Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems

David Barrera
School of Computer Science
Carleton University

William Enck
Department of Computer Science
North Carolina State University

Paul C. van Oorschot
School of Computer Science
Carleton University

***Abstract*—Application markets providing one-click software installation have become common to smartphones and are emerging on desktop platforms. Until recently, each platform has had only one market; however, social and economic pressures have resulted in multiple-market ecosystems. Multi-market environments limit, and in some cases eliminate, valuable security characteristics provided by the market model, including kill switches and developer name consistency. We outline a novel approach to retaining single-market security semantics while enabling the flexibility and independence of a multi-market environment. We propose Meteor as a security-enhancing application installation framework that leverages information (e.g., app statistics, expert ratings, developer history) from a configurable set of security information sources. We build a proof-of-concept Android application (Meteorite) to demonstrate the technical feasibility of our proposal. The Meteor approach provides valuable decision-making criteria useful not only for smartphone users, but technology consumers as a whole, as new and existing computing environments converge on a market-like model for software installation.**

I. Introduction

Consumer computing as we know it is currently undergoing a transition. The emerging software ecosystem frequently commoditizes functionality into discrete, purpose-driven applications ("apps"). Apps are plentiful, diverse, and frequently redundant. For example, a smartphone app consumer is often presented several, if not tens, of options when searching for to-do lists, location-aware utilities, and games.

Smartphones are at the forefront of this computing transition. However, it is likely that general computing will follow suit. Recent market statistics report more smartphones are sold per month than PCs [24], indicating a new breed of users whose only exposure to computing is the app-centric model.

Application markets for smartphones, such as Google's Android Market and Apple's App Store, provide *one-click software installation*. These markets have become the *de facto* method of installing apps on smartphones, as they serve as a central point of app distribution, sales and discovery.

A single, central application market offers an opportunity to improve consumer security. Thus far, this model has exhibited several clear advantages. First, a market acts as a centralized chokepoint for detection of malware (e.g., Google's Bouncer [16]).[1] Second, it provides a means to remotely uninstall distributed apps later identified as malicious. These remote uninstalls (or "kill switches" [6], [5]) provide faster clean-up than traditional antivirus software, as they push actions to devices and do not require definition updates or resource intensive scanning. Finally, search results display consistent developer names for applications. Once a developer has registered with the market, controls exist such that no other developer can easily distribute applications under that developer name. Hence, consumers have some assurance that all applications provided by "John Smith" are provided by the same John Smith. While this characteristic does not ensure that "John Smith" is the John Smith the consumer intended, it does allow the app market to apply sanity checks for well known, high-impact entities, e.g., "Bank of America."

Smartphone platforms such as Android have been designed to allow multiple application markets. Multiple-market environments are an inevitable response to social and economic pressures. For example, Apple is frequently chastised for denying distribution of apps that do not meet its moral. On Android, the Amazon Appstore offers sales and anti-piracy mechanisms, and the MiKandi market publishes adult applications deemed inappropriate for the official Android Market. However, while this flexibility and independence provides social and economic benefits, it undercuts valuable security properties such as kill switches and developer name consistency.

In this paper, we examine the convergence of one-click installation and multiple app markets. Specifically, we first identify security properties that are lost by allowing the existence of multiple application markets. We then propose a software installation framework with the objective of achieving single market security semantics while retaining the flexibility and independence provided by a multiple-market setting. Our proposed solution enhances app installation with an extensible set of configurable security *information sources* and *kill switch authorities*. Information sources provide the user with additional app information, ranging from app age, virus reports and privacy violations, to expert ratings on the app and other apps by the developer. Kill switch authorities allow consumer devices to be configured to subscribe to notifications of dangerous apps for removal.

IEEE MoST 2012 (Mobile Security Technologies workshop). This paper updates and obsoletes an earlier version that appeared as a Carleton University School of Computer Science technical report [4].

[1]We note that while market-level app vetting can be useful under some circumstances, the practical effectiveness and scalability of this approach remains uncertain [18].

A fundamental goal of our approach is to connect digital properties such as package signatures to human evaluable information such as developer and application names. Our key observations are that 1) name collisions in both developer and application names should be minimal; and 2) name collisions should in fact raise suspicion, i.e., the more frequent cause of a name collision is malicious substitution of an app.

Note that while our approach mitigates many threats introduced by a multiple-market ecosystem, it cannot address all threats. In particular, our architecture cannot automatically determine the most (or least) secure application for a specific task. This is a fundamental problem of app distribution even for single-market ecosystems, and is outside the scope of this paper.

Contributions. We believe we are among the first to look at the security implications of allowing single-click installation of software distributed through multiple app markets on a single platform. The initial exploration of this problem has lead us to identify the main design requirements necessary to regain single-market security semantics in a multi-market environment. With these requirements in mind, we design and implement a security-enhancing installation architecture which we call Meteor. Specifically, we discuss the need for the following architectural components: universal application identifiers, crowdsourced app and developer information repositories and a decentralized kill-switch infrastructure. Finally, to demonstrate the technical feasibility of our proposal, we build an Android-specific proof of concept app which can be installed and used without buy-in from application markets or developers.

The remainder of this paper proceeds as follows. Section II describes background on app installation as well as current protections provided by smartphone platforms. Section III presents the threat model for the multiple market ecosystem. Section IV describes our proposed architecture. Section V covers the implementation of a proof-of-concept Android application. Section VI discusses the advantages, limitations and additional security and privacy considerations of the Meteor architecture. Section VII presents related work. We conclude in Section VIII.

II. Background

We first briefly discuss the evolution of smartphone app installation. Next, we discuss the role of digital signatures on smartphones. The section concludes with a review of current malware mitigation techniques on smartphone platforms.

