Baton: Key Agility for Android without a Centralized Certificate Infrastructure

David Barrera Daniel McCarney
Jeremy Clark Paul C. van Oorschot

Carleton University, Ottawa, ON, Canada
{dbarrera,dmccarney}@ccsl.carleton.ca
{clark,paulv}@scs.carleton.ca

ABSTRACT

Android's trust-on-first-use application signing model associates developers with a fixed signing key, but lacks a mechanism to transparently update the key or renew their signing certificate. As an advantage, this feature allows application updates to be recognized as authorized by a party with access to the original signing key. Changing keys or certificates requires that end-users manually uninstall/reinstall apps, losing all non-backed up user data. In this paper, we show that with appropriate OS support, developers can securely and without user intervention transfer signing authority to a new signing key. Our proposal, Baton, modifies Android's app installation framework enabling key agility while preserving backwards compatibility with current apps and current Android releases. Baton is designed to work consistently with current UID sharing and signature permission requirements. We discuss the technical changes made to Android, and remaining open issues such as key loss and signing authority revocation on Android.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Android, application signing, mobile operating systems

1. INTRODUCTION

Modern operating systems use digital signatures as a mechanism to verify the integrity of downloaded software and/or authenticate developers. Platforms such as iOS, Windows Phone, and Blackberry use code signatures to restrict installation of third-party applications to only registered developers. These platforms use a centralized authority, where software is signed by the vendor prior to being distributed to user devices. A centralized authority is restrictive for users (*e.g.,* users *must* obtain software and updates from sources formally sanctioned by the platform vendor) while allowing vendor control over certificate issuance.

Android, one of the most widely deployed mobile operating systems, does not use a centralized authority. Instead, developers are responsible for obtaining suitable signing certificates (typically self-signed, but they can be issued by a certificate authority—*e.g.,* Symantec[1]). On Android, the OS allows installation of app updates only if they are sanctioned by the same developer. Such *update integrity* is enforced in the OS by comparing the set of signing certificates embedded in the already installed application against the set in the updated version. If the updated version set of certificates matches the set in the previously installed app, the update is allowed. Otherwise, the update fails. When certificate sets are not equal, the only method to install the updated app is for the user to manually uninstall the old app (which deletes the app's user data) and then install the updated version as a new install. Initial app installations are not subject to such *certificate continuity verification* since Android uses a trust-on-first-use [21] approach.

Aside from the above uninstall/reinstall method, Android has no mechanism for developers to renew, change, or revoke signing certificate(s). Enabling OS-level support for certificate changes, informally called *key agility* or *certificate agility*, is the main focus of this paper.

Motivation for Certificate Agility.

Developers may want to update app-signing certificates for several reasons, including the following:

1. Secure app interaction. Developers (or development teams) may wish to leverage Android's signature-level privileges for UID sharing or signature permissions (see Section 2) to securely integrate apps or app components. Without certificate agility, developers must decide ahead of time which apps will interact securely. It may not always be possible to predict future functionality or required interaction of an application.

2. External certificate management. Developers may wish to use certificates issued by a certificate authority to assert a validated identity on their apps, perhaps ones they have already obtained. However, many reputable CAs will not issue certificates with what is essentially an infinite lifespan (Google requires 25 years or more for apps submitted to the Play Store [4]). A few CAs (*e.g.,* Symantec[2]) will issue these long-lived certificates but will not release them to the developer; the developer must use a CA-provided signing service to sign

[1] `http://www.symantec.com/en/ca/verisign/code-signing/android`

[2] *Supra*, footnote 1.

applications, which is an additional cost and necessitates distinct certificates specifically for Android apps. Without certificate agility, developers cannot renew an expired certificate and still update user apps without user interaction.

3. App ownership transfer. Applications can be sold or otherwise transferred to other developers. Under the current model, transferring private signing keys is a likely component of the ownership transfer process. However, such sharing of private signing keys is problematic if a developer signs multiple apps with the same signing certificate (a common practice [6]).

Our Proposal.

In this paper we motivate, design, and implement Baton, a set of software changes to Android's app installation framework and developer tools that allow signing certificates to be updated without user involvement, user data loss, or changes to the decentralized code signing model. In Baton, app updates include a certificate chain that is cryptographically verified at update time. Upon validation of a chain linking the signing certificate embedded in the currently installed version of an app to the certificate embedded in the newly installed version, updates are allowed (preserving user data) without requiring the user to uninstall/reinstall the app. Baton is incrementally deployable and fully backwards compatible with currently deployed apps. The Baton component of Android's installation framework is only invoked when certificate updates are required, imposing no new overhead during regular application use or software updates not involving signing key changes.

The concept of forward certificate chains, where an old key signs a new key (or alternatively, the new key is transmitted over a channel secured with the old key) is a long-known key-management technique, almost as old as public key certificates, with applications to encrypted email [25], TLS certificates [3], and Linux files [20].

Limitations of Existing Proposals.

	Enables Key Agility	Allows Skipping Updates	Compatible with Jarsigner	Incrementally Deployable	Allows Revocation
Stock Android [4]		●	●	●	
Baton (Sec 4)	●	●	●	●	○
Self-Signed Executables [19]		●		●	
Bin-Locking [20]	●			●	
Baton Alternative (Sec 4.2)	●	●		●	○
Central Certificate Authority	●	●	●		●

Table 1: A comparison of proposed update integrity mechanisms, including ones with key agility.

An early proposal [19] for binary file update integrity through self-signing suggests signed executables that include an embedded set of public signing keys, sufficient for verifying signatures. This approach, functionally similar to Android's app update mechanism, does not allow key agility. In subsequent work [20], the verification key set is allowed to evolve which enables key agility, however it requires the user to diligently download and install all issued updates sequentially; skipping updates could lead to missing verification keys for future updates, precluding such updates from being installed. By contrast, Baton enables key agility while allowing users to skip intermediate updates. In addition, Baton is designed specifically for Android in a way that is compatible with `jarsigner`. None of the above fully enable revocation, which is known to be difficult in the absence of a central authority. We address Baton and revocation in Section 6.3. Our comparison is summarized in Table 1.

Contributions.

Our main contribution is the proposal, implementation and testing of a mechanism supporting signing certificate agility on Android. The Baton design fits within existing constraints in the Android ecosystem, preserving compatibility with currently deployed applications, and does not negatively impact Android's secure app interaction policy (see Section 2). We explain the design of Baton, including which parts of Android were modified to support it, giving an overview of technical details. We additionally resolve unexpected issues that became apparent during implementation and testing. We also consider remaining related open issues in Android signing such as threshold signing, key compromise and revocation.

