Reducing Attack Surfaces for Intra-Application Communication in Android

David Kantola, Erika Chin, Warren He, and David Wagner
University of California, Berkeley
{dkantola,emc,-w,daw}@berkeley.edu

ABSTRACT

The complexity of Android's message-passing system has led to numerous vulnerabilities in third-party applications. Many of these vulnerabilities are a result of developers confusing inter-application and intra-application communication mechanisms. Consequently, we propose modifications to the Android platform to detect and protect inter-application messages that should have been intra-application messages. Our approach automatically reduces attack surfaces in legacy applications. We describe our implementation for these changes and evaluate it based on the attack surface reduction and the extent to which our changes break compatibility with a large set of popular applications. We fix 100% of intra-application vulnerabilities found in our previous work, which represents 31.4% of the total security flaws found in that work. Furthermore, we find that 99.4% and 93.0% of Android applications are compatible with our sending and receiving changes, respectively.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.4 [**Operating Systems**]: Communications Management

General Terms

Security

Keywords

Android, message passing, mobile phone security

1. INTRODUCTION

Mobile platforms have rapidly evolved over the course of a few years. Within five years, Android has seen over 16 major revisions to the API. Over this time, developers have produced over 400,000 applications [21] and the Android Market has distributed over 10 billion application downloads, with over a two-fold increase in the last six months of 2011 alone [6, 4]. With over 700,000 Android devices being activated each day [22], millions of people are relying on the Android operating system to provide the functionality to run a multitude of applications.

Users also expect their personal data to be secure and their applications to be isolated from one another. Given the wide range of tasks users perform (from gaming to banking to personal email) and the number of applications users install on their phones, it is critical to protect applications from other (potentially malicious) third-party applications. To provide isolation between applications, Android runs them under separate UIDs, effectively sandboxing the applications [2]. However, problems arise when the Android API pokes holes in the sandbox by allowing applications to communicate with one another. Android provides a message-passing system in which applications can send and receive messages called *Intents*. This system enables the reuse of functionality across applications and provides the system with an interface to applications. It is also widely used to enable different components within an application to communicate with each other. However, if used incautiously, this messaging system can undermine the security of applications. In previous work, we identified many vulnerabilities in the message passing system [7]. We showed that the contents of messages can be sniffed, modified, stolen, or replaced (which can compromise user privacy) and data or otherwise malicious messages can be forged or injected into an application.

In this work, we identify a subclass of communication vulnerabilities and implement and evaluate a solution to automatically detect and patch these vulnerabilities. A common developer mistake is to expose a component or message unintentionally to third-party applications. Android messaging is used for both intra- and inter-application communication, and developer confusion about this distinction can lead to unnecessary exposure of internal application messages and components.

We aim to reduce message-related vulnerabilities by modifying the heuristics that the Android platform uses to determine whether communication is intended to be application-internal. We focus on ways that the platform can protect existing and future applications from these problems. This approach shifts the implementation burden from individual application developers to the platform developer. Instead of relying on developers to resolve their vulnerabilities (and waiting for them to make the fixes), platform changes could take effect with the push of one over-the-air update and be applied to currently installed applications. Our platform

SPSM'12, October 19, 2012, Raleigh, North Carolina, USA.

changes also eliminate the need for access to application source code and avoid complex program analysis.

To prevent unintended exposure of intra-application communication to third-party applications (and thus prevent data leakage and injection attacks), we alter the heuristics that Android uses to determine the eligible senders and recipients of messages. Specifically, we try to identify intended intra-application communication that has been sent and exposed to third-parties, and instead deliver those messages internally and prevent internal components from receiving external messages.

We analyze a large set of popular, third-party Android applications to estimate the compatibility cost of our changes. We find that 99.4% and 93.0% of applications are compatible with our sending and receiving changes, respectively.

In addition to compatibility, we analyze the extent to which our changes increase application security. We analyze a number of previously-identified security vulnerabilities related to messaging and measure how many would be fixed by our proposed platform changes. We find that our changes fix 100% of intra-application vulnerabilities found in previous work, which represents 31.4% of all security flaws found in that work. Our findings show that we can improve the security of applications with low backward-compatibility costs.

We make the following contributions:

1. We present heuristics to identify unintentional public communication.
2. We develop a platform-centric approach to applying this heuristic to automatically prevent this vulnerability.
3. We perform a large-scale analysis of $\sim 1,000$ applications and conduct an in-depth analysis of the compatibility cost (showing the feasibility of this approach) in addition to an analysis of the security gain.
4. We discuss alternative ways to apply the heuristic to improve the security of applications and further reduce the compatibility cost.

2. ANDROID PLATFORM OVERVIEW

An Android application consists of a number of modular components. Types of components include *Activities*, which represent user interface screens; *Services*, which are long-running background tasks; and *Broadcast Receivers*, which are short-running background tasks triggered by a broadcast Intent – a message that can be received by multiple components. An Intent sent to an Activity or Service, by contrast, can be received only by a single component. These three types of components can send and receive Intents.[1] Common terms are listed for reference at the end of this section.