Note that this and the following sections frequently discuss properties specific to the Android platform. We do this for two reasons. First, Android is the first platform to experience the negative effects of a multiple-market ecosystem. Second, focusing on a specific technology simplifies explanations. However, the architectural lessons are generally applicable across platforms.

A. Application Markets

The way users find and install applications on smartphones has changed drastically in recent years. Users of early smartphone platforms would typically download apps from a website, and then copy the downloaded file(s) to their device using a USB cable. In 2008, Apple introduced its App Store (along with the release of the iPhone 3G), which provided users with a new installation paradigm: an on-phone interface through which users could find, purchase, and install apps. The simplicity and ease of use of this installation model has led to over 10 billion downloads in only a few years [3]. This same use model has been adopted by others, including by Google on Android and by Apple and Microsoft on their desktop operating systems.

Each platform usually provides its own official application market. However, strict terms of service and questionable motivations for acceptance of an app in a specific market [12], [7] often motivate multiple markets. For example, the Cydia market has been created for "jailbroken" iPhones and the Amazon Appstore has been created to provide developers with better marketing capabilities. Multiple application markets present a different use model to the consumer. To use multiple markets, the user must currently download and install separate applications which serve as gateways into each market. This process may in fact negatively impact the security of the device (e.g., iPhones must be jailbroken to install Cydia; the Amazon Appstore requires users to disable the security feature that disallows applications from unknown sources). Next, managing apps becomes more difficult. Users may become confused when managing apps available in multiple markets. Finally, if a malicious app is identified, it is unclear which market has the authority, responsibility, and capability to employ a kill switch.

In a multiple-market environment, individual market vendors can compete by offering the same app at a lower price (e.g., the Amazon Appstore has a free "app of the day"). This allows consumers to comparison shop to find the cheapest version of an app. From an economics perspective, such competition is healthy. However, without a mechanism to determine if two or more apps are actually the same app, an adversary can lure users to install a malicious version of an app by distributing it into a market where the legitimate app does not exist. Attackers may also sell the app at a lower price to receive direct monetary revenue in addition to the return provided by the malware.

B. Package Signing and Application Namespace

Modern smartphone OSs require applications to be digitally signed [5]. On Android, app signatures are implemented as a form of *continuity signing* [26]: the OS verifies that subsequent application updates were signed by same developer as the original.

In order to sign an app, developers generate a public/private key pair and a self-signed certificate. As certificates are self-signed and never displayed to the user, nothing prevents the developer from inserting fake or incorrect information into the certificate. Once an app is ready for distribution, the developer uses the `jarsigner` [21] tool to sign the app and embed the signature into the application package. Android does not allow adding or removing certificates or keys during app updates [26], which forces developers to release a new application (under a new package name) in the event of losing their private signing key.

Android internally uses the *package name* (e.g., `com.company.app`) as an identifier for installed apps, and a new app is subjected to the continuity verification mentioned above if it is identified by the same package name as a currently installed app. If there is no currently installed app with same package name, the app is installed without additional checks. Continuity signing is verified for that package name from that point forward.

Package namespace collisions can and do occur. Occasionally, collisions result from careless developers who do not choose a sufficiently distinguished package name. However, package name collisions can also be a sign of an attack. For example, the *Geinimi* trojan [17] grafted SMS-sending malware on to the popular Monkey Jump 2.0 game. It was distributed through an alternative app market for Android and used the same package name (`com.dseffects.MonkeyJump2`) as the original game. As a side-effect, the name collision prevents a user from later installing the legitimate legitimate app without previously uninstalling the malicious version.

C. Malware Mitigation

Similar to commodity desktop platforms, antivirus software has emerged for smartphones. However, smartphone antivirus software does not operate analogous to desktop antivirus software due to differences in the execution environment. First, energy consumption prohibits routine scanning of files. Second, since smartphone apps run within a sandboxed environment (e.g., as in Android and iOS), antivirus apps do not have sufficient low level API hooks. Instead, smartphone antivirus programs frequently maintain blacklists containing identifiers for malicious applications. The resulting functionality strongly resembles that of kill switches.

The kill switches deployed by official markets are commonly integrated with the OS platform. If malware is detected to have been distributed through the market, the kill switch can remotely uninstall the app from phones. If the phone has network connectivity, the removal can occur nearly instantaneously. For example, Android maintains such a connection for Google services, including app installation and removal. Otherwise, removal occurs when connectivity is restored. An advantage of kill switches over antivirus software is the ability for a remote server to decide which installed apps to remove, as the market already maintains the list of installed apps. This eliminates the need for the phone to download and maintain blacklists, which conserves battery life.

We note that kill switches operate under the assumption that the malicious app was contained by a sandbox. In March 2011, Android malware exploited a local privilege escalation vulnerability [2], which required more attention than simply uninstalling the malicious app. As hinted above, Android's kill switch mechanism is implemented as part of a protocol that allows both removal and installation of apps. Therefore, the Google Android Market automatically installed a security fix to clean up devices known to have installed the malicious apps. In this case, Google effectively used Android's enhanced kill switch to distribute an OS patch. While kill switches clearly offer valuable functionality, it is less clear what trust consumers should place in each application market.

III. Threat model

The goal of our system is to help users minimize the risk of installing a malicious application. We consider "malicious" applications to be those that contain code to harm user devices, violate user privacy, or perform unwanted actions such as sending SMS without user approval. We classify malicious applications of interest to our architecture into three broad categories depending on the state of installed apps on the device, as well as the package name and signature of the malicious app.