We intend to make the Baton patches and corresponding developer tools publicly available for download.

2. BACKGROUND (ANDROID)

We quickly review, for convenience, background on the packaging and signing of Android apps and discuss the role of digital signatures on apps for application interaction.

Android applications are packaged and distributed as compressed (zip) archives, usually with an APK file extension. A typical app archive contains at least: digital signature data (in the `META-INF` directory); and application metadata such as version strings, unique package name, and permission declarations (*AndroidManifest.xml*).

Every installed Android app must have a unique, developer-chosen *package name* defined in *AndroidManifest.xml*, and should follow standard Java naming conventions to avoid collisions amongst applications. In general, developers reverse their domain name for uniqueness (possibly appending a name for an app if there is more than one app per domain, *e.g.,* `org.mozilla.firefox`). The package naming convention may be enforced by application markets such as Google Play, but developers are free to claim their package namespace, or re-use an existing namespace.

The *AndroidManifest.xml* file also contains a *version code*,[3] which is a developer-chosen monotonically increasing integer independent of the user-visible *version name* (*e.g.,* v1 or 2.5). During updates, the OS compares version codes and only allows installation if the version code is set to increase (*i.e.,* app downgrades are never allowed).

Application signing and terminology.

All Android packages must be digitally signed to run on

[3] `http://developer.android.com/tools/publishing/versioning.html`

user devices or environment simulators [4]. During app development and testing, the standard development environment automatically creates self-signed debug certificates. For application release, Android allows developers to independently generate or obtain a key pair and a corresponding certificate that can be used for signing apps. The certificate may be self-signed, or issued by a certificate authority. While self-signed certificates are not required, they appear to be implicitly encouraged by Google, since the Play Store only allows apps with certificates expiring after 2033 to be published and many CAs won't issue certs with a 20 year lifespan.

Developers sign their code through the `jarsigner` tool, distributed as part of the standard Java development environment. During the signing process, `jarsigner` creates the `META-INF` directory inside the Android package, and adds three files:

1. `MANIFEST.MF` - A manifest file containing a list of every file name (except files in `META-INF`) in the archive at the time of signing, and a corresponding SHA1 hash for each entry.

2. `CERT.SF` - A file containing a SHA1 hash of each entry in `MANIFEST.MF` file, along with a corresponding file path.

3. `CERT.RSA` - The developer's X.509 certificate, usually self-signed using the RSA algorithm. This file also includes the signature of the entire `CERT.SF` file.

Apps can be signed with multiple keys, in which case the `META-INF` directory is populated with multiple certificates (one per signing key), manifests, and signature files. The signature(s) on an app can be stripped by simply deleting the `META-INF` directory.

Initial install.

On initial installs (*i.e.,* where the package name of the application being installed is not already associated with an app on the device), Android only verifies the integrity of the app by performing a signature verification process on all files (except those in the `META-INF` directory). There is no external verification of the developer's certificate at install time, even if a certificate signed by a CA is used.

Updates.

On application updates (*i.e.,* where there is already an app matching the package name of the app being installed), the signing certificate on the app being updated is compared with the certificate in the downloaded update. If the certificates are the same, the update is allowed following Android's certificate continuity verification. If the certificates are different, the update fails. We note that it is the certificates themselves that are compared, not the signatures. Thus, even if two certificates are signed with the same private key, updates are not allowed.

Uninstalling apps.

Applications on Android cannot uninstall other applications without user interaction. Uninstalling an app typically requires the user to load the on-device application market or application manager. Uninstalling an app removes all locally stored user data, but applications may store data elsewhere, such as the cloud or the SD card. The method in which that data is handled after uninstallation is up to the developer.

Secure application interaction.

Android uses signature information to allow apps sanctioned by the same developer(s) to communicate and share data securely. Two app interaction features that rely on signatures:

1. *UID sharing.* Android apps are assigned unique UNIX UIDs at install-time to enforce application isolation and sandboxing. When two or more apps are signed with the same key, developers can specify that they want these apps to be assigned the same UID, allowing mutual access to file storage or process space. UID sharing is common amongst modular applications such as plug-ins and extensions and eliminates the need to use excessive inter-process communication to transfer data between apps.

2. *Signature permissions.* Developers can define public interfaces available to other apps. Some interfaces may be sensitive, so they can be protected by developer-defined permissions. One type of developer-defined permissions is a signature permission, which can only be granted to applications signed with the same key as the application exposing the interface. Developers often use signature permissions to securely expose functionality and interact with other apps.

3. MOTIVATION: CERTIFICATE UPDATE

Although Android has existed since 2008, key agility functionality is extremely limited and has caused documented real world problems. This section describes two high-profile examples.

Google Authenticator.

In March 2012, Google changed the signing certificate for their two-step authentication app Google Authenticator (package name `authenticator`[4]). Google released a new application (under package name `authenticator2`) signed with a new signing key and included a certificate used to sign other prominent Google properties (*e.g.,* Maps, Chrome, and the Play Store app). The certificate switch was ostensibly required to enable secure interaction (see Section 2) between Authenticator and this set of apps.

The upgrade path from one version of Authenticator to the other required that users take a series of steps, including a manual new install and uninstall. To assist users, Google created a help page[5] explaining the upgrade procedure. Below is an excerpt:

> *[...]Once you have confirmed as part of the previous step that you are able to successfully generate valid verification codes using the new Authenticator, it is safe to uninstall the old version of the app. Because both versions have the same icon, make sure to check the version number before uninstalling: you want to keep version 2.15.*

[4] The full package name is `com.google.android.apps.authenticator2`

[5] Upgrading to Google Authenticator v2.15 `http://support.google.com/accounts/bin/answer.py?hl=en&topic=1099586&answer=2544996`

In Appendix A, we perform a usability analysis technique known as a cognitive walkthrough on this upgrade process. We find that the overall process is convoluted and should not involve the user. However, given the constraints, Google did mitigate many potential usability issues. With Baton, we aim to provide a mechanism by which, when developers update apps which include changed signing certificates, no additional interactions are triggered for end-users when the updated apps install. Baton would have allowed Google to issue a standard update to `authenticator` which includes the new signing certificate.

Mozilla Firefox for Android.

