Defending Users Against Smartphone Apps: Techniques and Future Directions

William Enck

North Carolina State University
enck@cs.ncsu.edu

Abstract. Smartphone security research has become very popular in response to the rapid, worldwide adoption of new platforms such as Android and iOS. Smartphones are characterized by their ability to run third-party applications, and Android and iOS take this concept to the extreme, offering hundreds of thousands of "apps" through application markets. In response, smartphone security research has focused on protecting users from apps. In this paper, we discuss the current state of smartphone research, including efforts in designing new OS protection mechanisms, as well as performing security analysis of real apps. We offer insight into what works, what has clear limitations, and promising directions for future research.

Keywords: Smartphone security

1 Introduction

Smartphones are a widely popular and growing space of computing technology. In Q2 2011, over 107 million smartphones were sold worldwide, accounting for 25% of mobile devices [34]. Smartphones provide an ultra-portable interface to the Internet and the computational abilities to make it meaningful. Using environment sensors such as GPS, cameras, and accelerometers, they enhance everyday tasks with the wealth of information available from the Internet.

Fundamental to smartphones are applications, colloquially known as "apps." There were not many apps available for early smartphones, and hence adoption was slow. In 2008, a perfect storm emerged: 3G connectivity finally became widespread, handset technology provided "large" touch-screens and useful sensors such as GPS and accelerometers, and the first application market, Apple's App Store, was created. While all of these factors were crucial, the application market played potentially the most important role. There is a strong correlation, if not causation, between the number of applications in Apple's App Store and Google's Android Market and the rising dominance of iOS and Android.

Warnings of smartphone malware were early and succinct. In 2004, long before smartphones gained widespread popularity, Dagon et al. [19] and Guo et al. [37] discussed the dangers of enhancing cellular phones with network and computational power. These dangers derive from the very concept of a "smart" phone. Users have come to trust their cellular phones, carrying them day and

night, and using them for personal and intimate conversations. Increasing code functionality and diversifying its origin results in misplaced trust. It enables eavesdropping and privacy violations. As we place more information and reliance on smartphones, they become targets for information and identify theft, as well as denial of service attacks (e.g., battery exhaustion). Furthermore, their connection to telecommunications networks opens potential for emergency call center DDoS, voice spam, and other attacks on the network.

The initial smartphone security threats still exist, but smartphone malware surveys [56, 30] have reported trends that help focus attention. Smartphone malware is comprised primarily of Trojans, often designed to exfiltrate user information or use premium rate cellular services (e.g., SMS). That is, smartphone malware targets the user. Hence, this paper discusses available and proposed defenses for the user against apps they choose to install. We focus on both *malware* and *grayware* (i.e., dangerous functionality without provable malicious intent).

Current smartphone platforms have two promising characteristics not yet common on PCs. First, protection policies isolated or sandbox applications by default. Second, applications are frequently distributed via application markets, providing centralized software management. To date, security certification has only played a small role [43]; however, so called "kill switches" have proved to be a valuable means of cleaning up affected devices. Regardless of current implementations, opportunities exist to enhance the security of future markets.

In this paper, we survey research proposals for enhancing smartphone security. We classify existing research into two categories: *protection systems*, and *application analysis*. We overview proposals to enhance the existing smartphone protection systems, discussing their benefits and limitations. We then consider techniques for application analysis and their potential use in market-based security certification. In both areas, our goal is to highlight promising techniques and help direct future research.

Much of this survey focuses on the Android platform, which has been the platform of choice for researchers. This likely results because: 1) Android is open source and widely popular, allowing researchers to build prototypes to validate their ideas for real applications; and 2) Android is the only platform that allows (and encourages) flexible communication between applications, which introduces interesting security problems for study.

We begin our survey by discussing protections already in place by current smartphone platforms. Next, we introduce and contrast proposals to enhance these models. We then discuss approaches for analyzing applications to identify dangerous behavior. We conclude by highlighting promising research directions.

2 Background

Shown in Figure 1, smartphones retrieve apps from application markets and run them within a middleware environment. Existing smartphone platforms rely on application markets and platform protection mechanisms for security. We now overview protections currently implemented in popular platforms.

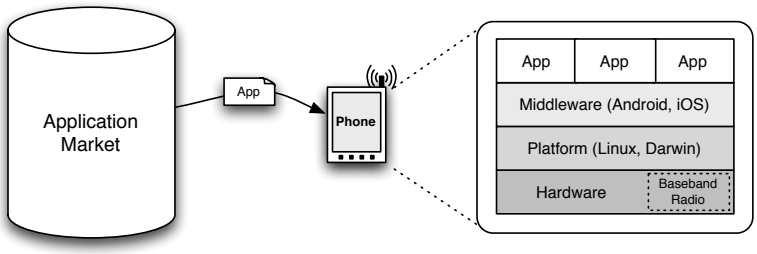

Fig. 1. Smartphone architecture

2.1 Application Markets

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. Using a PC Web browser, the user navigates to a search engine or app aggregation website to find and download an app, and then must connect a USB cable between the PC and the phone to install the application.

Apple's release of the App Store in 2008 triggered a surge in smartphone popularity. Markets benefit developers by simplifying app discovery, sales, and distribution. More importantly, markets benefit users by simplifying app discovery, purchase, and installation. In fact, 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], and was quickly adopted by all other major smartphone platforms.

Application markets can provide several types of security utility. First, they can implement a walled-garden, where the market maintainers have exclusive control over what applications users can install. Second, they can provide a choke point for application security certification. Finally, they can provide remote software management. We compare and contrast Apple's App Store and Google's Android Market to demonstrate these features.

Apple currently implements the walled-garden model for iOS devices. In contrast, Android allows users to install applications from any source, including additional application markets (e.g., the Amazon AppStore). This is often cited as both a feature and security drawback. However, to install a non-Android Market application, the user must change default settings. Most users leave default settings, and therefore are restricted to the applications available in the Android Market. Furthermore, the Android Market restricts what applications are available based on the cellular provider and handset model. Initially, AT&T disabled the ability for its Android devices to install applications from non-Android Market sources, and effectively implementing a walled-garden.