1) **Colliding malicious app:** A malicious application with a package name matching that of another currently installed application. The application signature is different than the currently installed app, so Android would prevent the app from being installed unless the current version is removed by the user.
2) **Non-colliding malicious app:** An application with a package name that is different from any currently installed apps. This category of apps includes look-alikes e.g., Google Maps (`com.goggle.maps`); new apps with original functionality; and weaponized apps with trojaned components. The main characteristic of these apps is that their package names do not collide with the namespace of currently installed apps, therefore triggering no alerts (aside from the standard permission approval) to the user.
3) **Rogue update:** A formerly benign application is updated with a properly signed new version that includes malicious functionality. This can happen when a developer's signing key is compromised or the developer him/herself (or part of the development team) goes rogue.

Meteor can help detect malicious apps in categories 2 and 3. Our system need not detect applications in category 1, since they are blocked from installation by the OS (see Section II-B). We note that intentional user circumvention of this block (e.g., by uninstalling the current version) would effectively move the app to category 2.

Assumptions: We assume the adversary can successfully submit malicious apps to any or all application markets. We also assume the adversary is capable of creating duplicate apps that are identical to the original, with the exception of the digital signature. That is, we assume the adversary cannot easily access or independently re-create to the original developer's signing key.

IV. Proposal: Meteor

Meteor provides a flexible architecture for addressing the security concerns of a multiple market ecosystem. Addressing

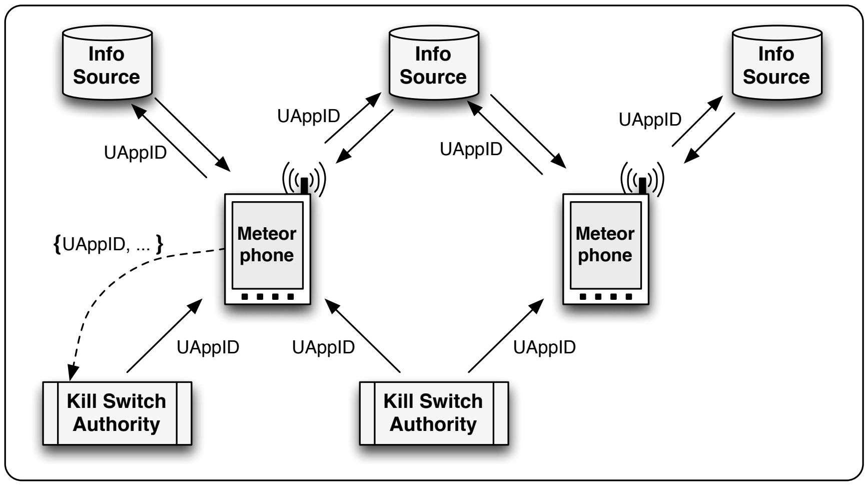


Fig. 1: Overview of the Meteor architecture. Each device running the Meteor client app (Meteorite) subscribes to a set of information sources and kill switch authorities. Information sources are queried (using UAppIDs) at install-time while kill switch authorities broadcast offending UAppIDs post-installation.

the app name consistency problem is a significant challenge. For example, simply forcing a fixed namespace based on DNS would be impractical to authenticate reliably, and it is unrealistic to expect consumers to separate safe from unsafe names. Instead, we leverage universally unique naming, and then aggregate statistics and expert ratings from multiple sources to enable a safer install-time environment.

Figure 1 overviews the Meteor architecture. At a high level, Meteor works as follows: after downloading an app (and prior to installation),[2] a pre-configured set of online information sources is queried to obtain additional information about the app or app developer. Each of the sources provides information which may help the user, or client-side app, gain confidence or reconsider the installation of the app. The device may also subscribe to one or more kill switch authorities, which enable the remote uninstallation of apps if they are found to be malicious.

Information sources queried by the Meteor smartphone app (Meteorite) may be grouped by areas of expertise. For example, one source might host information for apps related to social networking while a different source may focus on games. In fact it would be natural to expect multiple sources of each type, used by different users or user groups in different geographic regions. Narrowing the scope of coverage allows experts to use domain-specific knowledge to contribute higher quality information about apps.

The Meteor architecture is based upon the following four fundamental underlying concepts. The remainder of this section discusses these concepts in greater depth.

Universal application identifiers (UAppID): Meteor requires a reliable method of uniquely identifying app instances for reference by both information sources and kill switch authorities. To guarantee uniqueness, identifiers are cryptographically bound to the binary code of app instances, as and they incorporate the application name and digital signature data.

Developer registries: The first general type of information source is aimed at developers. Developer registries serve as repositories of developer submitted data such as contact information, news about app updates or issues, and other apps the developer authors. While a single developer registry is ideal from a security perspective, we expect region-specific developer registries to be more amenable to various global political climates.

Application databases: The second general type of information source targets applications. These sources host app information manually entered by experts in a particular domain (e.g., location-based shopping apps), or information automatically gathered from crowdsourced submissions. For example, databases may include expert reviews, ratings, permission descriptions, number of downloads, etc.

Kill switch authorities: In Meteor, trust in a remote party to uninstall malicious or inappropriate apps is decoupled from application markets, as markets have varying levels of trustworthiness. Depending on consumer privacy requirements, kill switch authorities may maintain a per-device list of installed apps

A. *Universal Application Identifiers*

The Amazon Appstore and the Google Market (and possibly others) use an app's package name to uniquely identify apps inside their stores. In a multi-market environment, using a package name alone will not uniquely identify apps because these names are sometimes arbitrarily chosen and may deliberately collide with another completely independent application (see Section III).