Before releasing Firefox for Android in 2010, Mozilla's intention appeared to be to use their existing Microsoft Authenticode certificates or to purchase a 2 or 3 year certificate from Verisign to sign the Firefox version for Android.[6] Mozilla correctly concluded that there is no support in Android for certificate renewal, even if there is no change to the signing key pair. Mozilla filed a bug report[7] on Android asking for confirmation or motivation for why certificate renewal is not supported. The bug report has remained unacknowledged and inactive since July 26, 2010 until the time of writing (May, 2013).

While the Android OS does not currently enforce certificate expiration (*i.e.,* apps with expired certificates can be installed as usual), the Android documentation [4] asserts that certificate validity is verified. This inconsistency leads us to believe that certificate expiration policies on Android may change in the future. Baton allows developers to use shorter lived (more typical lifespan) certificates, and update them as needed by issuing an application update. This may prove useful for companies, such as Mozilla, that already have (or wish to have) code signing certificates issued by certificate authorities.

4. BATON: DESIGN AND IMPLEMENTATION

Baton provides the ability for developers to delegate signing authority to a new private key. This is accomplished by creating a data structure (token) in which the old signing key is used to sign the new certificate and additional corresponding meta data. Each token is embedded in a certificate chain describing the history of delegations. The chain is cryptographically verifiable, and embedded inside subsequently released Android packages (*i.e.,* apps) after the first delegation occurs. The certificate chain and verifying code are implemented to meet the following design objectives:

1. *No user involvement.* Certificates and signatures are system-level components that need never be visible to the user. Baton provides a system-level mechanism to validate certificate changes and does not involve the user in any decisions or actions.

2. *Compatibility with Android's security model related to application signing.* Android uses certificates for software update continuity and for application interaction (see Section 2). Baton does not change the requirements for signature permissions and UID sharing.

3. *Minimal OS changes.* We add code to the Android application installation framework and developer tools, but make no other software modifications, and require no change of behavior by developers if certificates don't need to be changed.

4. *Backwards compatibility.* Baton supports incremental deployment with incremental benefit. Users with Baton-enabled Android will be able to upgrade applications that have changed their signing certificates (provided verification of the delegation succeeds). Users who don't use a Baton-enabled Android build can still install and upgrade applications that include Baton certificate chains. These users, as with current Android, will be unable to transparently apply software updates if there is a certificate change; instead they must uninstall the current version and install the update as a first install.

Baton is comprised of two components: (1) a set of patches to Android OS, modifying packages responsible for parsing the *AndroidManifest.xml* file and verifying application signatures; and (2) an Eclipse plug-in for assisting developers in generating key delegation metadata.

Delegating Signing Authority.

For a developer to successfully delegate (*i.e.,* endorse a new signing key) signing authority to a new signing key (shown as an example in Figure 1 as an update from $V2_{SigA}$ to $V3_{SigB}$), they must embed a valid delegation token in the update. In the example in Figure 1, a delegation token passing signing authority from $KeyA$ to $KeyB$ (*i.e.,* the private keys associated with certificates A and B, respectively) must be present in the update. The delegation token generation is described as step one in the signing key endorsement protocol given in Protocol 1.

4.1 Threat Model

We consider the following security objectives to be necessary for any secure update mechanism, including Baton:

1. *No Unauthorized Updates.* Updates to installed apps must be authorized (either directly or transitively) by the signer(s) of the originally installed version of the app.
2. *No Replays.* Key endorsement tokens should allow only the intended delegation of update authorization, and not allow unintended delegations through embedding potentially modified tokens on unintended applications.
3. *Mitigating Social Engineering.* The update mechanism should only require user actions that are easily distinguishable from the actions a target victim user would take in a social engineering attack.
4. *No Unauthorized Interaction.* Multiple apps may only interact through properly authorized privileged means (*e.g.,* sharing a UID or granting access to restricted APIs) with the mutual authorization of all the integrated apps.

We assume the attackers in the system to be computationally-bounded adversaries, who may hold their own signing

[6] `https://bugzilla.mozilla.org/show_bug.cgi?id=562843`

[7] `http://code.google.com/p/android/issues/detail?id=10020`

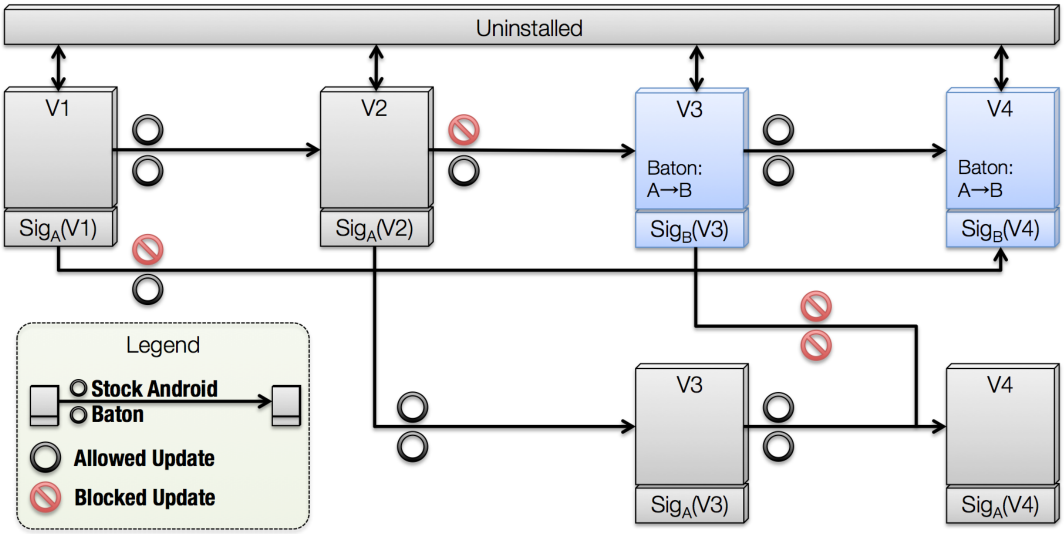

Figure 1: Version update diagram depicting updates that are allowed by stock Android (icons depicted over an arrow) and by Baton (icons depicted under the arrow). Sig$_A(Vn)$ and Sig$_B(Vn)$ are signatures on application version n with signing keys A and B, respectively. In all cases, updates are only allowed if signatures are successfully verified.

keys, have their own apps released on the market, and even have apps installed on a target user's phone. We assume the adversary is not capable of learning the private signing keys of other developers (we discuss key compromise in Section 6.2), nor are they able to modify or otherwise compromise the Android OS. We assume, however, that the adversary can tamper with any Android application package. A security analysis (see Section 5.3) is given after first describing the details of Baton.