Markets can also provide a choke point for security certification. A walled-garden ensures this, but it is not necessary. If Android users use default settings, they can also benefit. The level of security tests currently implemented is unclear. Apple performs software tests, but they are not forthcoming to the extent of

which are for security. Google performs no testing of applications acceptance into the Android Market; however, they have quickly removed malware when identified by researchers [14]. Given recent discoveries of Android malware [30], they likely perform some "unofficial" security analysis after acceptance.

Finally, markets can remotely manage software on handsets. Software management is a historically difficult challenge in desktop environments. Application markets provide a balance of remote administration that allows users to feel like they are in control, but can intervene when necessary. Google recently demonstrated the value of this model when it not only remotely uninstalled malware from handsets, but also pushed a security patch application that repaired changes to the OS made by the malware [2]. By placing this ability in the market, it is unclear whether users actually need antivirus software.

2.2 Platform Protection

In traditional desktop systems, OS protection policy is based on the user: applications run as the user and can access all the user's files. In contrast, smartphone OS protection policy is based on applications. By default, each smartphone application is isolated, e.g., sandbox policies in iOS, and `uid`s in Android.

Permissions An isolated and unprivileged application has very limited functionality. Therefore, smartphone platforms allow access to individual sensitive resources (e.g., address book, GPS) using permissions. A permission is a form of capability. However, unlike capabilities, they do not always support delegation. Each platform uses permissions in slightly different ways. Au et al. [4] compare the differences between the most prominent platforms.

There are two general types of permissions: *time-of-use* and *install-time*. A time-of-use permission is approved by the user when the application executes a sensitive operation, e.g., iOS's prompt to allow an application access to location. An install-time permission is approved by the user when the application is installed. For Android, this is the user's only opportunity to deny access; the user must accept all permission requests or not install the application.

Install-time permissions serve multiple purposes. They provide [31]: *a*) user consent, *b*) defense-in-depth, and *c*) review triaging. Install-time permissions provide defense-in-depth by defining a maximum privilege level, requiring an attack on an application to additionally exploit a platform vulnerability to perform tasks outside of the application's scope. Studies have also found that applications do not simply request every permission [6, 31], making them valuable attributes for security review triaging. For example, if an application does not have the permission to access location, it cannot possibly leak location information [24]. Felt et al. [31] further discuss the effectiveness of install-time permissions.

Android permissions have two additional important characteristics. First, *permission levels* restrict install-time approval; there are four levels: normal, dangerous, signature, and signature-or-system. Only dangerous permissions are presented to the user. Normal permissions are always granted and provide defense-in-depth and review triage. Signature permissions allow application developers

to control permissions that provide access to exported interfaces. They are only granted to applications signed with the same developer key. Finally, signature-or-system permissions are also granted to applications signed with the firmware key (or installed in Android's "`/system`" partition). Signature permissions are primarily used to prevent third-party apps from using core system functionality.

The second characteristic is Android's limited ability for permission delegation. Permissions protecting exported database interfaces can be delegated to other applications with row-level granularity (if allowed by the database, which is not default). This allows, for example, an Email application to give an image viewer application access to a specific attachment, but not all attachments.

Application Interaction Currently, Android is the only platform that allows flexible application communication. While Android is based on Linux, it has few similarities to a traditional UNIX-based OS. The Android middleware bases execution on *components*, not processes. By standardizing programing interfaces between components, application developers can seamlessly transfer execution between applications, and automatically find the best component and application for a task. Several articles [28, 13, 17] overview component interactions and security concerns, therefore, we restrict ourselves to the highlights.

Applications consist of collections of components. There are four component types: *activity*, *broadcast receiver*, *content provider*, and *service*. Android forces developers to structure applications based on the component types. Activity components define the application's user interface; each "screen" shown to the user is a different activity component. Broadcast receiver components are mailboxes to system and third-party application events, often acting as long-term callback methods. Content provider components are databases and are the primary way to share persistent data between applications. Finally, service components are daemons that define custom RPC interfaces. Each component type has standardized interfaces for interaction; one can, start an activity, broadcast a message to listening receivers, and bind to a service. This interaction is based on a primitive called an *intent message*. An important feature of intent messages is the ability to address them to implicit destinations, called *action strings*. Similar to MIME types, the Android middleware uses action strings to automatically determine which component or components should receive the intent message.

Android's application model requires developers to participate in the phone's security. They must specify (or at least influence) the security policy that protects component interfaces. This security policy is based on permissions. The Android platform defines permissions to protect itself, but developers may define new permissions. As discussed above, Android permissions are requested by and granted to applications at install time. At runtime, components can interact only if the caller application has the permission specified on the callee component. Enck et al. [28] describe additional Android security framework subtleties.

Because Android relies on developers to specify security policy, applications may introduce vulnerabilities for core system resources. Davi et al. [20] were the first to discuss privilege escalation attacks on permissions (not to be confused

with attacks resulting in root privilege). They describe an attack on the Android Scripting Environment (ASE) application. The ASE application is granted the `SEND_SMS` permission at install, and a malicious application is able to use the Tcl scripting interface to send SMS messages to premium-rate numbers. This scenario has also been discussed as a confused deputy attack, where a privileged application cannot (or does not) check if a caller is authorized to indirectly invoke a security sensitive operation [32, 22].

3 Protection Mechanisms

Each smartphone platform defines a protection system to defend users against dangerous functionality in applications. In the previous section, we discussed permission-based protection policy. In this section, we discuss research proposals for enhancing existing smartphone protection systems, as well as their limitations, which often restrict practical deployment.

3.1 Rule Driven Policy Approach

The often-cited limitation of smartphone protection systems is insufficient policy expressibility. To address this, researchers have proposed new policy languages supporting their requirements and demonstrated how to integrate the new policy language into their target operating system. However, to make full use of these policy languages, system developers, application providers, and users need to define an appropriate policy rule-set.

Ion et al. [39] were among the first to define an extended security policy framework for mobile phones. They propose xJ2ME as an extension for J2ME based mobile devices that provides fine-grained runtime enforcement. At the time, unlimited data service plans were rare, and their policies focused on limiting the consumption of network services (e.g., data, SMS, etc). While network service use is still a security concern, unlimited (or practically unlimited, multi-GB) data service plans reduce the need for such policies. Furthermore, determining appropriate quotas for individual applications is not always straightforward, and frequently must be defined by the end user.

