Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems*

David Barrera[1] William Enck[2] Paul C. van Oorschot[1]

[1]School of Computer Science, Carleton University, Canada
[2]Penn State University, University Park, PA

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 (integrity). We outline a novel approach to retaining single-market security semantics while enabling the flexibility and independence of a multi-market environment. We propose *Stratus* as an abstract, security-enhancing, application installation model that leverages information from a configurable set of security information sources. Information source content ranges from simple statistics to expert ratings for a specific application domain. The Stratus approach provides valuable decision-making criteria useful not only for smartphone users, but technology consumers as a whole, as new and existing computing environments, including for desktop software, converge on a market-like model for software installation.

1. INTRODUCTION

Consumer computing as we know it is currently undergoing a transition. The emerging software ecosystem frequently commoditizes functionality into discrete, purpose-driven applications. These applications, colloquially known as "apps," consist of both stand-alone utilities and front-ends to Internet-based services. 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. In the smartphone environment, the purpose-driven app characteristic is a result of *a*) limited user interfaces, *b*) a cottage industry of hundreds of thousands of developers attempting to be the first to market, and *c*) a new computing environment that enables innovative functionality (e.g., location sensitivity). However, it is likely that general computing will follow suit. Smartphone apps share much in common with Web apps running within a browser, and recent market statistics report more smartphones are sold per month than PCs [19], indicating a new breed of users whose only exposure to computing is the app-centric model.

Application markets, also known as "app stores," provide *one-click software installation*. Markets serve as a central point of app distribution, sales, discovery, and installation. They have become hugely successful on smartphone platforms: they are the *de facto* method of installing apps. Much of the success of application markets can be attributed to their ease of use for both developers and consumers. On the developer front, markets simplify app distribution and sales; on the consumer front, on-phone market interfaces allow wireless app discovery and installation.

A single, central application market offers an opportunity to improve consumer security. Thus far, this model has exhibited two clear advantages. First, it provides a means to remotely uninstall distributed apps later identified as malicious. On multiple occasions, Google has exercised so called "kill switch" functionality for its Android Market [4, 12]. Kill switches provide faster clean-up than traditional antivirus software, as they push actions to devices and do not require definition updates or resource intensive scanning. Second, search results present consistent developer names for applications.[1] 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 market to apply sanity checks for well known, high-impact entities, e.g., "Bank of America." As an aside, we note that while some believe security vetting is a valuable attribute of a central market, the practical effectiveness and scalability of certifying apps against security requirements is uncertain at best [14].

*Version: April 22, 2011.
Contact author: dbarrera@ccsl.carleton.ca

[1]When corporations commission developers to create smartphone applications the corporation name may be listed rather than the developer name.

For several smartphone platforms, *a multiple-market ecosystem* is emerging. A multiple-market environment is 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 code [5]. Even in an environment with minimal restrictions such as the Android Market, multiple markets are emerging. For example, 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 consider the convergence of one-click installation and multiple software distributors. Specifically, we seek to answer two questions: 1) *What security is sacrificed by moving to a multiple-market ecosystem?* 2) *How do we achieve 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 diverse characteristics, ranging from app age, frequency of updates, and number of downloads to expert ratings on the app and other apps by the developer. On the other hand, kill switch authorities allow consumer devices to be configured to subscribe to notifications of dangerous apps for removal. Such a decentralized architecture is valuable not only for smartphones, but also tablets, netbooks, and computing devices in general as software distribution and installation inevitably converges in the direction of application markets; for example, Apple has already introduced an App Store for Mac OS X.

Our security-enhanced installation architecture, *Stratus*, is based upon four fundamental underlying concepts:

- **Universal application identifiers**: We propose the use of *UAppID*s that incorporate not only the application and developer names, but also the cryptographic hash of the application and related signing key information. UAppIDs enable security information sources and kill switch authorities to reliably identify app instances.
- **Developer registries**: The first general type of security information source focuses on developers. Developer registries allow name consistency across markets and potentially include expert ratings regarding published applications. 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 security information source focuses on applications. The information provided by application databases is expected to be diverse, and will range by database. For example, one database may provide factual app age and number of downloads, while another may provide crowd-sourced expert security ratings for a narrow domain, e.g., location-based shopping apps.
- **Kill switch authorities**: Whereas developer registries and application databases are polled, kill switch authorities push information to devices. Depending on consumer privacy requirements, the kill switch authority may maintain a per-device list of installed apps to more efficiently manage kill switches for target devices.