2.1 Intents

Sending an Intent is a two-step process. First, the Intent must be created. Intents include fields such as *action*, representing either the action to perform or the action that is being reported; *data*, a reference to something to act on; and *category*, representing additional information about the kind of component that should receive the Intent.

Intents can also include fields explicitly referencing a desired recipient component or application. We define Intents including either of these fields as *explicit*; all others we call *implicit*, meaning the platform will search for appropriate recipient components in any application that supports an operation specified in the Intent. By contrast, explicit Intents can be delivered only to the specified recipient.

After creating the Intent, whether explicit or implicit, the sender must send the Intent through an Android API call. Since the API call implicitly specifies the type of the destination component, a broadcast Intent, for example, will never be delivered to an Activity.

[1]We do not discuss the Content Provider component, as it represents a database and cannot send or receive Intents.

2.2 Intent Filters

Components declare their ability to receive implicit Intents through the use of *Intent Filters*, which allow developers to specify the kinds of operations a component supports. Intent Filters include the same action, data, and category fields as Intents, and there are rules for matching Intents with Intent Filters. A component can declare zero or more Intent Filters, and if an Intent matches any Intent Filter, it can be delivered to that component. In the case of multiple possible recipients, the delivery selection depends on the component type. A broadcast Intent is delivered to all matching Receivers. A Service Intent is delivered at random to one of the matching Services. If there are multiple matching Activities, either the user will be prompted to choose among the possible recipients or the system will send it to a default Activity for that type of Intent.

2.3 Component Exposure

A component that can receive Intents only from other components in the same application is considered *private*. By contrast, a component that can receive Intents from other applications is considered *exported*, or *public*. Public components expose their functionality to other applications, creating a potential security risk. Developers can explicitly set a component as either public or private in the application manifest using the `exported` flag.

When developers omit this flag, it is up to the platform to heuristically infer whether the component should be exported. The platform's current heuristic is to export a component if it contains one or more Intent Filters (Figure 1). This means components may be exported by the platform, even if the developer is not expecting it. Thus, any time the platform implicitly makes a component exported, this can create a security risk.

Note that a component's exposure is unrelated to the contents of its Intent Filters: a public component can always be addressed with an explicit Intent. However, *Dynamic Receivers* (Broadcast Receivers declared at runtime) are an exception, since some can only receive implicit Intents. Thus, they are always public, and Intents they receive must match one of their Intent Filters.

Besides limiting a component's external interaction through its public or private status, developers can further limit it using Signature- or SignatureOrSystem-level permissions. (Hereafter, we refer to these permissions as simply signature-level permissions.) Components protected with such permissions can receive Intents only from other applications that are either pre-installed or signed by the same developer that created the permission.

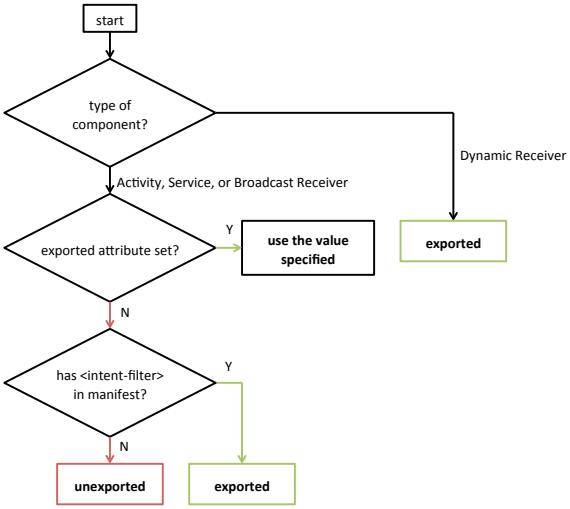

Figure 1: Android exports components that declare an Intent Filter or are explicitly marked for export. Dynamic Receivers are always public.

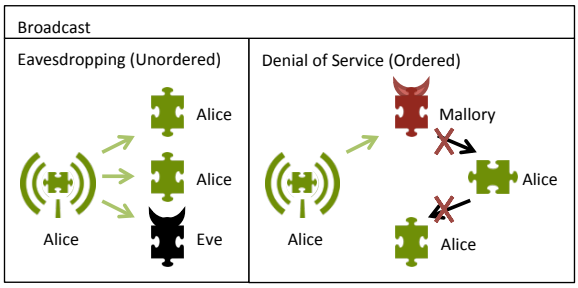

Figure 2: Broadcast Eavesdropping (left): Two of Alice's own components receive the broadcast, but so does Eve's component. Broadcast Denial of Service for Ordered Broadcasts (right): Mallory's component receives the broadcast first and prevents Alice's other components from receiving it.