Similar to this work, Desmet et al. [21] propose Security-by-Contract (SxC) for the .NET platform to enhance Windows CE based phones. Conceptually, SxC allows the user or application distributor to define a policy specifying how an application should operate when it is run. The contract provides a distinct advantage over simply signing "certified" applications, as the contract can be customized for the target environment. These contracts are similar to the install-time permission model later used by Android, but provide greater expressibility. The contract policies specify allowed security related events, including access and usage quotas for the file system, network, and screen. Similar to xJ2ME, their motivating policies are difficult to define per-application.

The Kirin install-time certification system, proposed by Enck et al. [27], was the first security policy extension for Android. Enck et al. observed that

while Android's install-time permissions inform the user what an application can access, they do not abstract the risk associated with specific combinations of permissions. Kirin uses both permissions and action strings listed in the application's package manifest to infer an upper bound on its functionality. Kirin modifies Android's application installer and can be used to prevent application installation, or to display statements of risk (rather than permissions) at install-time. Kirin is only as good as it's rules, therefore, Enck et al. proposed and followed a methodology based on security requirements engineering to define rules to prevent different types of dangerous functionality. Unfortunately, Kirin rules are limited by Android's permission granularity, and therefore cannot express certain policies, e.g., differentiate network destinations. Furthermore, some policies simply cannot be expressed at install-time, e.g., when an application conditionally accesses a sensitive resource such as location.

Shortly after Kirin, Ongtang et al. [52] proposed Saint. Whereas Kirin focuses on preventing malware, Saint focuses on providing more expressive security policy constraints for developers. Saint policies allow application developers to declaratively specify incoming and outgoing interactions from the point of view of their applications. It defines both install-time and runtime policies. Install-time policy rules place dependency constraints on permissions requested by applications, e.g., based on other permissions, application names, signatures, and versions. More valuable are runtime policies, for which Saint places reference monitor hooks within Android's middleware. The runtime policies specify both caller and callee constraints based on permissions, signatures, configuration, and context (e.g., location, time, etc). Providing both caller and callee policies allows an application to protect who can use its interfaces, as well as declaratively (as opposed to programmatically) restrict on who it can interface with. Like other rule-based policy frameworks, Saint's usefulness is limited by desirable policies. Ongtang et al. motivate Saint with a hypothetical shopping application that utilizes Android's ability to modularize functionality into separate applications. In follow on work [53], the authors demonstrate Saint's value by defining policies for several real applications from the OpenIntents project.

Ongtang et al. [51] also proposed Porscha to enforce digital rights management (DRM) policies for content. Porscha is specifically designed for Email, SMS, and MMS, and allows content owners to specify access control policies that restrict which applications can access the content, and under what conditions, e.g., location and maximum number of views. To do this, Porscha creates a shim in Android's SMS and network communication processing to: 1) intercept messages, 2) remove encryption that binds content to a specific device, and 3) place the messages in restricted storage that enforce content policies. Porscha provides valuable utility to enterprises and governments: the content sender can ensure only trusted applications and read and process messages. However, there is limited motivation to use Porscha for general user communication.

Several additional works have proposed fine-grained policies for Android. Conti et al. [18] proposes CRePE, an access control system for Android that enforces fine-grained policies based on context, e.g., location, time, temperature,

noise, light, and the presence of other devices. Nauman et al. [47] propose the Android Permission Extension (Apex), which allows users to select which permissions an application is actually granted. Apex also supports dynamic policies, such as SMS sending quotas, and times of day that GPS can be read.

Finally, Bugiel et al. [10] propose XManDroid to mitigate permission privilege escalation attacks in Android. XManDroid seeks to prevent both confused deputy attacks and collusion between to applications (which cannot be detected by Kirin). XManDroid tracks communication between components in different applications as an undirected graph with application `uid`s as vertices. System services using the same `uid` are separated using virtual vertices. Policies restrict component interaction based on communication paths and vertex properties. For example, "an application that can obtain location information must not communicate [directly or indirectly with] an application that has network access." The major hurdle for XManDroid is defining useful policies that do not result in excessive false alarms. Not all communication contains sensitive information, and when it does, it may be desired by the user. Therefore, XManDroid needs to define and maintain policy exceptions.

Observations The obvious limitation of rule driven policy frameworks is the definition and maintenance of the rules. When proposing new frameworks, researchers must *a*) motivate the need for enhanced policy expressibility, and *b*) discuss how new policies can be identified, defined, and maintained. If researchers cannot identify a set of rules that require the full extent of the policy expressibility, they should reconsider the requirements. This aids model clarity and rule specification. For example, the final Kirin policy language [27] is significantly simpler than the original proposal [26].

Motivating policy expressibility is difficult when it is designed to address application-specific needs. In such cases, researchers should survey real applications to motivate several scenarios in which the enhanced policy is needed. Ideally, existing applications will motivate the policy expressibility. However, Android applications have been slow to adopt the platform's "applications without boundaries" mentality, and mostly operate in isolation. Therefore, proposals such as Saint must use mostly hypothetical scenarios. Anecdotally, this trend is changing, thereby allowing better motivating examples.

Policy definition and maintenance is a difficult. New proposals often gloss over the fact that their system will require users to define appropriate policy. Simultaneously useful and usable policy systems are very difficult to create. This is likely the reason Android's existing protection system strikes a balance between security and usability. In general, more specific rules often result in fewer exceptions, but require more upfront work, whereas more general rules require less upfront work, but result in more exceptions.

3.2 High-level Policy Approach

Traditional OS protection systems such Bell-LaPadula [7] and Biba [9] define security with respect to information flow control. These approaches label processes

and resources and define a mathematical specification for label interaction, e.g., "no write down," "no read up." Such approaches allow proofs of high-level security guarantees and policy correctness. In contrast, it is difficult to show that a rule driven policy is complete or correct.

Mulliner et al. [45] propose a process labeling model for Windows CE smartphones. Their goal is to prevent cross-service attacks, e.g., to prevent an exploit of a WiFi application from making phone calls. To do this, they assign labels to sensitive resources, e.g., Internet and telephony. When a process accesses a sensitive resource, the resource label is added to the process label (i.e., high-water mark). The system policy defines sets of incompatible labels based on high-level goals of preventing service interaction. An additional rule-set is required to define exceptions to the label propagation model.