We propose the use of a universal application identifier (UAppID), to provide a common handle for referring to

[2]Optionally, additional checks could be deferred to the background and done after install-time, at the risk that malicious activity could occur in the meantime or alter security enforcement.

specific application instances. In Meteor, UAppIDs serve two main purposes: 1) *precise app lookups*, useful for comparing apps across multiple markets (similar to the way consumers search for the best price of a specific TV model as opposed to the best price of any TV); and 2) *instance-specific kill switches*, which allow for the removal of specific binaries instead of any app with a given package name.

Using a signed Android app as input, we construct its unique UAppID by extracting the package name and developer certificate (the two components used by Android to ensure correct application continuity as discussed in Section II-B) from the application package. Next we remove the application's signature[3] and certificate, and compute a cryptographic (collision-resistant) hash of the binary. Application names and versions are not explicitly incorporated because they are already embedded within the unsigned binary. Therefore:

$$\text{UAppID} = \{H(\text{package name, dev. cert}), H(\text{binary})\}$$

The UAppID is a tuple of size two. The first element is a hash of the package name and developer certificate[4]. The first element of the tuple can be used to identify all apps that Android considers to be equivalent. The second element is a hash of the application's binary. This portion of the UAppID can help identify repackaged (including pirated) apps across markets, or different version releases of the same app. The concatenation of both elements results in a globally unique string that can be used to identify a particular version of an application using a specific package name and written by a specific developer or organization.

This approach has the advantage that UAppIDs can be quickly reconstructed on the device, and provide strong guarantees that two identical UAppIDs refer to the same executable app if a strong hash function is used and verified. Of course, a universally agreed upon collision-resistant hash function must be designated and used (e.g., the Meteorite proof-of-concept app uses SHA256). UAppIDs can be further encoded to be more human-readable.

B. Developer Registries

One uncertainty users face when installing applications results from the lack of available information about the app's developer. Installation screens (when installing via an application market) typically show the developer's name and website, but these two pieces of information are provided by developers themselves, and are generally unverified by the market vendor. Furthermore, apps that are distributed outside of app markets (e.g., through a developer's website[5]) don't generally display *any* developer information at install-time.

We argue for the establishment of one or more central locations where developers can voluntarily disclose more information about their apps, development cycle, company, etc. These registries would be consulted to help resolve the *which John Smith?* problem, or simply to learn more about a developer before installing their software (similar to the way one might research a charity prior to donation). A widely used or relied upon central developer registry would ideally motivate developers to opt-in as a best-practice.

Developer registries can be used in several ways. First, application markets can obtain further confirmation of a developer's history and portfolio during sign-up. Second, application markets can show a "more info" section on an app information screen which hooks into the developer registry to show additional data about the developer. For example, *Registered developer since 04/2011*, *Developer of these 3 other apps*, *No apps killed to date*, *4 apps issued to date*, etc. Third, they provide a central point for a kill switch authority to look up developer contact information if necessary before deciding to throw a kill switch for their app. Finally, they provide a website that developers can use to make announcements about app issues or updates.

Our goals for such voluntary developer registries do not include becoming a certification authority for developers. These registries are intended to be lookup services for retrieving information that may help increase confidence in a developer. As such, the enrollment process is envisioned to include only minimal verification of submitted data (e.g., only verifying e-mail retrieval capabilities and control of a signing key), but fundamentally cannot provide assurance about the security or accuracy of provided information. We expect, however, that experts in each developer registry will help identify false or misleading information.

Participating in one or more developer registries should not be made mandatory as this would be both technically challenging and go against (e.g., Android's) open development philosophy. Indeed, if developers wish to remain anonymous, submit false information to the developer registry, or not register at all (e.g., an anti-censorship app developer afraid of political persecution) we envision that their apps would still be installable by those who choose to take that risk. However, we believe that in most cases, positive incentives and benefits would motivate developers to opt-in.

Enrollment: A developer creates a password-protected account on a registry by providing a valid e-mail address. After signing in, the developer asserts ownership of one or more certificates (i.e., app signing keys). The proof of control of the signing key is done using standard known techniques for challenge-response based on digital signatures [19, pp. 404-405], and could be designed to make use of current code-signing infrastructure and tools (see Section II-B).

C. Application Databases

Similar to the developer registries above, databases containing information about apps available both within and outside of markets would be valuable to end-users and experts. We see application databases as extensible repositories of application properties, statistics and reviews by subject area experts. For

[3]Removing the signature of a signed Android app can be done by deleting the META-INF directory inside the app package.

[4]Due to the two inputs to this hash function being developer-supplied, we also include the length of each string as input to the hash function to help prevent attacks that rely on the concatenation of variable-length inputs.

[5]Non-malicious developers might choose to distribute apps on their own to avoid paying registration fees or to keep 100% of the app's sale price.

TABLE I: Example types of information that may be provided by application databases.

Type of Information	Description
App binary properties	The name, version, package name and full developer certificates included in the binary.
App age and origin	How long an application has existed and a list of markets or sites on which it has been made available. This can help users determine if an app is brand new, or can be found elsewhere.
Expert reviews (or links to)	This service allows (ideally well-known experts or sets of) users to test software and submit technical or security reviews.
Blacklists and whitelists	The presence of an application (or other applications by the same developer) on a blacklist or whitelist. The reason(s) for being added to a list could also be recorded (e.g., privacy violations, tasteless content, malware, etc.).
Anti-virus and other security and privacy tests	Whether the app has been flagged by an anti-virus tool or has been reported for privacy violations.
Number of installs and uninstalls	Market provided data related to the total number of installations or number of active users of an app identified by a particular UAppID.

example, cartography enthusiasts can populate a database of apps that relate to maps, while gamers can maintain a database of game apps. We acknowledge that enthusiasts are not always security experts. However, they often have a vested interest in the security of apps in their area, so one might expect these enthusiasts to also communicate with security experts, or at least advanced users who report technical anomalies.