4.2 Implementation

Certificate Chain and Delegation Tokens.

In Baton, a *certificate chain* is a sequence of one or more delegation tokens. Each delegation token in the certificate chain is a signed collection of metadata including the following information:

1. The application package name.
2. The application version code.
3. A set[8] of previous certificates.
4. A set of current certificates.
5. A hash of the previous delegation token in the certificate chain.

A Baton delegation token acts as an endorsement of a transition from a set of previous certificates (*i.e.,* used to sign an already installed application version) to a new set of certificates ('current certificates') whose corresponding private keys will be used to sign the current version of the application. The generation of the delegation token is described in Protocol 1.

Each delegation token, including a signed hash of the delegation token prior to itself in the chain, allows cryptographic verification of the entire certificate chain. This prevents an adversary from removing, adding, or rearranging delegation token elements in the chain. Inclusion of the package name scopes the delegation to only the specified application. For example, if a developer signs three applications with the same signing key and generates a Baton delegation token to update the certificate of only one of the three applications, the scope prevents this same token from being embedded in the other two applications, as the package name will not match.

Baton XML.

Baton applications embed into the *AndroidManifest.xml* an XML representation of the certificate chain. To simplify the signing and verification procedure we detach[9] the delegation token signatures from the delegation token metadata to create two separate sets of nested elements, `certificate-chain` and `certificate-chain-signatures` (see Figure 2). The `delegation-token` elements in `certificate-chain` are matched to the corresponding `delegation-token-signature` elements in the `certificate-chain-signatures` by order within their parent element. The signing process is enacted following the `xmldsig` [1] standard best practices[10] outlined by the W3C working group.

To allow signature validation in the case of missed intermediate updates, each delegation token includes a Base64 encoding of each certificate in the previous certificate set as well as their fingerprints. Baton embeds the full certificate for each of the `previous-certs` as a convenience to handle updates from a very old application to one signed with a set of certificates valid only after processing several delegation tokens transitively. In this case, the certificates specified in intermediate tokens may not be present within the OS and must be loaded from the encoded version in the token.

[8] Baton assumes developers may use multiple signing keys on the same application. See Appendix B.

[9] http://www.w3.org/TR/xmldsig-core/#def-SignatureDetached

[10] `http://www.w3.org/TR/xmldsig-bestpractices/`

Protocol 1: Baton signing key endorsement protocol

Overview: The holder of $KeyA$ wishes to delegate signing authority to a new key $KeyB$.

Variables:
$KeyA, KeyB$ - the private keys corresponding to the public keys in $CertA$ and $CertB$ respectively.
$CertA, CertB$ - the signature verification certificate used to verify signatures on current application release, and the certificate being delegated to, respectively.

Pre-requisites: The fingerprint of $CertB$ has been communicated to the holder of $KeyA$ over a channel with guaranteed integrity.

Protocol:

1. Holder of $KeyA$ generates *token*=Sig_{KeyA}\{H(pkg name, version code, $CertA$, $CertB$ fingerprint, previous token hash†)\}.
2. *token* is communicated to holder of $KeyB$.
3. Holder of $KeyB$ includes token in *AndroidManifest.xml* when releasing updates signed with $KeyB$.

†: If there is no previous token to hash (*i.e.,* it is the first token to be included in a certificate chain) null may be substituted for the previous token hash value.

AOSP Implementation.

We modified The Android Open Source Project (AOSP) code to implement the Baton key agility functionality. The proposed set of patches totals under 500 lines of code which we plan to make available[11] under an open source license compatible for inclusion in AOSP.

The *android.content.pm.PackageParser* core class was modified in order to allow processing of the new *AndroidManifest.xml* entries. In the AOSP services subproject, the *com.android.server.pm.PackageSignatures* class was modified to store a *SignatureChain* reference, populated by the `PackageParser`. When the *com.android.server.pm.Settings* class loads or stores the on-disk *packages.xml* file, the *SignatureChain* is responsible for writing its own representation to the file, and restoring it when the operating system boots.

The *com.android.server.pm.SignatureChain* class is the only new class added to AOSP by Baton. This class is an in-memory representation of the XML certificate chain from the *AndroidManifest.xml*. It contains the logic for reading the *SignatureChain* to/from XML, as well as verifying delegation token signatures.

Finally the *com.android.server.pm.PackageManagerService* class was modified to change the signature verification logic and package update procedure. When the modified `PackageManagerService` processes an update for an installed application, it will now compare the installed application's set of signing certificates to the proposed update's set of signing certificates. If the sets match, the update proceeds following the existing Android certificate continuity policy. If the certificate sets do not match, then the `PackageManagerService` mandates that the proposed update must contain a delegation token for the correct version transition (*i.e.,* for the currently installed version to the update's version) endorsed by the installed application's certificate set. The version transition may be endorsed through one or more intermediate delegation tokens allowing the update to proceed in the event the user has missed interim updates.

The AOSP project does not include the `javax.xml.crypto.dsig` packages used to verify XML signatures. Therefore we additionally include the Apache Santuario[12] library, an independent implementation of the `xmldsig` [1] standard.

Developer Tools.

To facilitate adoption by developers, Baton augments the application signing life-cycle by integrating with the development environment used to produce Android application releases. As of writing, the official Android Developer Tools (ADT) plugin for Eclipse[13] is closed source, preventing a patch to enable Baton support directly. In lieu of a patch to ADT, we have opted to provide a third party Eclipse plugin.[14] After installing the Baton plugin, a developer is able to export their Android projects as a Baton-enabled APK. In addition to the Eclipse plugin, the Baton developer tools can operate as a stand-alone GUI, or a command line tool. The stand-alone versions of the plugin are more suited for integration with other IDEs that support external tools, or with more complex build management systems often used with large software projects.