A common high-level security goal for smartphones is *isolation* between business and personal applications. Isolation is achieved by defining two security domains (e.g., personal and business) and not allowing information flows between domains. OS virtualization provides a natural method of achieving this goal, e.g., run one OS instance for personal apps, and one for business apps. VMware provides a mobile solution [59]; however, it runs the business security domain as a hosted VM inside of the personal OS security domain. This slightly skewed threat model is a result of usability requirements: employees take their phone to the corporate IT for business VM installation. In contrast, Motorola is investigating bare metal hypervisors for mobile phones [36], which provide stronger security guarantees. Similarly, Lange et al. [41] propose the open source L4Android project, which uses an L4-based hypervisor.

Isolation between security domains can also be implemented within the OS. Bugiel et al. [11] propose TrustDroid, which provides lightweight domain isolation in Android. TrustDroid is extensible to many security domains, but is motivated with three: *system*, *trusted* (third-party), and *untrusted* (third-party). To allow system operation, TrustDroid allows interaction between the *system* domain and both *trusted* and *untrusted* domains. The policy prevents interaction between *trusted* and *untrusted*. To ensure an untrusted app cannot route through a system app to attack a trusted app, TrustDroid modifies system content provider and service components to enforce the isolation policy. By shifting the isolation mechanism within the OS, TrustDroid reduces the processing and memory overhead of running two separate operating systems. It also allows the user to consolidate common resources such as the address book. In the virtualized OS environment, the user must maintain two copies of such resources.

High-level policies have also been proposed to prevent confused deputy attacks in Android. Felt et al. [32] propose IPC Inspection to determine if an application should indirectly access a sensitive operation. IPC Inspection gets its name from Java Stack Inspection, which inspects the call stack for unprivileged code. However, its runtime logic has similarities to low-water mark Biba [9] in that it reduces the effective permission set on an application based on the permissions of the applications that invokes its interfaces. That is, if app A accesses app B, B's effective permissions will be reduced to the intersection of A and B's

permissions. Similar to low-water mark Biba, over time, B's permissions will be reduced to $\emptyset$, therefore, IPC Inspection uses poly-instantiation of applications to reset permissions. Unfortunately, IPC Inspection fundamentally changes the semantics of an Android permission, assigning it transitive implications. This change is incompatible with applications that modularize functionality. For example, the Barcode Scanner application has the `CAMERA` permission to take a picture and return the encoded text string. Normally, the application that calls Barcode Scanner does not need the `CAMERA` permission, nor does it need to read directly from the camera. However, IPC inspection requires the caller application to have the `CAMERA` permission, thereby moving away from least privilege.

IPC Inspection assumes application developers do not properly check caller privilege. However, the challenge is determining the context in which the call originated. To address this, Dietz et al. [22] propose Quire, which records the provenance of a chain of IPC invocations. This approach provides an access control primitive for application developers rather than an enforcement model.

Observations Android's permission-based protection system is rule driven, therefore, one must understand the semantics of individual permissions to understand the global policy. Android permissions are non-comparable and hence cannot be arranged in a lattice, nor are they intended to be transitive. Because of this, high-level policy approaches based entirely on Android permissions will inherently result in many policy exceptions. Permissions can make excellent security hints, if their semantics and limitations are kept in mind. Sensitive information is increasingly application-specific and introduced by third-party applications (e.g., financial). Therefore, application developers must contribute to the global protection policy.

3.3 Platform Hardening

Most smartphone functionality occurs within a middleware layer. This simplifies the underlying platform and allows application of traditional platform hardening technologies. As a result, mandatory access policies can be simpler. For example, Muthukumaran et al. [46] design a custom SELinux policy for OpenMoko to separate trusted and untrusted software. Shabtai et al. [57] describe their experiences porting SELinux to Android, and create a custom SELinux policy. However, they use targeted mode, whereas a strict mode would provide stronger holistic guarantees. Finally, Zhang et al. [60] apply SELinux to a generic Linux phone to provide isolated security domains consistent with the TCG's Trusted Mobile Phone specification.

Integrity measurement and remote attestation have also been applied to smartphones. The Muthukumaran et al. [46] SELinux-based installer was designed to support the policy reduced integrity measurement architecture (PRIMA). Similarly, Zhang et al. [61] discuss an efficient integrity measurement and attestation for the LiMo platform. Finally, Nauman et al. [48] provide integrity measurement of Android applications for enterprises and to prevent malware.

Observations Device security relies on its trusted computing base (TCB), therefore platform hardening is an important component for smartphone security. However, enterprises and users should keep in mind that while SELinux and remote attestation help security, it is a building block. The most significant challenges lie in defining application-level security policies.

3.4 Multiple Users

Smartphone platform designs assume there is one physical user. This simplifies protection systems and allows them to focus on applications. However, users occasionally lend their phone in social situations. Karlson et al. [40] studied how users of different smartphone platforms lend their phone to other physical users. Their findings motivate a reduced-capability guest profile. Liu et al. [42] report similar findings and propose xShare, a modification of Windows Mobile that creates "normal" and "shared" modes. Finally, Ni et al. [49] propose DiffUser for Android. DiffUser expands a phone from a single user model to one that has three classes: administrative users, normal users, and guest users.

Observations The studies confirm our intuition: users sometimes share their smartphones with friends for whom "full access" is undesirable. We will likely see many proposals claiming to have "the solution." Fundamentally, this problem requires user participation, unless the phone can reliably predict which applications the owner would like the current physical user to access (e.g., Web browser and games, but not Email, except when the user needs to share an Email). As existing proposals have shown, modifying a platform to provide a "guest mode" is not terribly complex. Therefore, future research must demonstrate usability.

3.5 Faking Sensitive Information

Studies [24, 23, 25] have identified many smartphone applications leaking phone identifiers and location to servers. In response, Beresford et al. [8] propose providing fake or "mock" information to applications. Their system, MockDroid, returns fake fixed values for location and phone identifiers. MockDroid also fakes Internet connections (by timing out connections), intent broadcasts (by silently dropping them), and SMS/MMS, calendar, and contacts content providers (by return "empty" results). To enable fake data, users must configure "mocked permissions" for each application. TISSA, proposed by Zhou et al. [62], has a similar design with slightly greater flexibility, allowing users to choose from empty, anonymized, or fake results for location, phone identity, contacts, and call logs. Finally, Hornyack et al. [38] propose AppFence. In addition to substituting fake data for phone identifiers and location, AppFence uses TaintDroid [24] (discussed in Section 4) to block network transmissions containing information specified by the user to be used on-device only. AppFence also uses "salted" phone identifiers, which are guaranteed to be unique to a specific application and phone, but different between applications on the phone. This technique allows application developers to track application usage without compromising user privacy.