List of Terms

component a module that can send and receive Intents.
- **Activity** a user interface screen.
- **Broadcast Receiver** a short-running background task.
  - **Dynamic Receiver** declared at runtime.
- **Service** a long-running background task.
- **private** *cannot* receive Intents from other applications.
- **public/exported** *can* receive external Intents.
- **exported flag** makes a component public or private.

Intent a message passed between components.
- **broadcast** can be received by multiple components.
- **explicit** specifies the recipient.
- **implicit** platform decides what component(s) receive it.

Intent field
- **action** what to perform or be reported.
- **category** information about the kind of recipient.
- **data** a reference to something on which to act.

Intent Filter a list of operations a component supports.
signature-level permission only granted by signature.

3. ATTACK SURFACES

Of the Intent-based attack surfaces identified in our previous work, many are exposed by developers unnecessarily using implicit Intents for application-internal messaging. In these cases, neither Intents nor components should be exposed to other applications. We summarize these attack surfaces in this section.

3.1 Threat Model

We only consider Intent-based attacks on applications by other applications that do not hold common signature-level permissions. The only other applications that could hold such permissions are applications signed with the same developer's key or those packaged with the system itself. We do not consider attacks in which a developer's signing key is compromised. An attacker may be any untrusted application installed from an online application distribution platform such as the Android Market [1], while a victim may be any application. Additionally, we only focus on attacks made possible by intra-application communication mechanisms being made available to other applications. We do not attempt to resolve attacks made possible due to intentional inter-application communication.

3.2 Unauthorized Intent Receipt

When components send implicit Intents intended for receipt by other components in the same application (without the protection of a signature-level permission), the Intent is exposed to other applications. Any component of the correct type with a matching Intent Filter can intercept the Intent. The possible attacks enabled by such unauthorized Intent receipt depend on the type of Intent.

Broadcast Intents are vulnerable to passive eavesdropping, which can harm security or privacy if the Intent contains sensitive data (Figure 2). Ordered broadcast Intents, which are delivered to Broadcast Receivers in priority order, are vulnerable to both active denial of service attacks (Figure 2) and malicious data injection. Each recipient of the broadcast has the options to stop the Intent from propagating to the rest of the recipients or change the data before passing it to the next recipient (which in turn may be able to inject the data back into the sending component).

Activity and Service Intents are vulnerable to hijacking attacks, in which an attacker intercepts a request to start an Activity or a request to start or bind to a Service and the malicious application starts its own Activity or Service in its place (Figure 3). This attack allows an attacker to steal data from the Intent, hijack the user interface in a way that may be transparent to the user, and return malicious data to the sending component.

3.3 Intent Spoofing

If a developer unintentionally leaves an internal component exposed to other applications, which can occur if the component declares Intent Filters for receiving implicit Intents, the component may be vulnerable to attacks in which a malicious application spoofs the internal Intent (unless the component is protected by a signature-level permission). Again, the possible attacks enabled by Intent spoofing depend on the type of component exposed.

Broadcast Receivers are vulnerable to broadcast injection, in which the receiving component is tricked into believing a

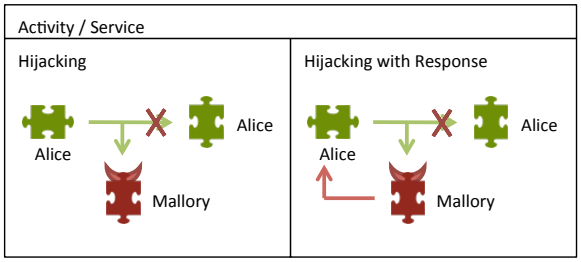

Figure 3: Activity/Service Hijacking (left): Alice's component unintentionally starts Mallory's component instead of her own. False Response (right): Mallory's component returns a malicious result to Alice's component. Alice thinks the result comes from her own component.

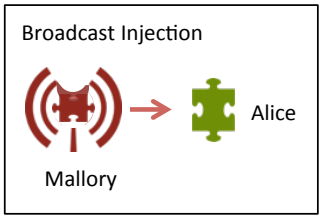

Figure 4: Broadcast Injection: Mallory's component broadcasts to Alice's exported Broadcast Receiver. Alice thinks the broadcast comes from her own component.

malicious broadcast Intent came from another component in its own application (Figure 4). The spoofed broadcast may then cause the victim component to change some state in a way that damages the user's security or privacy or even transmit malicious data contained in the broadcast elsewhere.

Activities and Services are vulnerable to unauthorized launch or bind attacks, in which the attacker either starts or binds to the victim component (Figure 5). These attacks are similar to cross-site request forgeries on the Web, and their exploitation can have similar consequences [17, 5].