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. Note that some goals sought by the creators of public key infrastructures (PKIs) often overlap with this goal, and numerous specific instances of PKIs have failed to meet the needs of users [8]. However, the characteristics of app distribution potentially make this goal more tractable, and we seek to centrally facilitate a distributed system, as opposed to the centralized control of typical PKI systems. Here, our key observations are that 1) name collisions should be minimal, i.e., given sufficient precision when identifying developers and applications, unintentional name collisions will be rare; 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.

We believe our proposed architecture for multiple-market app distribution is a valuable step moving forward. However, we believe our main contribution will end up being stimulation of alternate solutions meeting many of the requirements we have identified, including several elements of our proposed solution, rather than one specific solution proposal itself. Indeed, many design details are needed to practically instantiate such a model. We realize that a handful of university researchers alone are not in a position to successfully deploy such a system and hope that this paper is viewed as a call to arms motivating mobile platform manufacturers, and others who may gain from a robust global software distribution infrastructure.

The remainder of this paper proceeds as follows. Section 2 presents the threat model for the multiple market ecosystem. Section 3 describes existing protections provided by smartphone platforms. Section 4 describes our proposed architecture. Section 5 presents an example use case and discusses the advantages and limitations of our architecture. Section 6 presents related work. Section 7 concludes.

2. SECURITY CONCERNS IN MULTIPLE MARKET ECOSYSTEMS

Application markets are the primary method of delivering apps onto smartphones. This section begins by discussing the evolution of smartphone app installation. We then specify the threat model for the specific issues we wish to address.

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.

2.1 Smartphone Application Installation

A smartphone is classically defined as a mobile phone that allows the user to download and run third-party applications from the Internet. Finding and installing applications proved to be a major hurdle for users of early smartphone platforms such as Symbian OS, RIM BlackBerry OS, and Microsoft Windows Mobile, which required manual app installation: the user uses a PC Web browser and a search engine or app aggregation website to find and download an app, then must connect a USB cable between the PC and the phone to install the application.

Apple's release of the iPhone 3G and the App Store in 2008 triggered a surge in smartphone popularity. While 3G network speeds and hardware features such as GPS and accelerometers were positive catalysts, the App Store's installation model provided the user with an on-phone interface to find, purchase, and install apps. The simplicity and ease of use of this one-click 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 Google's Android Market.

Each smartphone platform provides its own official application market. An app's entry into a market depends on criteria set by the market owner. For example, Apple's App Store requires developers to pay a US$99 annual fee with apps subject to approval by Apple (based on non-public criteria). In contrast, Google's Android Market requires a one-time US$25 developer fee and no approval process. While a single market model is easier for users and may arguably provide more security, social and economic pressures motivate multiple markets. Apple is frequently criticized for questionable motivations when removing or rejecting apps [9, 5]. As one response, the Cydia market has been created for "jailbroken" iPhones. However even relatively open markets such as Google's Android Market face pressures motivating multiple markets. For example, Amazon's Appstore for Android provides sales on apps, and claims better protections against unauthorized app copying and distribution as part of the rationale for its US$99 annual developer fee [1].

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.

2.2 Threat Model

In a multiple-market environment, the same app may appear in more than one market. While this may complicate software management for the user, it also introduces an attack vector for an adversary to trick the user into installing a malicious, but identically named app. In light of this attack vector, we classify apps into one of four categories:

1. **Non-malicious and original**: This class characterizes most apps in the market. The app was written without malicious intent and provides a unique value-add to the user. It may provide similar functionality to other apps, but its uniqueness is apparent.
2. **Non-malicious and duplicate**: This class characterizes apps that piggyback on another app's success, but have no intent to harm the user. Duplication may be as trivial as copying the original app's name or icon, but also includes stealing an app's code and posting it on a different market (e.g., piracy).
3. **Malicious and original**: This class characterizes Trojans that do not duplicate another app's functionality or appearance. Such apps may frequently offer too-good-to-be-true features to lure users (e.g., free international phone calls).
4. **Malicious and duplicate**: This class characterizes Trojans that trick the user into into installing a malicious version of a popular legitimate app. Often, the user will not be able to detect the differences between original and malicious versions.