Observations Transparently incorporating fake information is an elegant way to get around the Android's limitation of not allowing users to deny specific permissions to applications. While permission selection is trivial to implement, it will likely break many existing applications, and therefore is unlikely to be included in the official Android distribution. A second argument against permission selection is usability. Proposals to insert fake information have the same, if not worse, usability limitations. Nonetheless, there is user demand for more control over privacy sensitive information. Finally, there are hidden consequences to faking sensitive information. Many suspect privacy sensitive values are the basis of an advertisement and analytics economy. Restricting privacy values may in turn increase the monetary cost of applications.

4 Application Analysis

As discussed in Section 2, application markets are the primary means of delivering applications to end users. Hence, they can be used as a security choke-point for identifying malicious and dangerous applications. One difficulty of using markets in this manner is a lack of a common definition for "unwanted" applications. Markets quickly remove malicious applications. However, malicious intent is not always clear. Should a market remove applications meant to monitor (i.e., spy on) children? Should an open market, e.g., the Android Market, remove applications that exploit system vulnerabilities to provide the user desired functionality? Beyond this, there is a class of dangerous functionality that many reputable applications include, specifically disclosing privacy sensitive information such as geographic location and phone identifiers without informed consent by the user.

There are limits to the security protections that can be provided by markets [43]. However, recent advancements in application analysis are moving towards more automated certification. In this section, we discuss several approaches for identifying malware and grayware (i.e., dangerous apps without provable malicious intent).

4.1 Permission Analysis

Permissions articulate protection policy, but they also describe what an application can do once installed. As described in Section 3, Enck et al. [27] were the first to use Android permissions to identify dangerous functionality. Kirin breaks dangerous functionality down into the permissions required to perform it. If an application does not have a requisite permission, the attack cannot occur (without exploiting a vulnerability). Enck et al. used Kirin to study 311 top free applications across different Android Market categories. Their rules flagged 10 applications, 5 of which were questionable after reviewing their purpose.

Following this work, Barrera et al. [6] performed permission analysis of the top 50 free applications of every category of the Android Market (1,100 apps in total). They report an exponential decay in the number of applications requesting individual permissions, i.e., many applications request only a small set

of permissions. Barrera et al. also use Self Organizing Maps (SOM) to analyze permission usage. SOM maps the highly dimensional permission space onto a 2-dimensional U-matrix, allowing visual inspection of application permission use. They use heat-maps to show permission frequency in the cells, generating a U-matrix for each permission. By comparing U-matrices for different permissions, one can identify permissions that are frequently requested together. Barrera et al. also labeled cells with categories using a winner-take-all strategy. That is, if most applications mapped to a cell are from the "Multimedia" category, then that cell is marked as "Multimedia." However, their findings do not indicate any correlation between categories and permission requests.

Finally, Felt et al. [31] studied the effectiveness of Android's install-time permissions. They considered 100 paid and 856 free applications from the Android Market. Similar to Barrera et al., they found that most applications request a small number of permissions. They also analyzed the frequency of permission requests, comparing free and paid apps. The `INTERNET` permission is by far the most frequently requested. They also found that developers make obvious errors, e.g., requesting non-existent permissions. In follow on work, Felt et al. [29] create a mapping between Android APIs and permissions and propose the Stowaway tool to detect over-privilege in applications. Note that to do this, Stowaway performs static analysis of applications (discussed below). Felt et al. report the 10 most common unnecessary permissions, the top 2 of which are `ACCESS_NETWORK_STATE` and `READ_PHONE_STATE`.

Observations Permissions are valuable for performance efficient security analysis, but they do not tell the whole story. The Android platform developers made security and usability trade-offs when defining permissions, and many researchers have noted granularity issues. For example, the `READ_PHONE_STATE` permission is used to protect the APIs for both determining if the phone is ringing *and* for retrieving phone identifiers. This leads to ambiguity during permission analysis. A second culprit of ambiguity is the `INTERNET` permission: most applications do not need access to all network domains. However, unlike `READ_PHONE_STATE`, making `INTERNET` more granular is nontrivial in Android, as enforcement is performed in the kernel based on a `gid` assigned to applications. At this enforcement point, the DNS name is no longer available. That said, to date, Android application developers are not significantly over-requesting permissions, which leaves some potential for identifying dangerous applications by their permissions. However, studies indicate considering permissions alone is limited, and they are likely best used to steer dynamic and static analysis.

4.2 Dynamic Analysis

Researchers began with permission analysis because application source code was not available. The next step in studying applications is dynamic analysis, i.e., watching applications run. Dynamic analysis can help resolve ambiguity in permission granularity. It also resolves configuration dependencies. For example, the

Kirin study identified applications that only send geographic location information to a network server if the user changes a default configuration.

Enck et al. [24] propose TaintDroid to identify when applications send privacy sensitive information to network servers. To do this, TaintDroid uses dynamic taint analysis, also known as taint tracking. This technique marks information at source APIs when its type is unambiguous. Smartphones have many such sources, e.g., location, camera, microphone, and phone identifiers. Next, the taint tracking system automatically propagates the markings on some granularity, e.g., individual instructions: $a = b + c$. Enck et al. modified Android's Dalvik VM to perform instruction-level taint tracking. They also integrate the taint tracking into the broader system using coarser granularities, e.g., files and IPC messages. Finally, at a taint sink, the taint tracking system inspects markings on API parameters and performs a policy action. TaintDroid uses the network APIs as taint sinks and logs the event if a taint marking exists in a data buffer. Enck et al. used TaintDroid to study 30 popular applications from the Android Market and found 15 sharing location with advertisers and 7 sharing phone identifiers with remote servers, all without the users knowledge.