An application database contains a new entry for each known instance of an app. Each entry in the database lists information obtained from the package metadata (e.g., app name, app version) and the package name. The entries would also list the hash(es) of the developer certificate(s), as well as a hash of the relevant application binary and any other useful data for evaluating an app. Example types of information are listed in Table I.

As an example, Table II shows a list of 5 app instances that could have originated from a board-game app database. By viewing and comparing entries, experts managing the database could make the following observations (items with * trigger a warning to the user):

- **App versioning.** (Apps 1 and 2). These apps use the same package name and are signed with the same key. The version numbers and application hashes differ, as expected for a new release.
- **Multiple app developer.** (Apps 1, 2 and 4). A developer in possession of a certificate with hash starting with 0x74A8 is developing the Checkers and Chess apps.
- ***Certificate change.** (Apps 2 and 5). App hashes are the same indicating no code modification, but the certificates do not match. Possible explanations are that the developer lost access to the signing key, the developer is using different certificates for different distribution channels, or an attacker has stripped off the certificate from the binary replacing it with a new one. All these cases trigger further investigation, or notification of suspicious behavior to the user or an agent working on behalf of the user.
- ***Namespace collision.** (Apps 1 and 3). These apps share a package name and app name, but differ in certificate hash and application hash. This could mean that a malicious developer has grafted malware on to to the legitimate chess application, or that by chance, 2 developers have chosen to use the same package and app name, as well as coincided in version number.

In the event of certificate change and namespace collisions, the application database itself cannot answer the question *which of the conflicting apps should I trust?* The information merely suggests the presence of malicious activity, and a supplementary mechanism is necessary to decide which of the apps should be downloaded. For example, the database might also contain initial submission times or overall number of submissions. The database could also include expert reviews, or link to the appropriate developer registry.

Populating Application Databases: Databases should contain relatively up-to-date information to be effective in helping users make informed decisions. The database could be populated in several ways: (1) by application market vendors relaying information about apps they have accepted into their respective markets; (2) by paid employees who look for apps and submit them to a database; (3) by crowdsourced submissions from volunteers and interested users.

The existence of an app in a database reflects the notion that the app is known or has been seen. The absence of an app in a database could also be leveraged to identify obscure or "fly-by-night" apps. The devices of conservative users may be configured to only allow installation of well-known apps for which no suspicious behaviour has been reported (see Figure 4a).

Transparency in Consulting a Database: It is unreasonable to expect typical end-users to actively inspect all information provided for an app. Detailed information retrieval is an option that can be enabled by experts, but ordinary end-users (or agents working on their behalf) would only be alerted in the event of suspicious behaviour (e.g., certificate change, namespace collision, blacklisted app, etc.). Deployed this way, the application database and the client smartphone app (Meteorite) would be transparent to users the vast majority of the time as the most common scenario for new app entries in the database would likely be new app versions. Mobile security vendors, security researchers and other automated tools may actively consult the database to find app inconsistencies and overall app statistics, and to compile aggregate statistics.

TABLE II: Example application database. Hashes truncated for space.

ID	Name	Version	Package Name	Cert. Hash	App Hash	First Seen	Submissions
1	Chess	1.5.2	`com.games.chess`	0x74A8...	0x93B1...	Jan 2, 2012	100
2	Chess	1.6	`com.games.chess`	0x74A8...	0x8F2C...	Jan 30, 2012	80
3	Chess	1.5.2	`com.games.chess`	0x1D51...	0xA33C...	Feb 21, 2012	3
4	Checkers	0.5-beta	`com.games.checkers`	0x74A8...	0xDB89...	Mar 1, 2012	10
5	Chess	1.6	`com.games.chess`	0xF307...	0x8F2C...	Mar 3, 2012	1

D. *Kill Switch Authorities*

The final component in the Meteor architecture is a kill switch authority responsible for the remote removal of apps on user devices. Kill switch authorities have the unique ability to remove an app from subscribed devices independently of the market or website from which they were obtained. This is different to the way kill switches are handled in the current smartphone ecosystem, where each market is responsible for removing apps that it distributed.

Kill switch authorities have the capacity to securely broadcast offending UAppIDs to registered devices at any given time (i.e., *push*). This is in contrast to developer registries and application databases which serve information based on a query (i.e, *pull*). Optionally, devices may transmit lists of installed apps to kill switch authorities (as denoted by the dotted lines in Figures 1 and 5) such that kill switch signals are only received for installed apps. Informing a kill switch authority of installed apps has significant resource consumption advantages for mobile devices, as the authority can perform management logic and only contact the device when necessary; this however brings with it potential privacy issues.

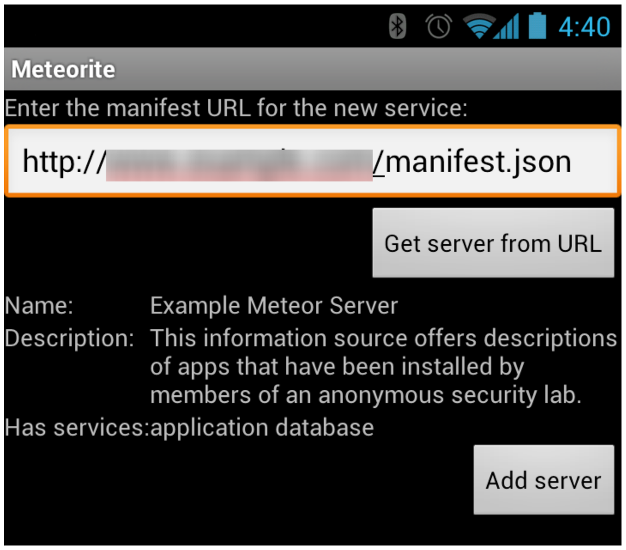
Similar to application databases and developer registries, we expect kill switch authorities to specialize on a small set of application domains. For example, a kill switch authority could concentrate on identifying and disabling apps which are unsuitable for children younger than a certain age.