In this paper, we concentrate on identifying and preventing the installation of malicious apps (types 3 and 4), but our approach can also help detect some types of non-malicious duplicates (type 2 apps); see the example in Section 4.3.

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 does not have access to the original developer's signing key.

There are two trivially exploitable attack scenarios introduced by the multiple-market environment that we specifically seek to address. We refer to these scenarios as *namespace collisions* and *bargain shoppers*.

Namespace Collisions: In the multiple-market environment, a namespace collision occurs when two or more independent parties or applications use the same package name (either intentionally or by accident). On an Android device, two apps with the same package name cannot be installed simultaneously. This creates a race condition where the first app installed on to a device "claims" that namespace, preventing another (possibly legitimate) app with the same package name from being installed. While this can occur accidentally if developers do not choose a sufficiently distinguished package name (e.g., `com.games.chess`), deliberate attacks are possible. For example, the *Geinimi* trojan [13], distributed through a third-party application market for Android, grafted SMS-sending malware on to the

popular Monkey Jump 2.0 game. It also used the same package name (`com.dseffects.MonkeyJump2`) as the original game, preventing a user from later installing the real app without previously uninstalling the malicious version.

Bargain Shoppers: 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, from a security perspective, it introduces a simple attack vector: without a mechanism to determine if two or more apps are actually the same app, an adversary can graft malware onto an existing app and distribute it into a market where the existing app does not exist. Clever attackers may also sell the app at a lower price to receive direct monetary revenue in addition to the return provided by the Trojan.

3. EXISTING PROTECTIONS

Existing security mechanisms can help mitigate threats in a multiple-market environment. We now discuss package signing and existing malware mitigation. As previously noted, we use the Android platform to drive the narrative, but lessons are more broadly applicable.

3.1 Package Signing

All Android applications must be digitally signed to run on user devices [10]. App signatures in Android are a form of *continuity signing* [20]: the OS verifies that subsequent application updates were signed by same developer as the original. Android 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 if it is identified by the same package name as a currently installed app. Note that on Android, app signatures also allow two applications signed by the same developer key to share resources; however, this functionality is beyond our current domain of interest.

Before signing an app, the developer generates 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` [16] tool to sign it and embed the certificate(s) and public key(s) into the application package. Google recommends a certificate validity period of 25 years or more, and the Android Market automatically rejects apps with certificates expiring before October 22, 2033 [10]. In Android (currently version 2.3), certificate validity is only checked at install-time. Therefore, as long as the app is installed before the certificate expiration date, it can be used indefinitely. Additionally, Android does not allow adding or removing certificates or keys during app updates [20], which forces developers to release a new application (under a new package name) in the event of losing their private signing key.

Current signing practices: To explore how developers are using the signing architecture, we analyzed 8800 apps from the Android Market. The dataset consists of monthly snapshots of the top 1100 free apps (top 50 in each of the 22 app categories) downloaded each month between April and November 2010. Developers on average signed 1.52 apps with the same certificate (see Table 1), and one developer (a book publisher) signed as many as 42 different apps, each being a different book. Google for example has signed 12 of their apps (e.g., Youtube, Goggles, Voice, Finance) with one certificate, and 5 others (e.g., Maps, Voice Search) with a different certificate.

Table 1: Application and developer certificate stats

Snapshot (2010)	Unique Certs	Apps/Cert
April	788	1.39
May	801	1.37
June	795	1.38
July	791	1.39
August	811	1.35
September	829	1.32
October	845	1.30
November	842	1.30
Overall (8800 apps)	1593	1.52 (2426 uniq.)

We also compared several free apps from the Android Market with versions in the Amazon Appstore. We found that in most cases, the same certificate was used to sign the app on both markets. While developers have the opportunity to create a new market-specific certificate, doing so does not allow users to seamlessly transition between markets. This provides an incentive for developers to use one certificate for all their apps, set an expiration date well into the future, and use the same certificate in multiple markets.

3.2 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 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. Kill switches are a conceptually valuable tool. 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 energy.