TaintDroid has several limitations, discussed in the paper. First, TaintDroid can only identify that privacy sensitive information has left the phone, and not if the event is a privacy violation. Determining a privacy violations requires knowledge of (a) if the user was aware or intended it to occur (there are many desirable location-aware applications), and (b) what the remote server does with the value. Most researchers and users are only capable of identifying (a), therefore leaking information without the user's knowledge has generally been considered a privacy violation. Second, TaintDroid only tracks explicit flows. Therefore, a malicious developer can use implicit flows within an application to "scrub" taint markings from variables. However, such actions are likely identifiable using static analysis and draw attention to developers for attempting to hide their tracks.

The TaintDroid analysis framework was made open source and subsequently used by several researchers. MockDroid [8] and TISSA [62] (discussed in Section 3.5) use TaintDroid to evaluate their effectiveness. AppFence [38] (also discussed in Section 3.5) adds enforcement policies to TaintDroid. The authors also study additional applications and characterize privacy exposure. Finally, Gilbert et al. [35] extend TaintDroid to track specific types of implicit flows and discuss approaches for automating application analysis. They find that random inputs commonly get "stuck" in parts of applications' UI. Therefore, they use concolic execution, switching between symbolic and concrete execution as necessary.

Observations Dynamic analysis identifies what actually happens when an application is run. Static analysis (discussed next) cannot capture all runtime configuration and input. For example, the AdMob SDK documentation [1] indicates it will only send location information if a configuration value is set in the application's manifest file. Furthermore, applications can download and execute code, which is not available for static analysis. However, dynamic analysis is limited by scalability. As discussed by Gilbert et al. [35], generating test inputs

is hard. Finally, any automated analysis is limited in its ability to understand user intentions. Ideally, automated privacy analysis should only raise alarms for privacy violations. Researchers seeking to scale tools such as TaintDroid must attempt to characterize identified leaks.

4.3 Static Analysis

Static program analysis can be done with or without source code. Egele et al. [23] propose PiOS to perform static taint analysis directly on iOS application binaries. PiOS reconstructs control flow graphs from compiled Objective-C, which is nontrivial because object method invocation is funneled through a single dispatch routine. Interestingly, Egele et al. found that iOS's handling of user interactions disrupts the control flow in the CFG. Therefore, to identify potential privacy violations, PiOS uses control flow analysis on the CFG, followed by data flow analysis to confirm information reached the sink. Egele et al. use PiOS to study 825 free applications form Apple's App Store, and 582 applications from Cydia's BigBoss repository. They find that more than half leak the privacy sensitive device ID without the user's knowledge. They also report a strong penetration of ad and analytics libraries.

Android researchers have also performed static analysis of low-level representations. Chin et al. [17] propose ComDroid, which operates on use disassembled DEX bytecode. ComDroid identifies vulnerabilities in Intent communication between applications, including: broadcast theft, activity hijacking, service hijacking, malicious broadcast injection, malicious activity launch, and malicious service launch. Chin et al. used ComDroid to analyze 50 popular paid and 50 popular free applications, manually inspecting the results of 20. In these 20 applications, they found 34 exploitable vulnerabilities. Other tools developed by this group, including IPC Inspection [32] and Stowaway [29] (discussed above), build upon ComDroid. However, working directly on DEX bytecode is difficult. As noted in the ComDroid paper [17], its control flow analysis follows all branches, which can result in false negatives.

In contrast, Enck et al. [25] propose `ded` to reverse Android applications to their original Java form, for which sophisticated static program analysis tools already exist. Reversing DEX bytecode to Java bytecode is nontrivial: the JVM is stack-based while the DVM is register-based; DEX inserts scalar constants throughout the bytecode, and most importantly, DEX loses the type semantics of scalars in several important situations. Using `ded`, Enck et al. decompile 1,100 popular applications and perform a breadth of security program analysis. They target both dangerous functionality and vulnerabilities using custom rules specified for the Fortify SCA framework and follow the program analysis with substantial manual inspection of results. In doing so, they report many observations that provide insight into how Android applications are developed. Overall, their findings were similar to previous privacy studies, and echo concerns with Intent APIs. Similar to the iOS study [23], Enck et al. also found a strong penetration of ad and analytics libraries.

Finally, researchers modeled Android component interaction using source code analysis. Chaudhuri [15] proposes a formal model for tracking flows between applications using permissions as security types. In follow-on work, Fuchs et al. [33] propose SCanDroid for automated application certification using the WALA Java bytecode analysis framework. However, using permissions as the basis of security type analysis in Android is limited, since most permissions are non-comparable and cannot be partially ordered. SCanDroid was proposed before `ded` was available, and therefore was only evaluated against open source applications. Moving forward, combining SCanDroid's formal model and analysis tools with the motivations of ComDroid [17] and IPC Inspection [32] and applying it to code recovered by `ded` has potential for more accurate results.

Observations Static code analysis of Android applications is not as simple as one might initially think. While Fortify SCA was useful, Enck et al. [25] found that custom tools are required to overcome analysis hurdles created by the Android middleware. For example, component IPC must be tracked through the middleware, the middleware API has many callbacks that indirectly use IPC, and APIs frequently require depend on variable state (e.g., the address book content provider authority string). Additionally, researchers should continue to look beyond privacy analysis. While static analysis can scale the identification of potential privacy leaks, their existence is well known. The challenge for privacy leak analysis is automatically determining if the leak was desired.

4.4 Cloud-based Monitoring

Early smartphone security analysis monitored application behavior from the cloud. Cheng et al. [16] propose SmartSiren, which sends logs of device activity, e.g., SMS and Bluetooth, to a server for aggregate analysis to detect virus and worm outbreaks. Oberheide et al. [50] use virtualized in-cloud security services provided by CloudAV for SMS spam filtering, phishing detection, and centralized blacklists for Bluetooth and IP addresses. Schmidt et al. [55] send device features such as free RAM, user activity, process count, CPU usage, and number of sent SMS messages to a central server for intrusion detection analysis. A similar approach is taken by Shabtai et al. [58] in their "Andromaly" proposal for Android. Portokalidis et al. [54] propose "Paranoid Android," which creates a clone of an Android phone in the cloud. A proxy sits in the network so that the network traffic does not need to be uploaded to the server from the phone, and they use "loose synchronization" to only send data when the user is using the device (to safe energy). Finally, Burguera et al. [12], propose Crowdroid, which crowd-sources intrusion detection based on syscalls used by applications.