V. Implementation

We have implemented a Meteor client app (called Meteorite) for Android along with the corresponding server-side components to test the technical feasibility of our proposal. Meteorite can be installed on any device running Android 2.2 or greater (which covers approximately 93% of the worldwide Android install-base as of April 2012 [14]. It has three main components as discussed in the following subsections, and requires neither modifications to the underlying OS nor root access. Section V-D discusses the advantages and limitations of operating within unprivileged app boundaries.

A. *Information source management*

When the Meteorite app is first installed, the user adds new information sources specifying the type (application database, developer registry or kill switch authority) as well as a URL to query. To minimize the amount of data to be manually entered, servers (regardless of their type) publish a JSON-formatted manifest file that lists services offered and additional server information (see Figure 3). A screenshot of the information source entry process is displayed in Figure 2.


4:40

Meteorite

Enter the manifest URL for the new service:

http://[redacted]/manifest.json

Get server from URL

Name: Example Meteor Server

Description: This information source offers descriptions of apps that have been installed by members of an anonymous security lab.

Has services:application database

Add server

Fig. 2: Adding an information source to the Meteorite app

```
manifest.json
{ "name" : "Example Meteor Server",
  "description" : "This information source
                   offers descriptions of apps
                   that have been installed by
                   members of a security lab",
  "services" :
  [ { "type" : "application database",
      "url" : "http://XXXXXXX/query.php",
      "license" : "no" } ] }
```

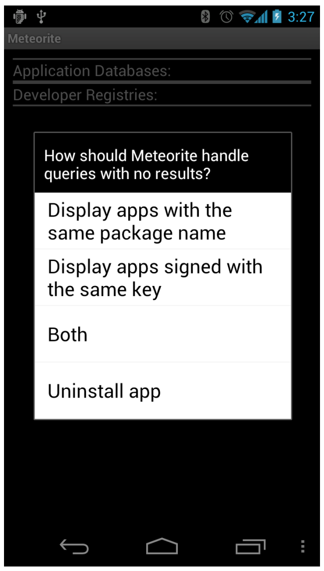
Fig. 3: The server manifest corresponding to the example in Figure 2.

From a practical standpoint, popular or well-known information sources could be preconfigured within the app. We expect users to find other candidate servers via colleagues and expert websites that recommend good sources for given tasks and report poor quality or malicious sources.

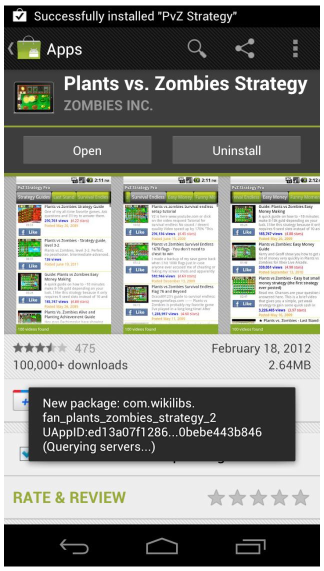
The ideal number of configured servers will vary according to security, usability, and cost requirements of each user. Expert users who want as much information as possible with full control over their installed apps may choose to configure a large number of information sources and no kill switch authorities. Non-experts and users who don't want to be involved in low-level technical details of making security decisions may consider installing more kill switch authorities.

B. *Information source query*

Once information source(s) have been added, the Meteorite app will receive a *Broadcast Intent* every time a new package is installed or updated.[6] Upon receiving the intent, Meteorite computes the newly installed app's UAppID, and issues a standard HTTP POST request to all enabled application databases and developer registries (see item 2 in Figure 5). Information sources currently send back information related to the submitted UAppID if it exists. Depending on how Meteorite has been configured, the retrieved app information can be displayed to the user, who is then prompted to continue or to uninstall the app.


3:27
Meteorite
Application Databases:
Developer Registries:
How should Meteorite handle queries with no results?
Display apps with the same package name
Display apps signed with the same key
Both
Uninstall app

(a) Configuration


Successfully installed "PvZ Strategy"
Apps
Plants vs. Zombies Strategy
ZOMBIES INC.
Open
Uninstall
475
February 18, 2012
100,000+ downloads
2.64MB
New package: com.wikilibs.fan_plants_zombies_strategy_2
UAppID:ed13a07f1286...0bebe443b846
(Querying servers...)
RATE & REVIEW

(b) New app installed notification

Fig. 4: Screenshots of the Meteorite app. On the left, the settings window. On the right, a notification that a new app has been installed and a query is in progress in the background.

C. *Kill Switch Listener*

The final component of the Meteorite app is the kill switch listener (depicted as item 4 in Figure 5). We avoid polling (which is expensive in terms of battery and network performance) kill switch authorities by using Google's Cloud to Device Messaging (C2DM) Framework[7] for push notifications to devices. C2DM allows a server to send short data messages (up to 1024 bytes) to registered devices. The messages are sent from the kill switch authority to Google, and then relayed to the Meteorite app using the existing connection most Android devices maintain with Google servers.

[6]These intents are received regardless of the app's installation origin (e.g., sideloaded, third party app market, official market).