In Eclipse, when the developer right clicks an Android project and chooses "Add Baton," the plugin will generate the XML for a certificate chain and insert it automatically into the project *AndroidManifest.xml* file. The developer is then able to launch a Baton token generation wizard. The wizard follows the protocol outlined in Protocol 1 by walking the developer through the process of generating a delegation token. To start, the developer is prompted to select one or more signing certificates to endorse for future updates, or to enter the certificate fingerprint. For example, this may be a certificate generated by the new owner of the project if the developer is transferring control (*e.g.,* selling to another developer) of their application. The developer must also choose one or more signing certificates whose corresponding private keys will be used to sign the delegation token to endorse it. In practice the developer will most often select the signing certificates presently used to sign production releases of their project. The version code and the package

[11] Proof of concept available upon request.

[12] `http://santuario.apache.org/`

[13] `http://developer.android.com/tools/index.html`

[14] Proof of concept available upon request.

```
<manifest ... >
 ...
  <certificate-chain>
    <delegation-token android:versionCode=
        "10">
      <previous-certs>
        <cert encoded="..." fingerprint=
"907EB3F2E8447054446A2A4B3ED8CA78DB04B188"
    />
      </previous-certs>
      <current-certs>
        <cert fingerprint=
"6E532E87A468052DA2EE8E9D6E56080181D3E2F9"
    />
      </current-certs>
      <previous-token hash="null" />
    </delegation-token>
  </certificate-chain>

  <ceritificate-chain-signatures>
    <delegation-token-signatures>
      <Signature>
        ....
      </Signature>
    </delegation-token-signatures>
  </certificate-chain-signatures>
  ...
</manifest>
```

Figure 2: Example Baton certificate chain entry in *AndroidManifest.xml*. Base64 encoded certificate content and non-Baton related Android entries removed for brevity. Signature element defined in `xmldsig` [1] standard.

name values in the token are pre-filled from the values in the *AndroidManifest.xml* file. Commonly, the private keys corresponding to the chosen token signing certificates will be protected by a passphrase, which the developer must enter to unlock the private keys. After unlocking the private keys (if required) the XML for the delegation token is generated, signed, and inserted into the certificate chain in the *AndroidManifest.xml* file. It is also displayed to allow communication of the token to other parties.

Design Alternatives.

It is possible to avoid embedding a certificate chain at all by retaining all previous versions of *AndroidManifest.xml* and the `META-INF` directory in future versions of the APK.[15] Thus, signing a new APK will bind the current *AndroidManifest.xml* to the complete history of previously signed *AndroidManifest.xml* files. In this case, to delegate control of the application to a new set of signing certificates, the *AndroidManifest.xml* specifies the new certificate in an APK signed by the current certificate. With Baton, certificate delegations only add a few lines to *AndroidManifest.xml* instead of creating an archive of past files containing mostly irrelevant information (but must be retained in order for the signatures to be verifiable by the OS).

5. EVALUATION

This Section discusses compatibility of Baton with other signature-based security mechanisms and previous versions of the OS. Additionally, we perform a security analysis of Baton.

5.1 Compatibility

Compatibility with Related OS Functions.

Android currently uses code signing certificates for security operations outside of application update integrity. Code signing certificates provide a form of access control, selectively allowing applications to join a shared UID group or to be granted a signature permission (see Section 2). At application install time, for the application to be allowed access to a signature protected resource (*e.g.,* UID group or permission), the certificates on the application must be identical to those associated with the protected resource. This requirement remains unchanged in Baton.

Using Baton, if one application in a shared UID group or an application providing a signature protected permission transitions to a new set of signing certificates, the certificates associated with the group or permission within the OS are updated to reflect the transition. As a consequence, future updates to other applications in the UID group, or requesting the signature permission must also transition to the new set of signing certificates.

We have designed Baton such that developers cannot issue a certificate transition to one application that "evicts" other applications from a UID group by changing the associated certificate set. This behavior is consistent with Android's current model, where applications cannot arbitrarily change UID groups during updates.[16]. An eviction would, by definition, change the UID of the evicted application, leading to an inconsistent state. Our design instead honours the membership of already installed applications until they are updated. This prevents previously functional applications groups from losing functionality (by evicting a member) while introducing no detrimental security properties.

Compatibility with Stock Android.

Application releases that perform a certificate transition using Baton must package a modified *AndroidManifest.xml* containing a Baton certificate chain in the released APK. For this reason, we consider the compatibility of the modified application release with existing versions of Android (*i.e.,* those without Baton).

At application install-time, the Android OS parses the *AndroidManifest.xml* using the `android.content.pm.PackageParser` class (located within the frameworks directory of AOSP). Once patched to enabled Baton support, the Android OS is aware of the new XML tags introduced for the certificate chain (see Figure 2), and can react accordingly. The default unmodified behaviour of the `PackageParser` class, as of the time of writing, sets an internal `RigidParser` constant to **false** causing unrecognized XML tags to be skipped without error. Based on the behaviour of the code in AOSP, if a Baton enabled application release is installed on an unpatched OS, the certificate chain will be ignored. The application will still install correctly pending successful certificate continuity validation.

[15] The hash tree structure of `MANIFEST.MF` allows the signature on a single file to be verified.

[16] `https://android.googlesource.com/platform/frameworks/base/+/d0c5f515c05d05c9d24971695337daf9d6ce409c`

Assuming Baton was merged into AOSP, we recommend developers who release application containing a Baton certificate chain use the `android:minSdkVersion` parameter in *AndroidManifest.xml* to preclude install on systems lacking Baton support. It is unavoidable that users without Baton support may only install application updates signed with a different set of signing certificates by first uninstalling the old application.

5.2 Implementation Evaluation

Traditional update.

We tested Baton by creating multiple releases (each with an incremental version code) of a test application, and sideloading each version on to an emulated Android environment in various orders (*e.g.,* V1-V2-V3, V1-V3, V2-V3-V1). All releases were signed with the same signing certificate and used the same package name. As per stock Android policy, updates succeeded but downgrades did not. The user experience was no different in the Baton environment and in the unmodified Android environment.

Changing certificates.

We created a sample application with an embedded Baton certificate chain and tested delegating signing authority from one certificate to another (keeping the package name the same). After installing the application signed with one certificate, the app was transitioned to a new certificate by embedding a token generated by the Baton developer tools in an update. Baton successfully validated the delegation token, and user data related to the test app was preserved and accessible by the application with the new certificate. We also tested changing certificates but not including the Baton certificate chain, as well as changing certificates and including an invalid certificate chain. These updates failed with the "failed inconsistent certificate error" thrown by the Android OS as expected.

AOSP Unit Tests.