Observations Before all of this work, Miettinen et al. [44] discussed the limitations of network based intrusion detection for malicious behavior in smartphones. Their arguments include: (1) administrational boundaries, (2) technical boundaries (e.g., network connection), and (3) conception limitations (e.g., attacks to

local storage not in view of network). While sending logs and virtualization address (3), the former to remain valid. Specifically, Miettinen et al. discuss the need to ensure that systems do not expose private data to the cloud services. It is unclear what level of privacy and administrative control users are willing to lose in order to gain security. As mentioned in Section 2, application market kill switches and software management strike a careful balance.

5 Additional Research Directions

In Sections 3 and 4, we discussed existing research proposals, their limitations, and concluded each discussion area with potential enhancements and future directions. In this section, discuss several additional areas with promise. None of these areas are new for computer security, and each has inherent limitations.

Application Discovery There are hundreds of thousands of applications available for iOS and Android, many of which are practically useless and duplicates of one another. When searching for a new application, the user has to balance *a*) price, *b*) functionality, *c*) aesthetics, and *d*) security (and security is unfortunately often the last consideration). Recommendations are often made via word of mouth, but social search will likely soon emerge. Review services such as Consumer Reports have addressed the first three criteria for decades. As discussed in Section 4, there is no one-size-fits-all criteria for security and privacy. Users have different requirements, particularly when privacy is concerned. One potential model is to use Kirin [27] rules to influence security ratings. To be successful, security reviews need to be integrated into application discovery user interfaces, e.g., application markets. Along these lines, Barrera et al. [5] propose Stratus to consolidate multiple application markets, which can address malware opportunities that arise when bargain shoppers compare prices between markets.

Modularity and Transitivity Android allows developers to be compartmentalize functionality into multiple applications. This has several advantages: 1) it supports least privilege, 2) it creates boundaries that allow OS mediation, and 3) it simplifies application analysis by defining distinct purposes for applications. Advertisement and analytics functionality is an immediate and real example of where compartmentalization can benefit security. Often, applications only require Internet access to support ads or analytics. Splitting off this functionality reduces the privilege needed by applications and allows certification tools to focus on ad and analytics functionality. However, as noted by several researchers [20, 32, 22], separating functionality into applications can result in privilege escalation attacks, because Android's permissions are not transitive. Unfortunately, as discussed in Section 3.2, making permissions transitive is not a practical solution. Therefore, a new security primitive may be required.

Security via UI Workflow Security policies are difficult for users to understand, and there have been many complaints that Android relies on the user to approve install-time permission requests. Security enforcement does not always need to be an explicit permission or policy statement. Consider the two methods of making phone calls in Android. If an application uses the "`CALL`" action string, it requires the `CALL_PHONE` permission, and the call is connected immediately; however, if the application uses the "`DIAL`" action string, no permission is required, and the user is presented the phone's default dialer with the number entered. Realistically, all applications should use the "`DIAL`" action string (unless it replaces the dialer), because the user is naturally involved in the security decision via the workflow. There is no security question, e.g., "allow location," and the user is never aware that a security decision was made. Future research should investigate opportunities to integrate security into the UI workflow.

Developer Tools Studies [25, 17, 32] have shown that developers need more oversight when using security sensitive APIs. In particular, these studies have reported vulnerabilities at application interfaces, i.e., Intents. Developer tools should be enhanced with checks that look for Intent forging attacks, unprotected Intent broadcasts, and confused deputy attacks. For confused deputies, the developer may not have sufficient context to prevent an attack, therefore new primitives such as IPC provenance [22] are required. Additionally, research is needed to ensure that the new security enhanced developer tools are usable, and not simply discarded by developers.

6 Conclusion

Smartphone security research is growing in popularity. To help direct future research, we have described existing protections and surveyed research proposals to enhance security, discussing their advantages and limitations. The proposals have discussed enhanced on-phone protection, as well as application analysis that will aid future certification services. Finally, we discussed several additional areas for future smartphone security research.

References

1. AdMob: AdMob Android SDK: Installation Instructions. `http://www.admob.com/docs/AdMob_Android_SDK_Instructions.pdf`, accessed November 2010
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. Au, K., Zhou, B., Huang, Z., Gill, P., Lie, D.: Short Paper: A Look at SmartPhone Permission Models. In: Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM) (2011)

5. Barrera, D., Enck, W., van Oorschot, P.C.: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. Tech. Rep. TR-11-06, Carleton University, School of Computer Science, Ottawa, ON, Canada (April 2011)
6. Barrera, D., Kayacik, H.G., van Oorshot, P.C., Somayaji, A.: A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In: Proceedings of the ACM Conference on Computer and Communications Security (Oct 2010)
7. Bell, D.E., LaPadula, L.J.: Secure Computer Systems: Mathematical Foundations. Tech. Rep. MTR-2547, Vol. 1, MITRE Corp., Bedford, MA (1973)
8. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: MockDroid: Trading Privacy for Application Functionality on Smartphones. In: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile) (2011)
9. Biba, K.J.: Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, MITRE (Apr 1977)
10. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Tech. Rep. TR-2011-04, Technische Universitat Darmstadt, Center for Advanced Security Research Darmstadt, Darmstadt, Germany (Apr 2011)
11. Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.R., Shastry, B.: Practical and Lightweight Domain Isolation on Android. In: Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM) (2011)
12. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: Behavior-Based Malware Detection System for Android. In: Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM) (2011)
13. Burns, J.: Developing Secure Mobile Applications for Android. iSEC Partners (Oct 2008), http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf
14. Cannings, R.: Exercising Our Remote Application Removal Feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html (Jun 2010)
15. Chaudhuri, A.: Language-Based Security on Android. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS) (Jun 2009)
16. Cheng, J., Wong, S.H., Yang, H., Lu, S.: SmartSiren: Virus Detection and Alert for Smartphones. In: Proceedings of the International conference on Mobile Systems, Applications, and Services (MobiSys) (Jun 2007)
17. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing Inter-Application Communication in Android. In: Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys) (2011)
18. Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: Context-Related Policy Enforcement for Android. In: Proceedings of the 13th Information Security Conference (ISC) (Oct 2010)
19. Dagon, D., Martin, T., Starner, T.: Mobile Phones as Computing Devices: The Viruses are Coming! IEEE Pervasive Computing 3(4), 11–15 (October-December 2004)
20. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege Escalation Attacks on Android. In: Proceedings of the 13th Information Security Conference (ISC) (Oct 2010)
21. Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., Vanoverberghe, D.: Security-by-contract on the .NET platform. Information Security Technical Report 13(1), 25–32 (Jan 2008)

22. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight Provenance for Smart Phone Operating Systems. In: Proceedings of the 20th USENIX Security Symposium (August 2011)
23. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS) (Feb 2011)
24. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Oct 2010)
25. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proceedings of the 20th USENIX Security Symposium (August 2011)
26. Enck, W., Ongtang, M., McDaniel, P.: Mitigating Android Software Misuse Before It Happens. Tech. Rep. NAS-TR-0094-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (Sep 2008)
27. Enck, W., Ongtang, M., McDaniel, P.: On Lightweight Mobile Phone Application Certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS) (Nov 2009)
28. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security & Privacy Magazine 7(1), 50–57 (January/February 2009)
29. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2011)
30. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A Survey of Mobile Malware in the Wild. In: Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM) (2011)
31. Felt, A.P., Greenwood, K., Wagner, D.: The Effectiveness of Application Permissions. In: Proceedings of the USENIX Conference on Web Application Development (WebApps) (2011)
32. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission Re-Delegation: Attacks and Defenses. In: Proceedings of the 20th USENIX Security Symposium (August 2011)
33. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: ScanDroid: Automated Security Certification of Android Applications. `http://www.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf`, accessed January 11, 2011
34. Gartner: Gartner Says Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year; Smartphone Sales Grew 74 Percent. `http://www.gartner.com/it/page.jsp?id=1764714` (Aug 2011)
35. Gilbert, P., Chun, B.G., Cox, L.P., Jung, J.: Vision: Automated Security Validation of Mobile Apps at App Markets. In: Proceedings of the International Workshop on Mobile Cloud Computing and Services (MCS) (2011)
36. Gudeth, K., Pirretti, M., Hoeper, K., Buskey, R.: Short Paper: Delivering Secure Applications on Commercial Mobile Devices: The Case for Bare Metal Hypervisors. In: Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM) (2011)
37. Guo, C., Wang, H.J., Zhu, W.: Smart-Phone Attacks and Defenses. In: Proceedings of the 3rd Workshop on Hot Topics in Networks (HotNets) (2004)

38. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2011)
39. Ion, I., Dragovic, B., Crispo, B.: Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (Dec 2007)
40. Karlson, A.K., Brush, A.B., Schechter, S.: Can I Borrow Your Phone? Understanding Concerns When Sharing Mobile Phones. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI) (Apr 2009)
41. Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., Peter, M.: L4Android: A Generic Operating System Framework for Secure Smartphones. In: Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM) (2011)
42. Liu, Y., Rahmati, A., Huang, Y., Jang, H., Zhong, L., Zhang, Y., Zhang, S.: xShare: Supporting Impromptu Sharing of Mobile Phones. In: Proceedings of the International conference on Mobile Systems, Applications, and Services (MobiSys) (Jun 2009)
43. McDaniel, P., Enck, W.: Not So Great Expectations: Why Application Markets Haven't Failed Security. IEEE Security & Privacy Magazine 8(5), 76–78 (September/October 2010)
44. Miettinen, M., Halonen, P., Hatonen, K.: Host-Based Intrusion Detection for Advanced Mobile Devices. In: Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA) (Apr 2006)
45. Mulliner, C., Vigna, G., Dagon, D., Lee, W.: Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In: Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2006)
46. Muthukumaran, D., Sawani, A., Schiffman, J., Jung, B.M., Jaeger, T.: Measuring Integrity on Mobile Phone Systems. In: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 155–164 (Jun 2008)
47. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In: Proceedings of ASIACCS (2010)
48. Nauman, M., Khan, S., Zhang, X., Seifert, J.P.: Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In: Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (Jun 2010)
49. Ni, X., Yang, Z., Bai, X., Champion, A.C., Xuan, D.: DiffUser: Differentiated User Access Control on Smartphones. In: Proceedings of the 5th IEEE Workshop on Wireless and Sensor Networks Security (WSNS) (Oct 2009)
50. Oberheide, J., Veeraraghavan, K., Cooke, E., Flinn, J., Jahanian, F.: Virtualized In-Cloud Security Services for Mobile Devices. In: Proceedings of the 1st Workshop on Virtualization in Mobile Computing (Jun 2008)
51. Ongtang, M., Butler, K., McDaniel, P.: Porscha: Policy Oriented Secure Content Handling in Android. In: Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC) (Dec 2010)
52. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC). pp. 340–349 (Dec 2009)
53. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. Journal of Security and Communication Networks (2011), (Published online August 2011)

54. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Android: Versatile Protection For Smartphones. In: Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC) (Dec 2010)
55. Schmidt, A.D., Peters, F., Lamour, F., Albayrak, S.: Monitoring Smartphones for Anomaly Detection. In: Proceedings of the 1st International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (MOBILWARE) (2008)
56. Schmidt, A.D., Schmidt, H.G., Batyuk, L., Clausen, J.H., Camtepe, S.A., Albayrak, S.: Smartphone Malware Evolution Revisited: Android Next Target? In: Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE) (Oct 2009)
57. Shabtai, A., Fledel, Y., Elovici, Y.: Securing Android-Powered Mobile Devices Using SELinux. IEEE Security and Privacy Magazine (May/June 2010)
58. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "Andromaly": A Behavioral Malware Detection Framework for Android Devices. Journal of Intelligent Information Systems (2011), published online January 2011
59. VMware, Inc.: VMware Mobile Virtualization Platform. `http://www.vmware.com/products/mobile/`, accessed January 2011
60. Zhang, X., Aciiçmez, O., Seifert, J.P.: A Trusted Mobile Phone Reference Architecture via Secure Kernel. In: Proceedings of the ACM workshop on Scalable Trusted Computing. pp. 7–14 (Nov 2007)
61. Zhang, X., Aciiçmez, O., Seifert, J.P.: Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms. In: Proceedings of the First International ICST Conference on Security and Privacy in Mobile Information and Communication Systems (MobiSec) (Jun 2009)
62. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming Information-Stealing Smartphone Applications (on Android). In: Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST) (Jun 2011)