[7]https://code.google.com/android/c2dm/index.html

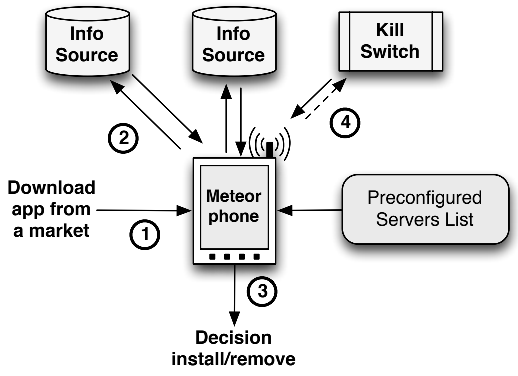


Fig. 5: Overview of the Meteor app installation process.

When adding each kill switch authority to the Meteorite app, a registration process must take place to inform the kill switch authority that the user is willing to receive kill notifications. The user can remove or temporarily disable kill switch authorities from within the app, as well as choose to ignore kill switch messages when they are received.

Our demonstration kill switch authority currently sends the UAppID of the app to be killed, along with a reason the kill switch was thrown. Receiving a kill switch launches Meteorite in the background and compares the received UAppID against a list of UAppIDs for all installed apps. If a match is found, the user's current Android activity is interrupted to display the kill switch data. The user is then prompted to either uninstall the app or dismiss the message.

As noted earlier, Meteorite does not have root privileges so it is not capable of removing other apps without the user's consent. Meteorite only invokes Android's package manager and the user must confirm or deny the final uninstall process.

D. *Implementation Notes*

The Meteorite app leverages Android's application lifecycle and IPC to avoid running in the background and constant server polling. The OS only invokes Meteorite at app installation time or upon receiving a kill switch, meaning that there is negligible impact on battery life or overall performance.

Our choice to use non-privileged APIs comes with the trade-off of creating a vulnerability window between app installation and first launch. Malicious apps could register hooks to launch automatically (e.g., after a device boots, after receiving an SMS, or as a handler for specific file types). This window could be avoided if the Android app installer were modified to perform Meteor queries prior to installation, but would require installing a full new Android firmware which may discourage widespread adoption.

Source code for the Meteor app as well as UAppID utilities are available for download at http://www.ccsl.carleton.ca/software/meteor.

VI. Discussion

Here we discuss the advantages and limitations of the Meteor architecture as a whole, and review the specific benefits of grouping information sources by domain and security focus.

A. Advantages

Incrementally deployable and extensible: The proposed architecture provides security benefits even if using as little as one information source or kill switch authority. The client-side software could initially be distributed as an app download, and later possibly bundled as a core OS feature. The ability to customize information sources offers an extensible solution that can be adapted to a wide range of users/skill levels, and facilitates the existence of a democratic application marketplace.

Scalability: By allowing experts in small domains to handle application reviews and database maintenance, high quality information may be produced and distributed quickly.

Low resource requirement: Servers are only queried upon app installation, where a small amount of application-specific data is sent and received. Meteor involves no massive (e.g. antivirus) signature downloads and no CPU-heavy operations performed on the device aside from hashing the app in question.

Only intrusive on suspicious behaviour: The Meteorite app can be configured to only display warning messages in the event of detecting suspicious activity (by the developer or app). If no malicious activity has been reported for the app being installed, the installation process is not visibly different to the end-user than the current app installation procedures.

B. Limitations

Usability: Meteorite requires some level of user input either for adding servers or interpreting the results of the information source query. While we can attempt to automate some of these tasks (e.g., by specifying policies), our proposal is not a one-size-fits-all solution. Future work is needed to identify usability challenges when deploying a system like this.

Quality of information sources: There is no mechanism to automatically handle malicious information sources, which may allow attackers to craft a complex attack where a user is tricked into installing a malicious information source as well as a malicious app. In this example, the information source could return positive comments and ratings for all apps, or at least those an attacker wishes to be installed, giving the user a false sense of trust that the app being installed is benign (false negatives). The information source could also attempt a denial of service by replying with warnings on all queries, creating false positives which must be identified and resolved by the user. While attacks like these are less likely as more information sources are selected (assuming at least one has identified the malicious app), they may still be possible for users with small information server lists.

We believe that over time, individuals and sub-communities will place their trust in information sources that deserve such trust. While Meteor and other (e.g., single-market ecosystems with state-of-the-art vetting) systems fundamentally cannot preclude abuse of trust, Meteor can deal with abuse in the longer run by user choice.

C. Security and Privacy Considerations

Separating content creators from content hosts: Meteor does not require that databases be hosted by the same individuals who create the content within them. For various reasons (e.g., high cost or unavailable expertise), expert users may not want to operate a dedicated server even though they have access to app or developer information. Similarly, end-users may trust an entity with the app information they create, but may not be willing to reveal to that entity the apps they are installing. By allowing this separation, database hosts can act as relays or proxies to help preserve end-user privacy. Meteor could be extended to allow private information retrieval (PIR [9]) for users with high privacy requirements.

Authenticating information: In its most basic form, Meteorite clients communicate with Meteor servers via HTTPS using Android's default root CA list. Future work is needed to design a proper way to handle man in the middle attacks on the local network (e.g., an attacker intercepting both the app file download and the Meteor query). We are exploring the option of using signed database records that can be authenticated locally, similar to the approach of Samuel et al. [25].

Lack of consensus in database entries: In a distributed system like Meteor, it is possible that users subscribing to many information sources will receive conflicting information. For example, an app may exist on a whitelist for one reason and simultaneously be listed on a blacklist for a completely different reason. The Meteorite app currently requires consensus among information sources that are queried, but future work will look at ways to automatically resolve conflicts based on pre-defined policies.

VII. Related Work

Meteor builds upon known proposals for unique file identification, cross-checking information, cloud-based malware detection and information crowdsourcing. This section reviews some of these proposals.