We ran the bundled unit tests for the `PackageManagerService` class. These unit tests are included with the AOSP source code and are used for automated testing and to prevent any bugs from being introduced to previously functional code. We checked that the Baton system introduces no such regression errors by running the unit tests and verifying that a Baton patched system to pass the tests without failure or warning.

AOSP Code Inspection.

We searched the AOSP source code tree looking for references to signing certificates. Code found interacting with the `PackageManagerService`, or with the `Signature` objects used internally to represent code signing certificates was manually inspected and examined for conflicts with Baton. No conflicts were discovered.

5.3 Security Analysis

Here, we informally analyze Baton under the security objectives and threat model presented in Section 4.1.

No Unauthorized Updates. Baton does not modify the requirements of Android's standard certificate continuity verification. Baton only introduces cryptographic verification of certificate chains. Thus, with Baton, an adversary must still compromise a developer's private signing key to issue an update or create a valid certificate chain to transition to a new signing certificate.

The certificate chain and delegation tokens in Baton are included in the *AndroidManifest.xml*. They are not secret; digital signatures provide integrity protection. Deleting the chain or delegation tokens inside the *AndroidManifest.xml* has the same effect as removing a signature from an Android package (deleting the `META-INF` directory, also known as signature stripping [6]). Apps without a Baton certificate chain or `META-INF` directory will fail to validate as legitimate updates and will not succeed in replacing an installed binary.

No Replays. Delegation tokens include a package name, version code, and are digitally signed (see Section 4). Replaying a delegation token on a different application (*i.e.,* copying the relevant section of the *AndroidManifest.xml* file into an application with a different package name) causes Baton to fail the certificate chain verification. Certificate transitions do not succeed unless all tokens in the chain reference the package name being updated, and corresponding signatures can be verified.

Mitigating Social Engineering. With Baton, users apply app updates as usual. However, unlike with stock Android, there is no legitimate reason to require the user to manually uninstall applications for the purpose of a key update. Training users that sometimes this action may be required can lead to social engineering attacks by malicious developers; Baton eliminates the need to do so, reducing this risk.

No Unauthorized Interaction. Baton does not modify UID sharing nor signature permission requirements. Applications must be signed with the same signing key(s) at install time to leverage signature privileges (see Section 5.1). It is not possible to leverage Baton to arbitrarily join a UID group for which a key is not held.

6. DISCUSSION

This Section discusses practical implications of enabling key agility on Android.

6.1 Certificate Expiration

The Android OS currently ignores the validity of signing certificates at install time, despite official documentation stating otherwise [4]. As of Android 4.2, we have verified that it is possible to install (without warning or user intervention) apps with a signing certificate that has expired. Our belief is that Android does not currently check for certificate expiration at the OS level because there is no way to update or renew certificates in applications. Additionally, the Google Play Store requires that all apps submitted carry a certificate valid for at least 25 years, making expiration verification redundant for marketplace installations. With Baton, certificate renewal is possible, thus enforcement of certificate validity can be re-enabled, limiting the impact of key compromises (see below) and allowing the optional use of CA-issued signing certificates that have more generally acceptable validity periods (*e.g.,* 1–5 years).

6.2 Private Key Compromise

In the current Android security model, if a developer's private signing key is compromised by an attacker, the attacker may permanently release unauthorized updates. If the signing key is used on an app distributed on an application marketplace, the attacker would need to successfully gain access to the marketplace account to publish an update. Alternatively, the attacker could convince users to sideload the unauthorized version (*e.g.,* from a non-official site). Similarly in Baton, unauthorized updates will be possible if keys are compromised. This includes both standard updates as well as updates with a certificate chain. Developers using Baton must protect signing keys as usual. Additionally, if key compromise is detected in a timely fashion, it may be possible for the legitimate developer to issue a Baton app update *before* the adversary, effectively "locking" users who upgraded into a new uncompromised replacement signing certificate.

6.3 Transferring Authority

Using Baton, developers can delegate signing authority to the holder of a different key, but the original certificate and corresponding key pair will remain authorized for issuing updates to versions of the app not containing the certificate chain. For example, when an app is being sold, the seller may continue to issue updates to the app under the original key (see $\mathrm{V2}_{SigA} \rightarrow \mathrm{V3}_{SigA}$ in Figure 1). Clients who do not update to the Baton version ($\mathrm{V3}_{SigB}$) may be tricked into installing updates with the old signing certificate instead. While there is generally a trust relationship established when ownership of an app is being transferred (*e.g.,* the buyer is already exposed to potential backdoors in the app), best practices would encourage revoking the original certificate from updating the app. This must be a finer grained revocation than certificate revocation: a seller of an app may have other apps signed with the same certificate that are not being sold. We consider two conditions—with and without a marketplace—under which apps could require a proper transfer of signing authority.

Assisted by a central marketplace. When an app is installed through an application marketplace, there are effectively two authentication mechanisms in place to ensure source continuity: the signature enforced at the OS-level and the developer account with which the app is associated at the market-level. Application marketplaces such as Google Play Store allow developers to transfer apps to another account,[17] which effectively prevents an app seller from continuing to issue updates through the marketplace. For marketplace users, app updates will proceed as usual. Users who install apps from multiple markets or by sideloading may still be vulnerable to installing unauthorized updates that are signed with the original developer's key.

Without a central marketplace. When users install apps from only side-loaded sources, it seems difficult to communicate the revocation of a certificate's signing authority over a specific app. Certificate revocation remains a open problem in self-signed environments, where no single entity is authoritative except perhaps the OS itself.

Detection instead of prevention. It could prove advantageous to keep a public record of package names and associated certificate chains as a type of public notary to identify if different certificate chains emerge for the same app. This principle can be seen in other domains: *e.g.,* Convergence [21] and Certificate Transparency [2] which aim to detect fraudulent certificates in TLS. It could be used in Android to confirm the uniqueness of a certificate chain at install-time. Baton could be augmented to submit certificate chains or query valid chains for a given application by leveraging an install-time server query mechanism like that of Barrera *et al.* [7]. The server-side component, which reports back on valid or invalid chains, would require manual curation by experts.

6.4 Threshold Signatures

Presently in Android, while a set of more than one key can be used to sign an application, all keys used to sign a given application must sign each subsequent update. Baton does not change this requirement but it could be easily extended to allow a larger set of keys to be authorized for signing updates than the subset (or threshold) required for each update. More complicated access structures could be supported as well (*i.e.,* logical rules specifying the combination of keys necessary). All of this can be currently achieved using an external threshold signature mechanism [18], but Baton would remove the complexity of doing so. Threshold signatures can improve resilience to key compromise by requiring multiple shares to be compromised before the full signing key can be reconstructed by an attacker.