We note that kill switches operate under the assumption that the malicious app was contained by a sandbox. Re-

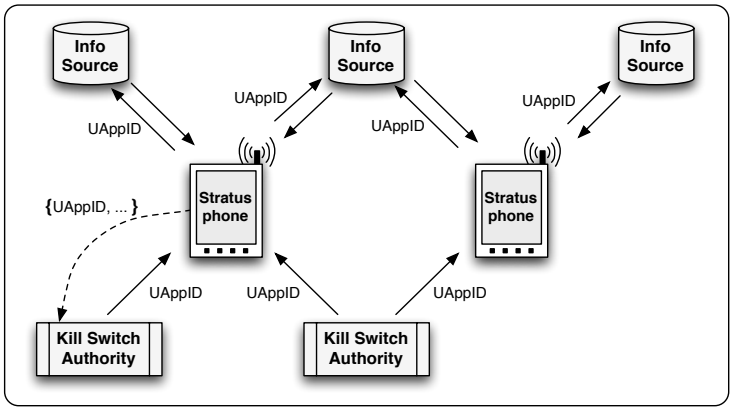

Figure 1: Stratus overview. Each Stratus-enabled device subscribes to a set of information sources and kill switch authorities. Information sources are queried (using UAppIDs) at install-time while kill switch authorities broadcast offending UAppIDs post-installation.

cently, Android malware exploited a local privilege escalation vulnerability, which requires 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 [2]. Note that 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. For example, should consumers allow the MiKandi adult app market to remotely install and uninstall applications?

4. MULTI-SOURCED INSTALL-TIME INFORMATION (STRATUS)

Stratus provides a flexible architecture for addressing the security concerns of a multiple market ecosystem. Addressing the app name consistency problem is a significant challenge. Simply forcing a fixed namespace based on DNS is insufficient. Such an approach is impractical to authenticate reliably, and it is unrealistic to expect consumers to separate safe from unsafe names. Instead, we only require universally unique naming, and then aggregate statistics and expert ratings from multiple sources to enable a safer install-time environment. Note that in doing so, the Stratus design can also benefit single-market environments.

Figure 1 overviews the Stratus architecture. At a high level, Stratus works as follows: after downloading an app (and prior to installation),[2] a pre-configured set of information sources is queried to obtain any additional information about the app or app developer. Each of the sources provides information which may help the user, or client-side middleware to be determined, gain confidence or reconsider the installation of the app. Note that access to sources may be free or provided at a cost. 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.

Security information sources 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 physics-based 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.

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

Universal application identifiers: The Status architecture requires a reliable method of uniquely identifying app instances for reference by both information sources and kill switch authorities. Identifiers must be cryptographically bound to the binary code of app instances.

Developer registries: The first general type of information source targets developers. These sources host developer submitted data such as contact information, news about app updates or issues, and other apps the developer authors.

Application databases: The second general type of information source targets applications. These sources host app information submitted by experts. For example, expert reviews, ratings, permission descriptions, number of downloads, etc.

Kill switch authorities: Trust in a remote party to uninstall malicious or inappropriate apps is decoupled from application markets, as markets have varying levels of trustworthiness.

4.1 Universal Application Identifiers

The Amazon Appstore and the Google Market (and possibly others) use an app's package name to uniquely identify apps

[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.

inside their stores. In a multi-market environment, using a package name alone will not uniquely identify apps because these names are arbitrarily chosen and may deliberately collide with another completely independent application (see Section 2.2).

We propose the use of a universal application identifier (UAppID), to provide a common handle for referring to specific application instances. In Stratus, 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 UAppID by extracting the package name and developer certificate (the two components used by Android to ensure correct application continuity as discussed in Section 3.1) from the application package. Next we obtain a cryptographic (collision-resistant) hash of the *unsigned* binary,[3] as the signed binary includes the X.509 developer certificate. 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 the concatenation ("::") of two components. The left part is a hash of the package name and developer certificate. The right hand portion is a hash of the application's code. The concatenation 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 app if a strong hash function is used and verified. Of course, a universally agreed upon strong hash function must be designated and used. UAppIDs can be further encoded to be more human-readable.

4.2 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 generally unverified by the market vendor. Furthermore, apps that are distributed outside of app markets (e.g., through a developer's website[4]) don't generally display *any* developer information at install-time.

[3]Removing the signature of a signed Android app can be done by running `zip -d example.apk META-INF*`.

[4]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.

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 making a final decision 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 make claims 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., signing keys). The proof of control of the signing key is done using standard known techniques for challenge-response based on digital signatures [15, pp. 404-405], and could be designed to make use of current code-signing infrastructure and tools (see Section 3.1).