4. SYSTEM PLATFORM CHANGES

While the Android platform provides tools for developers to defend intra-application Intent messaging from such security vulnerabilities, we find that many developers use implicit

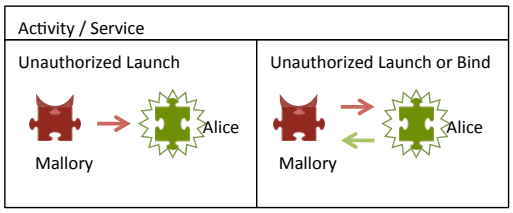

Figure 5: Unauthorized Launch (left): Mallory's component starts Alice's component. Unauthorized Launch or Bind (right): Mallory's component starts or binds to Alice's component and Alice's component returns a result to Mallory's component. Alice thinks she is returning a result to her component.

Intents in an insecure way for application-internal messages. Specifically, developers frequently use the action field of an Intent like an address instead of explicitly addressing the Intent, exposing the communication to both unauthorized Intent receipt and Intent spoofing. Our findings motivate our recommendation for a backward-compatible change to the heuristics the platform uses to determine whether implicit Intents and components that receive them should really be exposed to other applications. Modifications to the platform immediately fix application vulnerabilities. If the onus were placed on individual developers (through a modified communication API, documentation of common communication pitfalls, better developer education, etc.), developers may not choose to update their programming practice and they may not update their legacy applications. The fix would depend on the vigilance of individual developers. As security has often not been a top priority for rapid developers, many applications may remain vulnerable. By implementing a platform-centric solution, we reduce the burden of the developers and immediately patch all applications, including those already installed on users' phones. Unfortunately, changes may break backward-compatibility with current Android behavior (which we evaluate in Section 5).

We propose heuristics that we implemented in Version 2.2 of the Android platform, revising and extending the changes to increase compatibility with legacy applications as we gained a better understanding of the platform. In addition, we were able to identify ways to modify the heuristics to increase the security gain without incurring additional compatibility cost. In this section, we introduce terminology, discuss our heuristics, and discuss how Android was modified to implement our heuristics.

4.1 Terminology

We define some terminology that we will be using later in this section:

- Standard action - A standard action is an action string that is either (1) defined in the Android documentation or (2) used in Android applications bundled with the open-source distribution of Android, since these applications and their public APIs could be considered as convention. We identified 299 standard actions. (See Table 1 for examples.) Non-standard actions are any other actions (i.e., actions created by third-party developers).

- System-only broadcast Intents - A system-only broadcast Intent is a special broadcast Intent that can be sent only by the operating system. Non-system processes cannot spoof these system-only broadcasts. There are 62 system-only broadcasts. (See Table 2 for examples of system-only broadcasts.) This is also referred to as a protected broadcast. While the operating system can send these messages, it is not limited to sending only this type of message.

- Entry-point Activity - An entry-point Activity is an Activity that is the starting point of an application. Specifically, it is any Activity with an Intent Filter that receives either the `MAIN` action or the `APPWIDGET_CONFIGURE` action, as the presence of either of these actions signifies that the Activity is intended to be started by another application.

Standard actions
android.intent.action.DIAL
android.intent.action.EDIT
android.intent.action.MAIN
android.intent.action.SEARCH
android.intent.action.VIEW

Table 1: A few standard actions (non-exhaustive).

System-only broadcast actions
android.backup.intent.CLEAR
android.intent.action.ACTION_POWER_DISCONNECTED
android.intent.action.BATTERY_CHANGED
android.intent.action.REBOOT
android.intent.action.TIME_TICK

Table 2: A few system-only broadcast actions (non-exhaustive).

4.2 Heuristics

We present heuristics that the platform can use to distinguish the use of Intents for communication between components of the same application from the use of Intents between multiple applications.

Preventing Unauthorized Intent Receipt.

If an implicit Intent can be delivered to any component in the same application, then we assume the developer intended the Intent be used for intra-application communication, and thus the Intent should not be delivered to any other application. We apply this heuristic only if the Intent contains non-standard actions. This effectively restricts modifications of expected Intent delivery to Intent actions that were uniquely created by the developer. When Intents containing these developer-created actions can be delivered within the same application, it is unlikely the Intents were intended to be exposed to other applications, as developers commonly misuse implicit Intents in this way for intra-application messaging. Also, we do not restrict delivery of broadcast Intents protected by a signature-level permission, since this type of Intent cannot be intercepted easily.

Preventing Intent Spoofing.

Android exports a component either (1) if it has the `exported` flag, or (2) if it lacks the flag but has an Intent Filter. (Android infers that registering an Intent Filter indicates that the component is expecting external messages.)

To replace this behavior, we propose a set of more restrictive heuristics for inferring whether a component was intended to be an interface for other applications (Figure 6). If a component is protected with a signature-level permission, we follow the original Android behavior, so such a component that does not set the `exported` flag is exported if it contains any Intent Filters. This behavior is justified because requiring a signature-level permission already protects a component from Intent spoofing, and furthermore may indicate that the component is intended to be exposed to either system applications or applications authored by the same developer.