6.5 Limitations

One of Baton's main limitations is the need for developers to include the certificate chain (which includes corresponding full certificates) in potentially all[18] subsequent versions after a certificate transition. Failure to include the chain of certificates would prevent users who have not yet upgraded to the latest version from seamlessly upgrading, since there is otherwise no easy way to verify the chain. Android certificates are typically 600 bytes to 2 kilobytes, so overall app size is not expected to be adversely impacted by including several certificates. Since certificates and certificate chains are intended to be public, backup copies may without risk be stored in the cloud or on a shared drive.

Key loss, even with Baton, remains a difficult problem. Losing a signing key means it is no longer possible to issue a Baton certificate update, unless a threshold system (see Section 6.4) is used. We believe solving this limitation would weaken Android's overall security model since a mechanism to issue an update without the original key could be abused by an adversary.

7. RELATED WORK

A comparison with proposals [19, 20] closely related to Baton was given in Section 1. In the broader literature on software updates, Cappos *et al.* [9] examine security issues in package managers which commonly distribute verifying keys as part of the installation media. Samuel *et al.* [16] describe a software update framework (TUF) that is resilient to a number of key compromise attacks. TUF overlap is essentially

[17] `https://support.google.com/googleplay/android-developer/contact/publishing`

[18] The certificate chain should be included in all subsequent versions which the developer wishes to allow transparent upgrades from, or as long as there is reason to believe not all users have performed the most recent certificate transition.

an alternate PKI tailored to allow multiple roles (*e.g.,* release and timestamping) such that an adversary would have to compromise multiple keys to trick a user into installing a malicious update. Baton leverages Android's TOFU model and focuses on the continuation and delegation of the initial trust without the need for a central PKI.

Specific to Android, Barrera *et al.* [6] examine Android's update integrity mechanisms, noting the lack of key agility and briefly discussing the Google Authenticator example. While the paper has several proposals, including improvements to UID sharing, none directly address key agility, the goal of Baton. Much of the existing body of Android security research has focused mainly on a three areas: Android app analysis and malware [12, 14, 24]; permissions [13, 5]; and privilege escalation through IPC and covert channels [8, 17, 15]. Our work is focused on a less explored area of the Android security literature: the security of signing keys and certificate evolution.

8. CONCLUSION

This paper has discussed, including motivation from real-world instances involving high-profile applications, the need for a mechanism to allow changing of signing keys and certificates associated with Android apps. We have designed, implemented and tested a mechanism to do so, Baton, as an enhancement to the Android installer and developer tools. Baton builds upon well known concepts of establishing cryptographically verifiable trust chains between certificates. Baton places the responsibility of ensuring integrity and authenticity of updates on the developer and the OS, lightening the load on the user.

Acknowledgements.

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)—the first author through a Canada Graduate Scholarship; the third 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.

9. REFERENCES

[1] W3C Standard: XML signature syntax and processing. `http://www.w3.org/TR/xmldsig-core/`.

[2] Internet-Draft: Certificate transparency, 2012.

[3] Internet-Draft: Public key pinning extension for HTTP, 2012.

[4] Signing your applications. `http://developer.android.com/tools/publishing/app-signing.html`, Accessed Feb. 18, 2013.

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *ACM CCS*, 2012.

[6] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android. In *ACM SPSM: Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[7] D. Barrera, W. Enck, and P. C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *IEEE MoST: Mobile Security Technologies Workshop*, 2012.

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

[9] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *CCS*, 2008.

[10] J. Clark, C. Adams, and P. van Oorschot. Usability of anonymous web browsing: An examination of Tor interfaces and deployability. In *SOUPS*, 2007.

[11] L. Cranor. Privacy policies and privacy preferences. In *Security and Usability*. O'Reilly, 2005.

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

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

[14] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. *ACM SPSM: Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

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

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

[17] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, 2011.

[18] V. Shoup. Practical Threshold Signatures. In *EuroCrypt*, 2000.

[19] P. van Oorschot and G. Wurster. Self-signed executables: Restricting replacement of program binaries by malware. In *HotSec*, 2007.

[20] P. van Oorschot and G. Wurster. Reducing Unauthorized Modification of Digital Objects. *IEEE Transactions on Software Engineering*, 38(1), 2012.

[21] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Usenix ATC*, 2008.

[22] C. Wharton, J. Rieman, C. Lewis, and P. Polson. The cognitive walkthrough method: A practitioner's guide. In *Usability Inspection Methods*. Wiley & Sons, 1994.

[23] A. Whitten and J. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *USENIX Security*, 1999.

[24] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE S&P*, 2012.

[25] P. Zimmermann and J. Callas. *The Evolution of PGP's Web of Trust*, chapter 7. O'Reilly, 2009.

APPENDIX

A. COGNITIVE WALKTHROUGH OF THE GOOGLE AUTHENTICATOR UPGRADE PROCESS

To illustrate the deficiencies in the process currently required to modify an application's signing certificate(s), *i.e.,* by requiring users to be involved in updating the application, we perform a cognitive walkthrough of the update process as implemented by Google when switching signing keys for their Authenticator app (see Section 3). We refer to the versions of Authenticator with the initial signing key certificate[19] as Auth1 and the versions with the current certificate[20] as Auth2. Technically, these are considered by the OS to be distinct applications and thus must each have a unique package name[21]. However, users never see package names on Android. Apps are displayed to the user with an app name and icon, which are identical in both Auth1 and Auth2.

A cognitive walkthrough aims to shed light on the user experience of performing a specific task by relying on the interface for guidance. In a cognitive walkthrough, the evaluator (both a domain and usability expert) performs the core tasks required of the user and evaluates the experience against a set of guidelines or heuristics.

We consider a single core task: migrating from an installation of Auth1 to a fully functional installation of only Auth2. Since the core task is software installation, we looked to the literature for usability guidelines for installation, rather than regular software use, and borrow the installation-relevant guidelines from a cognitive walkthrough of the installation and use of Tor [10]. These guidelines, in turn curated from the literature, are:

(G1) Users should be aware of the steps they have to perform to complete a core task [23].

(G2) Users should be able to determine how to perform these steps [22, 23].

(G3) Users should know they have successfully completed each core task [22, 11].

(G4) Users should be able to recognize, diagnose, and recover from non-critical errors [22].

