute the certificate checks.

There exist a lot of proposals of improvement of mobile platforms that tackle different security goals: for example, improvement of the permission system [16], enhancing the user control over her private data [10,14] or protecting applications from malicious interactions and banning the privilege escalation [8, 34]. The TruStore concept is limited in comparison with some of those techniques in the device context aspect. Namely, the frameworks for platform hardening at runtime are able to monitor the current situation on the device; they prevent the attacks because they have knowledge of each individual app and its actions. TruStore is not able to monitor the device security status, e.g., it is not able to prevent application collusions. Yet, TruStore achieves security by preventing loading of malicious apps.

Other Techniques The crowdsourcing technique for enhancing trust on mobile platform is an approach evangelized by all app markets and investigated by security researchers (see e.g. [6,39]). The user is encouraged to read app reviews and install only the apps that have received the community approval. However, the deficiency of this approach is lack of assurance for new or updated apps. For example, adversaries could publish an innocent game which would receive a high feedback, and then push an update with a malicious payload.

BYOD Techniques for the BYOD policies enforcement overlap with the approaches examined above, but their main goal is to allow an enterprise to secure its assets (corporate apps, frameworks, data or network), while allowing an employee to use her own device for interacting with these assets. TruStore can be used as a stepping stone for implementation of a BYOD solution in a company, when each application loaded onto a devices for company environment is validated and certified by TruStore. For instance, this scenario can be implemented for the MOSES system described in [33, 43].

8 Concluding Remarks

The openness of Android has provided possibilities for other players (e.g. Amazon) to run their own markets of applications. At the same time, the popularity of the platform also attracts adversaries. The last reports show that even the users of the official Google Play market cannot feel safe because of the lack of the strict vetting process there.

In this paper we introduced TruStore (Trusted Store) to enhance app trust. The main challenge for enabling a trusted store for Android is preservation of openness of the ecosystem. Our solution is intended to build a secure infrastructure for provisioning validated apps and their updates. We have designed and developed the client part of the TruStore architecture, showing that implemented modifications to the Android platform can enable this functionality. TruStore does not hinder the app management process for end-users and developers, but it can provide the app trustworthiness assurance for end-users and enterprises enabling the BYOD programs.

References

1. Android application: Building and Running. `http://developer.android.com/tools/building/index.html`.
2. Android Open Source Project (AOSP). `http://source.android.com/`.
3. Android Security Discussions: Multiple Certificates and Upgrade process. `https://groups.google.com/forum/?fromgroups#!topic/android-security-discuss/sY7Ormv3uWk`.
4. Jar File Specification. `http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html`.
5. jarsigner - JAR Signing and Verification Tool. `http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html`.
6. Yuvraj Agarwal and Malcolm Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *Proc. of MobySys'2013*, 2013.
7. Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard – enforcing user requirements on Android apps. In *Proc. of TACAS'2013*, pages 543–548, 2013.
8. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. of NDSS'2012*, 2012.
9. Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proc. of MobiSys'2011*, pages 239–252, 2011.
10. Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. CRêPE: A system for enforcing fine-grained context-related policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426–1438, 2012.
11. Amy Coopes. Australia warns of iPhone security risk. `http://www.google.com/hostednews/afp/article/ALeqM5gw3h9CaSr41wcnCsPda4CD5mqnyw?docId=CNG.a748b69f22077ddd5d23e00c220bc69a.381`, October 2010.
12. Dancho Danchev. iHacked: jailbroken iPhones compromised, five-dollar ransom demanded. `http://www.zdnet.com/blog/security/ihacked-jailbroken-iphones-compromised-5-ransom-demanded/4805`, November 2009.
13. Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proc. of NDSS'2011*, 2011.
14. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of USENIX OSDI'10*, pages 1–6. USENIX Association, 2010.
15. William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proc. of USENIX Security'2011*. USENIX Association, 2011.