Hosting registries: Registries would ideally be provided free of direct charge for both developers and users retrieving information. Candidate host providers for this type of service might be universities, non-profits, or commercial organizations funded by advertisement revenue.

Table 2: 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.).
Popularity ratings	Numeric or "star" ratings for an app in different (usually subjective) categories (e.g., usefulness, value, etc.).
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.

4.3 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 example, cartography enthusiasts can populate a database of apps that relate to maps, while gamers can maintain a database of game apps. 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 or their community, to also communicate or work cooperatively with security experts.

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) in the app, 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 2.

As an example, Table 3 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 * should 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.

We note that in the case 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.

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.

Transparency in Consulting a Database: It appears 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 end-users (or agents working on their behalf) would only be alerted in the event of suspicious behaviour (e.g., certifi-

Table 3: 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, 2011	100
2	Chess	1.6	com.games.chess	0x74A8...	0x8F2C...	Jan 30, 2011	80
3	Chess	1.5.2	com.games.chess	0x1D51...	0xA33C...	Feb 21, 2011	3
4	Checkers	0.5-beta	com.games.checkers	0x74A8...	0xDB89...	Mar 1, 2011	10
5	Chess	1.6	com.games.chess	0xF307...	0x8F2C...	Mar 3, 2011	1

cate change, namespace collision, blacklisted app, etc.). Deployed this way, the application database and Stratus software 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 and security researchers may actively consult the database to find app inconsistencies and overall app statistics, and to compile aggregate statistics.

4.4 Kill Switch Authorities

The final component in the Stratus 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 servers ideally have the capacity to broadcast offending UAppIDs to 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 2) such that kill switch signals are only received for installed apps. Note that 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.

5. USE CASE AND DISCUSSION

Implementing Stratus for a specific smartphone platform is a significant undertaking. It requires many design decisions that are arguably better made with the cooperation of platform manufacturers, market vendors, and representative bodies for developers. We hope, however, that highlighting the problem and outlining the required components of a solution herein will motivate stakeholders in multi-market app ecosystems to orchestrate the creation of a Stratus-like solution, or an analogous one that addresses the relevant challenges.

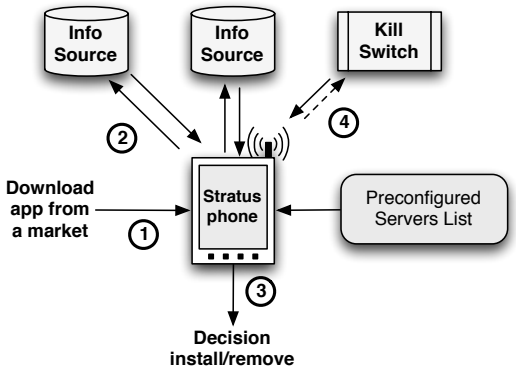

Figure 2: Overview of the Stratus workflow.

In this section, we provide an example use case for Stratus, and discuss the advantages and limitations of our approach.

5.1 Example Use Case

Figure 2 depicts an example use case for Stratus from an end user perspective installing a single app. The following discussion describes the steps required for this workflow.

Step 0. Server configuration: Before Stratus can be used, information sources and kill switch authorities must be configured. A device may ship with a preconfigured list of servers (e.g., added by the carrier or device manufacturer) and new servers may be added by the user (or an agent). We expect users or their agents to find candidate servers via expert websites that recommend good sources for given tasks and report poor quality or malicious sources.

We note that the ideal number of configured servers will vary according to security, usability, and cost[5] 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-expert users and users who don't want to be involved in making security decisions may consider installing more kill switch authorities.

Step 1. App download: The user decides to download an app (e.g., after seeing it on a friend's phone or reading about it on a website), and may optionally search for the app on multiple markets. The app is downloaded and the Stratus client is launched. The app's UAppID is reconstructed.

[5]Well known anti-malware vendors may choose to offer some or all of the Stratus services for a fee to users.

Step 2. Information source query: If the app was downloaded from a new or untrusted site or market, the user or their agent may be given the option to query the preconfigured information sources to obtain additional information about the app. As discussed in Section 4.3, if there are inconsistencies or namespace conflicts, the user or user agent is alerted. Alternatively, installation can proceed as usual while information sources are queried in the background and the user or their agent may be alerted shortly after installing the app if there is a problem.