If, however, a component is not protected with a signature-level permission, we propose it only be exported if at least one of the following is true: it (1) sets the `exported` flag, (2) has an Intent Filter with a data field specified, (3) has an Intent Filter that registers to receive system-only actions, (4) is an entry-point Activity, or (5) has an Intent Filter that registers to receive Intents with a standard action. In addition, we impose the restriction that for non-entry-point Activities to be exported without the `exported` flag set, they must have an Intent Filter that receives the `DEFAULT` category.

Condition 1 identifies when a developer explicitly makes a component public or private. This flag indicates that the developer is aware of the status of the component, so we will not change the component's status contrary to the developer's explicit specification. Condition 2 specifies to make a component public when the Intent Filter contains a data field, an indicator that a sender may be trying to share data references with the external component. Condition 3 identifies a case when the component expects to receive a message from the operating system. These components must be public, sometimes subject to an additional protection discussed later. Condition 4 identifies Activities that are intended to be invoked when a user launches an application. These Activities must be exported so they can be started by external launcher applications. Condition 5 identifies components that are expecting standard actions, which may come from third-party applications, as standard actions represent a kind of Intent messaging protocol. Finally, we only export non-entry-point Activities implicitly if they support the `DEFAULT` category because the standard API calls to start an Activity with an implicit Intent require the Activity to support the `DEFAULT` category. Without this category, the component will receive only explicit Intents. Absent this restriction, Activities that cannot typically receive Intents from external applications become vulnerable to Intent spoofing.

We also define a new `protected` property for Broadcast Receivers. If a Broadcast Receiver declares Intent Filters that *only* receive system-only broadcast actions, we export the component but flag it as *protected*, which means we enforce at runtime that only system-only broadcasts are delivered to the component. As system-only broadcasts alone match the component's Intent Filter, the only way to inject a spoofed Intent into such a component is with an explicit Intent. Enforcing the `protected` property prevents any malicious explicit Intents from reaching the component.

Finally, every time an Intent Filter is associated with a Dynamic Receiver, we created a separate `exported` flag for *each* such Intent Filter. Thus, only exported Intent Filters are considered when resolving a broadcast Intent from another application. An alternative would be to create a single `exported` flag for the Dynamic Receiver. We chose this more restrictive heuristic because developers may not be aware that associating a new Intent Filter with a Dynamic Receiver does not remove previous associations.

Note that our heuristics are more restrictive in exposing both Intents and Components than the existing Android heuristics, and as such they cannot increase the attack surface. Also, we developed these heuristics before evaluating their compatibility and security effects on applications. The only changes we made were in expanding the list of standard actions to include undocumented actions used by applications included with the open-source distribution of Android. On this basis we argue that our evaluation results generalize to other applications.