16. E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing Androids permission system. In *Computer Security ESORICS'2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2012.
17. Olga Gadyatskaya, Fabio Massacci, and Yury Zhauniarovich. Security in the Firefox OS and Tizen Mobile Platforms. *IEEE Computer*, (Special Issue on Mobile Application Security), 2014. to appear.
18. Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: examining the landscape and impact of android application plagiarism. In *Proc. of MobiSys '13*, pages 431–444, 2013.
19. Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: automated security validation of mobile apps at app markets. In *Proc. of MCS'2011*, pages 21–26, 2011.
20. Shantanu Goel. Android vs iPhone: Security Models. `http://tech.shantanugoel.com/2010/06/26/android-vs-iphone-security-models.html`, June 2010.
21. Dan Goodin. Backdoor in top iPhone games stole user data, suit claims. `http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/`, November 2009.
22. Dan Goodin. World's first iPhone worm Rickrolls angry fanbois. `http://www.theregister.co.uk/2009/11/08/iphone_worm_rickrolls_users/`, November 2009.
23. Apple Inc. iPhone SDK Agreement. `http://www.wired.com/images_blogs/gadgetlab/files/iphone-sdk-agreement.pdf`, August 2009.
24. Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: fine-grained permissions in Android applications. In *Proc. of SPSM'12*, pages 3–14, 2012.
25. Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering iOS mobile applications. In *Proc. of WCRE'2012*, pages 177–186. IEEE Computer Society, 2012.
26. Jeremy Kirk. New iPhone malware steals data from jailbroken phones. `http://www.infoworld.com/d/security-central/new-iphone-malware-steals-data-jailbroken-phones-918`, November 2009.
27. Robert Lemos. Want to Track People? There's an App for That. `http://www.technologyreview.com/computing/27128/page1/`, January 2010.
28. Denis Maslennikov. IT threat evolution: Q1 2013 `http://www.securelist.com/en/analysis/204792292/IT_Threat_Evolution_Q1_2013`, May 2013.
29. John Oberheide. Dissecting the Android Bouncer, 2012. `http://jon.oberheide.org/files/summercon12-bouncer.pdf`.
30. Erica Ogg. Apple sheds light on App Store approval process. `http://news.cnet.com/8301-13579_3-10315328-37.html`, August 2009.
31. Bill Ray. iPhone app grabs your mobile number. `http://www.theregister.co.uk/2009/09/30/iphone_security/`, September 2009.
32. Steven B. Roosa and Stephen Schultze. Trust Darknet: Control and Compromise in the Internet's Certificate Authority Model. *Internet Computing, IEEE*, 17(3):18–25, 2013.
33. Giovanni Russello, Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. Demonstrating the effectiveness of MOSES for separation of execution modes. In *Proc. of CCS'12*, pages 998–1000, 2012.
34. Giovanni Russello, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. YAASE: Yet another Android security extension. In *Proc. of SocialCom/PASSAT*, pages 1033–1040, 2011.

35. The SecurityLedger. Exploit code released for android security hole `https://securityledger.com/2013/07/exploit-code-released-for-android-security-hole/`, Jul. 2013.
36. Nicolas Seriot. iphone privacy. In *Black Hat DC 2010*, Arlington, Virginia, USA, 2010.
37. Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *Proc. of NDSS'13*, 2013.
38. Symantec. Securing the mobile app market: How code signing can bolster security for mobile applications. White paper, 2012.
39. TrendMicro. Mobile app reputation service, 2011. `http://www.trendmicro.co.uk/media/ds/mobile-app-reputation-service-datasheet-en.pdf`.
40. Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for Android applications. In *Proc. of USENIX Security'12*, pages 27–27, 2012.
41. Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo. DEMO: Enabling Trusted Stores for Android. In *Proc. of CCS '13*, pages 1345–1348. ACM, 2013.
42. Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. FSquaDRA: Fast Detection of Repackaged Applications. In *Proc. of DBSec '14*, pages 131–146, 2014. to appear.
43. Yury Zhauniarovich, Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. MOSES: Supporting and Enforcing Security Profiles on Smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223, May 2014.
44. Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proc. of CODASPY '13*, pages 185–196, 2013.
45. Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of S&P '12*, pages 95–109, 2012.
46. Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of NDSS'2012*, 2012.