Kim and Spafford propose Tripwire [15], a tool for recording hashes of important system files and later detecting modifications or intrusions. UAppIDs are similar in that they involve cryptographic hashes of the app's binary code, but these hashes are not stored to detect modification (code signing already does this). UAppIDs uniquely identify and index (e.g., for app lookup) app instances.

Oberheide et al. [22], [23] highlight advantages of offloading malware detection to the cloud such as low resource requirements and better detection coverage. The authors run multiple antivirus engines on each submitted binary, and hashes of binaries are stored to improve performance (i.e., avoid scanning the same file if the result is already known). This is conceptually different from Meteor, which aggregates information from multiple expert sources on the device rather than aggregating multiple services on a central server.

A number of researchers have analyzed large collections of apps across multiple markets. The results of these experiments, including detection of malware [28], privacy leaks [11], and poor developer practices [13] would be good candidates for inclusion in a Meteor application database.

Perspectives [27] (and related projects such as Convergence [20]) uses a set of "notary hosts" which monitor web servers' public keys from multiple vantage points on the Internet. Clients query the notaries to detect man-in-the-middle-attacks or changes to public keys. Information sources in Meteor play a role somewhat similar to Perspectives' notaries, but app information is collected by more than passive monitoring (e.g., experts actively trying apps and submitting reviews).

Meteor shares similarities with browser-based ad blocking tools such as Adblock Plus [1], which allow users to subscribe to ad blocking lists maintained by experts around the web. Similar to Meteor information sources, each ad blocking list filters specific types of advertisements (e.g., region-specific, content-specific), allowing users to build a custom filter set tailored to their needs.

Aggregate and personalized ratings from users in a social circle can be helpful to find inappropriate apps, as users in the same social circle tend to have similar definitions of appropriateness [8]. However, detecting malicious applications generally requires experts, who may not be present in all social circles. Meteor attempts to create domain-specific services that can individually crowdsource information. However, Meteor does not specify how to deal with the problem of expert recruiting or "fame management" [10]. We defer this aspect to each information source.

VIII. Conclusion

In this paper, we have drawn attention to new security concerns introduced by multi-market environments and single-click software installation. We have proposed Meteor as a scalable security-enhancing software installation architecture designed to bridge the gap between non-cooperating application markets. Each component in Meteor (i.e., UAppIDs, developer registries, application databases and kill switch authorities) plays a key role in providing security semantics similar to those achieved in single-market environments while retaining the benefits of multi-market environments. Future work includes exploring usability issues of Meteor and developing a policy language to further minimize novice user involvement.

Acknowledgements

We thank Michelle Burrows and Michael Doherty for their contributions to the Meteor app. We also thank Patrick Traynor for his valuable feedback. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)—the first author through a Canada Graduate Scholarship; and the third author through a Discovery Grant and as Canada Research Chair in Authentication and Computer Security. We also acknowledge support from NSERC ISSNet.

References

[1] Adblock Plus. https://adblockplus.org, June 2011.

[2] Android Market. March 2011 Security Issue. http://googlemobile.blogspot.ca/2011/03/update-on-android-market-security.html, Mar. 2011.

[3] Apple Inc. Apple's App Store Downloads Top 10 Billion. http://www.apple.com/pr/library/2011/01/22appstore.html, Jan. 2011.

[4] D. Barrera, W. Enck, and P. van Oorschot. Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. Technical Report TR-11-06, Carleton University, School of Computer Science, Apr 2011.

[5] D. Barrera and P. Van Oorschot. Secure software installation on smartphones. *IEEE Security and Privacy*, 9(3):42–48, 2011.

[6] R. Cannings. Exercising Our Remote Application Removal Feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html, June 2010.

[7] B. X. Chen. Want Porn? Buy an Android Phone, Steve Jobs Says. Wired Gadget Lab, Apr. 2010. http://www.wired.com/gadgetlab/2010/04/steve-jobs-porn/.

[8] P. Chia, A. Heiner, and N. Asokan. Use of Ratings from Personalized Communities for Trustworthy App. Installation. In *Proceedings of the 15th Nordic Conference in Secure IT Systems (Nordsec)*, Oct 2010.

[9] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS '95)*, pages 41–50, 1995.

[10] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing systems on the World-Wide Web. *Communications of the ACM*, 54(4):86–96, 2011.

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

[12] Federal Communications Commission. Letter to Apple regarding Google Voice and related iPhone applications. DA 09-1736, July 2009. http://hraunfoss.fcc.gov/edocs_public/attachmatch/DA-09-1736A1.pdf.

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

[14] Google. Platform Versions - Android Developers. http://developer.android.com/resources/dashboard/platform-versions.html, Feb. 2012.

[15] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM CCS*, 1994.

[16] H. Lockheimer. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, Feb. 2012.

[17] Lookout. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. http://blog.mylookout.com/2010/12/geinimi_trojan/, Dec. 2010.

[18] P. McDaniel and W. Enck. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine*, 8(5):76–78, September/October 2010.

[19] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC, 1997.

[20] Moxie Marlinspike. Convergence (Beta). http://convergence.io/index.html, Aug. 2011.

[21] S. Oaks. *Java Security. Chapter 12. Digital signatures*. O'Reilly Media, 2001.

[22] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: Executable analysis in the network cloud. In *USENIX HotSec*, 2007.

[23] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *USENIX Security*, 2008.

[24] S. Perez. Smartphones Outsell PCs. The New York Times, Feb. 2011. http://www.nytimes.com/external/readwriteweb/2011/02/08/08readwriteweb-smartphones-outsell-pcs-74275.html.

[25] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *ACM CCS*, pages 61–72, 2010.

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

[27] D. Wendlandt, D. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, 2008.

[28] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS*, 2012.