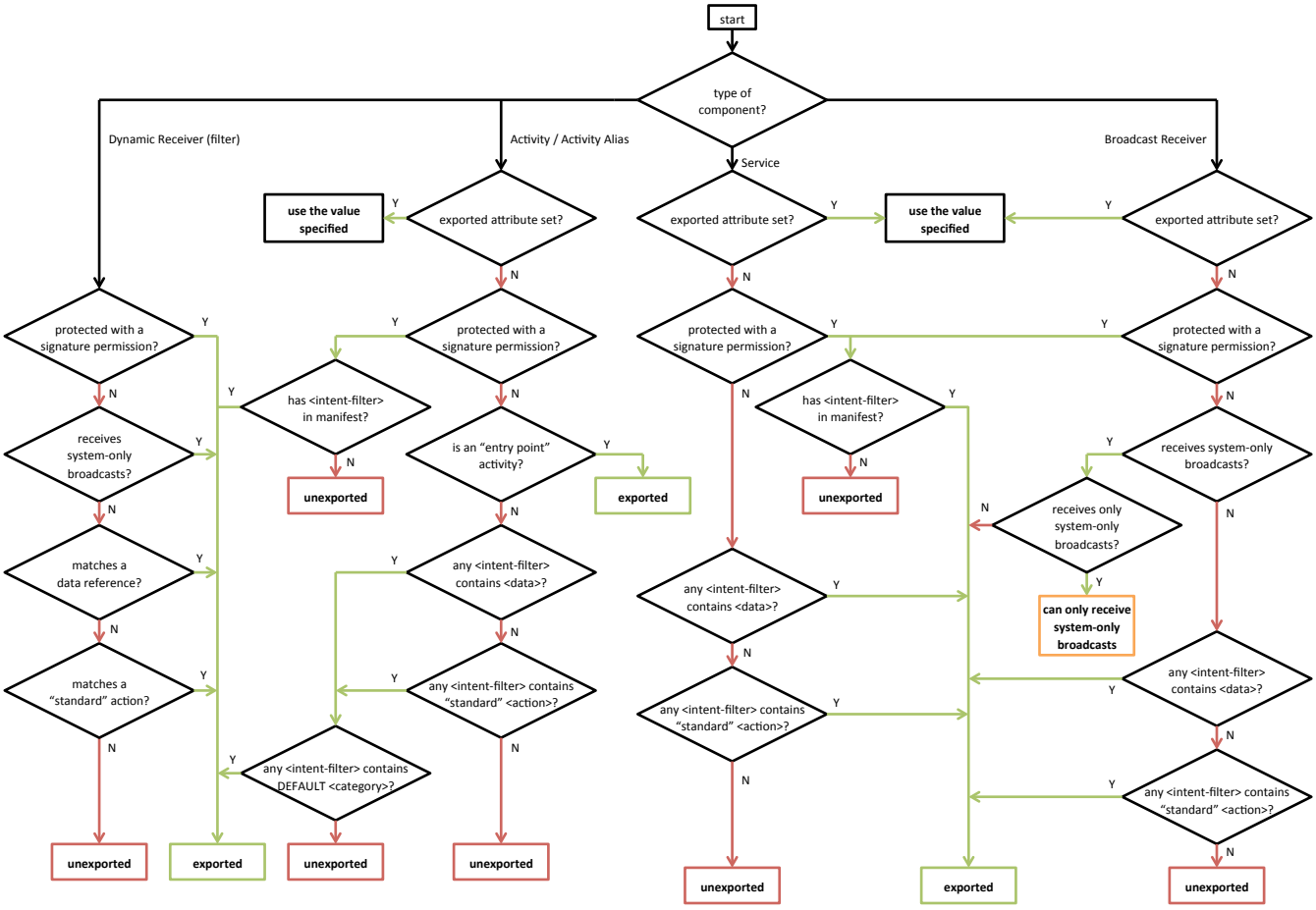

Figure 6: Our heuristic for when components should be exported.

4.3 Implementation

The relevant portions of the Android framework architecture for our heuristic changes are the system server and high-level APIs. The system server is a privileged process containing many threads that has central control over loading applications, managing their meta-data, and delivering Intents, among other things. Each Android application runs in a separate process, which has high-level APIs loaded into the address space of its instance of the Dalvik VM. An application sends and receives Intents through the high-level APIs, which in turn communicate with services running in the system server through a lower layer of IPC that marshals objects across process boundaries.

The two services we modified in the system server are the Activity Manager and the Package Manager. The Activity Manager is responsible for running components, including accepting and delivering Intents. The Package Manager both loads applications and maintains their meta-data, which includes their Intent Filters, so the Package Manager resolves Intents to components.

Our implementation logs a message each time our changes differ from Android's default heuristic (i.e., an implicit Intent is prevented from escaping an application or a component is made private that would have otherwise been public in the original Android heuristic).

4.3.1 Implicit Intent Exposure Changes

To reduce the exposure of implicit Intents, we modified the Intent delivery system to try to deliver the Intent to the origin application first before trying to resolve the Intent to other applications. More specifically, we leveraged the existing `setPackage(callerPackage)` call which limits delivery to a specified application (effectively making the implicit intent temporarily application-explicit) and modified the Activity Manager to call it on any implicit Intent (with the destination set to the origin application) before attempting to resolve it to a component through the Package Manager. If the resolution fails, then there must be no application-internal component that can respond to the Intent, so we call `setPackage(null)` to make the Intent implicit again and attempt to resolve it once more.

To utilize `setPackage(callerPackage)`, we had to modify the implementation of the Intent sending APIs to pass the name of the calling Android package name, a string that uniquely identifies each application, to the Activity Manager. This is necessary because otherwise the Activity Manager can learn only the calling application's UID, PID, and the primary application associated with its process. Since multiple applications can share a process and UID, this information is insufficient to identify the calling application. We modified the implementation of the sending methods, not the interfaces, so this change does not affect the API for developers. We list those methods in Table 3.

The `PackageManager` class also provides methods for resolving Intents to components without sending anything. These are `queryBroadcastReceivers()`, `queryIntentActivities()`, `resolveService()`, `queryIntentServices()`, `resolveActivity()`, and `queryIntentActivityOptions()`. We modified all of these in the same manner.

For broadcasts, one challenge we encountered was the lack of an interface for resolving an Intent to a list of Dynamic Receivers that were created in a specified application. (Due to a bug in the Android source code, `setPackage()` does not limit the recipient to specific applications for broadcast Intents.) We created separate lists for internal Broadcast Receivers and Dynamic Receivers, and if the Intent does not match either kind of receiver, we attempt resolution to external receivers.

Component Type	Methods Modified
Service (Context class)	`startService()`, `bindService()`, `stopService()`
Receiver (Context class)	`sendBroadcast()`, `sendOrderedBroadcast()`, `sendStickyBroadcast()`, `sendStickyOrderedBroadcast()`, `removeStickyBroadcast()`
Activity (Context and Activity class)	`startActivity()`, `startActivityForResult()`, `startActivityIfNeeded()`, `startActivityFromChild()`

Table 3: A list of the Intent sending methods that were modified.