Step 3. Decision to install/remove the app: Expert users will review the retrieved app details and decide if they want to keep the downloaded app. Smartphones of non-experts may be configured to have a Stratus policy to automatically prevent installation of apps that have any type of conflict or negative reviews. Policies may help reduce required user involvement during the installation process. If the app is kept, Stratus can optionally register the app's UAppID with kill switch authorities.

Step 4. Kill switch: At a later time if an app is found to be malicious or inappropriate, the kill switch authorities subscribed to will push offending UAppIDs to subscribed devices. The matching app is uninstalled from devices (possibly preserving user data), and the user or user agent is informed of the removal.

5.2 Application and Security Domains.

We have discussed grouping information sources by type of content or domain. While this grouping may help motivate the participation by experts in the domain, databases could also be grouped by security focus. For example, an organization that specializes on detecting and reporting privacy violations in apps could host a database listing apps that have been analyzed. A different database could contain reports of average data consumption while an app is in use. Such specialization may help recruit and motivate expert members in different communities.

5.3 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. Stratus involves no massive (e.g. antivirus) signature downloads and no CPU-heavy operations performed on the device aside from hashing the individual app in question.

Only intrusive on suspicious behaviour: The Stratus client 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.

5.4 Limitations

Usability: Stratus 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 or using middleware agents), Stratus 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. The information source would 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. 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.

Conflicting information: Handling conflicting information from information sources (e.g., an application being on both a whitelist and a blacklist, or having positive and negative expert reviews) is non-trivial. Writing automated policies and policy software to deal with these conflicts may prove difficult, especially with a highly extensible architecture like Stratus. However, we expect experts in specific domains to identify these conflicts and provide non-conflicting information.

6. RELATED WORK

Stratus 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 [11], 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. [17, 18] 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 Stratus, which aggregates information from multiple expert sources on the device rather than aggregating multiple services on a central server.

Perspectives [21] 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 Stratus 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).

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 [6]. However, detecting malicious applications generally requires experts, who may not be present in all social circles. Stratus attempts to create an ecosystem of domain-specific services that can individually crowdsource information. However, Stratus does not specify how to deal with the problem of expert recruiting or "fame management" [7]. We defer this aspect to each information source.

7. 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 Stratus as a scalable security-enhancing architecture designed to bridge the gap between non-cooperating application markets. Each component in Stratus (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.

While we have not provided an implementation, and we are not in a position to do so easily, we hope that our proposed high level architecture serves as a platform and trigger for stakeholders in the app ecosystem to progress these ideas, or others motivated by them.

8. REFERENCES

[1] Amazon, Inc. Amazon Appstore Digital Rights Management simplifies life for developers and customers. `http://www.amazonappstoredev.com/2011/03/amazon-appstore-digital-rights-management-simplifies-life-for-developers-and-customers.html`, Mar. 2011.

[2] Android Market. March 2011 Security Issue. `https://market.android.com/support/bin/answer.py?answer=1207928`, 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] R. Cannings. Exercising Our Remote Application Removal Feature. `http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html`, June 2010.

[5] 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/`.

[6] 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)*. Springer, Oct 2010.

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

[8] C. Ellison and B. Schneier. Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[9] 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`.

[10] Google. Signing Your Applications - Android Developer Guide. Website. `http://developer.android.com/guide/publishing/app-signing.html`, Mar. 2011.

[11] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the ACM Conference on Computers and Communications Security (CCS)*, pages 18–29, Nov. 1994.

[12] D. Kravets. Android Market Apps Hit With Malware. Wired Threat Level, Mar. 2011. `http://www.wired.com/threatlevel/2011/03/android-malware/`.

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

[14] 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.

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

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

[17] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: Executable analysis in the network cloud. In *Proceedings of the 2nd USENIX workshop on Hot Topics in Security*, page 5. USENIX Association, 2007.

[18] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *Proceedings of the 17th USENIX Security Symposium*, pages 91–106. USENIX Association, 2008.

[19] 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`.

[20] P. van Oorschot and G. Wurster. Reducing Unauthorized Modification of Digital Objects. *IEEE Transactions on Software Engineering*, 2011.

[21] D. Wendlandt, D. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 321–334. USENIX Association, 2008.