(G5) Users should not make dangerous errors from which they cannot recover [23].

(G6) Users should be comfortable with the terminology used in any interface dialogues or documentation [22, 11].

A.1 Evaluation

Since Auth2 is technically a new installation instead of an update (we assume users can perform standard updates), it will not appear as an update in the Play Store. Thus, a user of Auth1 must first become aware of the existence of Auth2 through some other means (G1). Upon launching the latest (and last) version of Auth1 (v0.91), the user will encounter a prominent ribbon bar displayed at the top of the screen noting that the app will "no longer be supported." The phrase "Learn More" is offered as a link. The warning conforms to G6 but does not communicate the idea that a new version is available (G1), as opposed to the app simply being abandoned. Users may grasp that no more updates will be issued and then henceforth ignore the ribbon, never completing the core task of updating to Auth2.

If the user taps on "Learn More", they are then informed in plain language (G6) that a new version is available and are directed to the Play Store to download it. This information is sufficient for G1 and G2, and should be displayed directly in the app screen without requiring a user click-through to read it. The Play Store page for the application displays no information that distinguishes Auth2 from the already installed Auth1—it has the same app name and icon, and no language about the unusual update process for this particular app is present. A user may conclude they already have the app (contra G3). Diligent users, however, will notice the install button, which does not appear if an app is already installed (it is replaced with the option to open or uninstall).

If the user clicks to install Auth2, the app installs, automatically launches and transfers the user data from Auth1 to Auth2. (Technically, arranging for the app to open without a user click and securely transfer the data, which is private data used for authentication, requires sophisticated instrumentation of both apps by very good developers.) This automation prevents dangerous errors (G5). The user is then notified in plain language (G6) that the data has been transferred (G3) and is prompted to "uninstall the prior version of the app" (G1 and G2).

If the user clicks to uninstall the app, the OS displays a dialogue containing the app icon, app name (Authenticator), and question "do you want to uninstall this app" (G6)? In isolation, this screen does not adequately communicate to the user that the prior version is being uninstalled. Were the user instead to cancel the prompt to uninstall, perhaps believing that they do not want to uninstall what appears to be the exact app they just installed (based on the name and icon), they would have on their homescreen two identical icons with identical names and no indication of which is Auth1 and which is Auth2. If they manually uninstalled Auth2, they will lose their data (G5). However, if they opened Auth1, a warning would appear stating that the new version is already installed and offering to uninstall this version (G4). In addition, the user data is no longer available and the app is no longer functional (G4).

The user may successfully complete the core task by uninstalling Auth1 by following the instructions to do so when prompted during the installation of Auth2, or at any time later by following the prompts in either Auth1 or Auth2.

A.2 Interpretation of Results

The intention of our cognitive walkthrough is not to criticize Google's handling of Authenticator's certificate migration. If anything, the process was relatively seamless, much of it automated, with care given to preventing dangerous errors and allowing recoverability. While there is room for improvement, this represents a nearly ideal execution of the certificate update process under the constraints of the OS. However in the hands of less skilled developers (*e.g.,* without clear instructions or the automation of the data transfer

[19] SHA: `38918A453D07199354F8B19AF05EC6562CED5788`

[20] SHA: `24BB24C05E47E0AEFA68A58A766179D9B613A600`

[21] Respectively: `com.google.android.apps.authenticator` and `com.google.android.apps.authenticator2`

```
1 int checkSignaturesLP(Signature[] s1,
      Signature[] s2) {
2   ...
3   final int N1 = s1.length;
4   final int N2 = s2.length;
5   for (int i=0; i<N1; i++) {
6     boolean match = false;
7     for (int j=0; j<N2; j++) {
8       if (s1[i].equals(s2[j])) {
9         match = true; break;
10      }
11    }
12    if (!match)
13    return PackageManager.
          SIGNATURE_NO_MATCH;
14  }
15  return PackageManager.SIGNATURE_MATCH;
16 }
```

Figure 3: Android OS signature checking method used in versions 2.2 and earlier.

```
1 int checkSignaturesLP(Signature[] s1,
      Signature[] s2) {
2   ...
3   HashSet<Signature> set1 = new HashSet<
        Signature>();
4   for (Signature sig : s1)
5     set1.add(sig);
6   HashSet<Signature> set2 = new HashSet<
        Signature>();
7   for (Signature sig : s2)
8     set2.add(sig);
9   // Make sure s2 contains all signatures
        in s1.
10  if (set1.equals(set2))
11    return PackageManager.SIGNATURE_MATCH;
12  return PackageManager.SIGNATURE_NO_MATCH;
13 }
```

Figure 4: Android OS signature checking method used in versions 2.3 and higher (current).

process upon install), the process could be much more difficult for users. Since Android currently leaves this migration process to app developers, we are apprehensive of how bad a less thoughtful execution could be, and note that a consequence of user error could be data loss.

By contrast, Baton removes all the uncertainty of developer execution and user behaviour from the equation. With Baton, the same core task can be accomplished through a standard update indistinguishable from any other update, which we already assume a user can perform. Thus we can conclude, even without a cognitive walkthrough or user study, that any user able to update apps can use Baton to complete the core task.

B. CERTIFICATE SETS

Android applications can be signed with multiple private keys, generating additional signatures inside the `META-INF` directory. As explained in Section 4.2, Baton is implemented to maintain compatibility with multiple signing keys, and thus treats signatures and certificates as sets.

In stock Android, before applying application updates, the following certificate comparison is performed (see Figures 3 and 4). On current releases of Android (2.3 and higher), the installer builds two Java `HashSet`s from the available certificates[22] in the currently installed app (s1) and the app being installed (s2). If the sets match (*i.e.,* all elements in the one set are identically present in the other), the update is allowed, otherwise the update fails. The code in Figure 4 (found within the Android source tree) prohibits developers from ever changing either the number of signatures (and corresponding certificates) in apps, or the certificates themselves.

In previous versions of Android (2.2 and earlier), it was possible to shrink the certificate set as long as all certificates in the old app version were also present in the new (updated) version. As seen in Figure 3, this functionality (intentional or otherwise) was possible by iterating through the certificate set in the new app and looking for matches in the currently installed app. Ensuring that N1 is equal to N2 before entering the loop on line 5 would make the two code snippets functionally equivalent.

[22] Although the source code references "signatures", signature objects are simply DER-encoded byte representations of X.509 certificates.