4.3.2 Component Exposure Changes

To implement our component exposure heuristic, we added functionality to help protect Broadcast Receivers and Dynamic Receivers that only expect system-only Broadcasts (from explicit Intent spoofing attacks).

We implemented the enforcement of the protected property in the Activity Manager. If a broadcast Intent resolves to a protected Broadcast Receiver, we allow the Intent to be delivered if the caller has the capability to send a system-only broadcast (i.e., it is one of the operating system processes). Otherwise, we ask the Package Manager whether the Intent's action is system-only. If it is, we log the error and prevent delivery to the Broadcast Receiver.

Implementing the heuristic for Dynamic Receivers was more complex. As a Dynamic Receiver can register multiple Intent Filters, each Intent Filter needs state to track its exposure status. First, we added an `exported` field to the `BroadcastFilter` class, which represents a single Intent Filter. Second, we implemented the code to set the `exported` field using our heuristic for each call to `registerReceiver()` in the Activity Manager. Finally, we added code to enforce the exported property in the Activity Manager. If a `BroadcastFilter` is not exported, we check whether the caller UID and the UID of the application that registered the filter match. If they do not, we log the error and skip the current Dynamic Receiver.

5. EVALUATION

We evaluated our proposed changes on a collection of 969 popular (top free and paid) applications from the Android Market.[2] We believe this to be a suitable dataset as popular applications are more likely to be on users' phones, representing a realistic approximation of potential application interaction. With this dataset, we built upon ComDroid (a static analysis tool that identifies general message vulnerabilities [7]) to look for specific instances where our changes

[2]We originally started with $1,000$ applications and removed applications from the dataset that only consisted of keys to unlock paid features for free applications or were duplicates.

prevent intentional inter-application communication, contributing to incompatibility, as well as instances where our changes eliminate ComDroid vulnerability warnings, contributing to increased security. We call this new tool IntraComDroid. (Hereafter we use IntraComDroid to refer to the tool that we use for our compatibility analysis and component and Intent modication tracking and we use ComDroid to refer to the tool that produces all message-related vulnerability warnings.)

In addition, we used IntraComDroid to examine the extent to which our changes fix concrete security vulnerabilities and unintentional, unnecessarily exposed Intents and components we identified in our previous work. We revisit a case study of a bus schedule application with multiple security vulnerabilities and find that our changes patch all the vulnerabilities.

5.1 Compatibility Analysis

We used static analysis to guide our compatibility investigation. To identify situations where our changes may break inter-application communication, IntraComDroid resolves and records all messages each application receives and sends. Then it analyzes all messages and receiving components in the set to determine the pairs of applications that can communicate with one another. It also analyzes each application and flags all cases in which our proposed heuristics would change the exposure of an Intent or component. Using the previous analysis of all communication in the dataset, IntraComDroid logs two types of potential incompatibilities that warrant manual examination.

The first are instances where an application sends an Intent that one of its own components can receive but components in other applications can also receive. If such a case is an instance of intentional inter-application communication, our changes may break compatibility, as they prevent the Intent from being delivered to the other application.

The second are instances where our changes make a component private, but where IntraComDroid either found other applications that could send Intents that could be received by the component or where it could not find any Intents that address the component at all. In these cases, the concern is that these components were intended as public APIs that our changes will break or that there was an error in the analysis and an Intent was not properly identified.

We manually analyzed each list of potential incompatibilities using several methods. We:

- searched for documentation of public Intent APIs to confirm intentional inter-application communication
- checked archives of Android applications to see whether two applications were different versions of the same application, and presumably not communicating
- read disassembled code to find undetected, internal Intent senders and to understand how Intents were being used in applications
- ran the applications on our modified Android platform, attempting to trigger breakage

Intent exposure compatibility. Out of 969 applications each checked against all 968 other applications, we found 99.4% are compatible with our proposed changes to implicit Intent exposure (Table 4). We classify the six incompatible applications into two categories and show that both can be fixed easily in either application code, in the platform, or in both. First, four applications broadcast Intents to other applications, but also declare they themselves can handle the Intent. In the case where all of these applications are by the same developer, simply protecting the broadcasts with a signature-level permission declared in all applications resolves the incompatibility. If there is no restriction on who developed the receiving applications, we call this a *broadcast protocol* and propose fixing the incompatibility by adding a flag to broadcasts that makes them explicitly public (implemented by application developers). In the second category, four applications share common Service code between two applications by the same developer. Since both applications have the same developer, the developer can resolve the incompatibility by simply protecting the Services with a Signature-level permission declared in both applications.

	Intent Exposure Changes	Component Exposure Changes
Apps Analyzed	969	100
Compatible	99.4%	93.0%
Incompatible	0.6%	5.0%
Uncertain	0.0%	2.0%

Table 4: Compatibility analysis results

Component exposure compatibility. Out of 100 of the most popular applications each checked against all 968 other applications, we found 93% were compatible with our proposed changes to the heuristics used to export components. We were unable to determine whether our changes are incompatible with two applications. We found five incompatible applications, which fell into two types, both of which can be easily fixed in application code. First, two applications use third-party libraries based on Intents for inter-application communication. In this case, only the library developer need document how to explicitly export the appropriate components. New documentation would provide compatibility for new applications, while simply exporting the right components would make legacy applications compatible. Second, three applications allow components to be exported for inter-application communication between applications developed by the same party. The incompatible components were all Receivers, so protecting the broadcasts with a Signature-level permission declared in all applications would make the applications compatible.

5.2 Security Analysis

We evaluated our proposed heuristics by examining the extent to which our changes concretely increase application security for the 20 applications we manually analyzed in previous work. We find our applied heuristics would patch 100% of the subset of warnings that were marked as intra-application communication. Of all of the vulnerabilities and bugs that were detected with ComDroid, our new heuristics patch 31.4% (11/35) of the security vulnerabilities and 100.0% (15/15) of the unintentional exposures. We examined the unpatched vulnerabilities and bugs, and they are all in the class of vulnerabilities where external communication is intended (but still vulnerable to third-party attack). Of the

	Component Exposure	Intent Exposure
Total Warnings	3182	8680
Eliminated Warnings	1431 (45.0%)	1608 (18.5%)

Table 5: The proportion of ComDroid warnings eliminated by our heuristics.

17 remaining unauthorized Intent receipt vulnerabilities and bugs, 4 could be fixed by adding a requirement that certain Intents can be received only by system applications (e.g., Intents that send `android.settings.INPUT_METHOD_SETTINGS` or `android.intent.action.DELETE`). This is the reverse of the existing restriction that some Intents can be sent only by the system. However, this is outside of the scope of our work. Of the 7 unpatched Intent spoofing vulnerabilities, 6 could be fixed by making some of the actions the components expect system-only actions (e.g., `android.intent.action.TIME_SET`, `android.appwidget.action.APPWIDGET_UPDATE`, `android.provider.Telephony.SMS_RECEIVED`). With these changes, our heuristic would then identify them as protected Receivers which would be protected by our system.

We also evaluated our proposed heuristics by examining the proportion of potential security vulnerabilities detected by ComDroid that our changes would eliminate in the set of 969 applications. We use the term "potential security vulnerabilities" because ComDroid issues warnings for exposed communication. Manual examination is required to identify a vulnerability, which we classify as something that exposes data or functionality that can be detrimental to the user, so vulnerabilities are context-dependent. For example, gaining control of an Activity is not considered an exploitable vulnerability unless it could lead to theft of payment or password information.

We find that our platform changes would eliminate 45.0% and 18.5% of ComDroid's receiving and sending warnings, respectively (Table 5). While we can only speculate on how many concrete vulnerabilities this is, we do know that these changes make up 25.6% of the total warnings. This means that developers using ComDroid have 25.6% fewer warnings that they would have had to examine otherwise and may reduce vulnerabilities by 25.6% as well.

Finally, we revisit a case study from our previous work on Nationwide Bus, an application that provides bus location and arrival information for Korean cities [18]. Three kinds of security vulnerabilities were found in the application. First, the application uses an implicit broadcast Intent to send bus information to its own Broadcast Receiver, exposing privacy-sensitive information to eavesdropping applications. Second, the receiver is exported, exposing it to malicious injection of false bus stops and schedules. Third, the receiver forwards the bus information to an exported Service, which is also exposed to malicious injection of false information.

This application illustrates precisely the attack surfaces our new heuristic aims to reduce. As the application can receive its own broadcast, our heuristic detects the Intent as intra-application and prevents eavesdropping applications from receiving it. Furthermore, since our heuristic for component exposure makes the affected components private, other applications can no longer inject malicious information into the components.

5.3 Discussion

We discuss the limitations of our approach, alternative implementations, and the implications for future systems.

Limitations. To prevent unauthorized Intent receipt, our heuristics prevent an implicit Intent from being delivered to external applications when the originating application can receive the Intent. This heuristic restricts an application from sending an Intent to both internal and external recipients. However, our evaluation shows this is currently not a common use of Intents. If this is judged to be an important use case for Intents, a future API could accept a flag explicitly allowing Intents to be delivered externally.

Our compatibility evaluation is limited by the analysis used by IntraComDroid. If IntraComDroid fails to correctly resolve the contents of an Intent for external communication, we lose knowledge of the sent Intent, and thus miss a possible breakage (if our heuristic also makes the recipient component private). For example, static analysis cannot determine the contents of Intents that are dynamically received and forwarded to other components. Similarly, we are also limited by the size of our dataset. If an application communicates with an application outside of our dataset, we have no knowledge of what that other application does. This could result in false negatives, which may cause us to underestimate the compatibility cost of our changes. Despite these limitations, we believe the dataset size and ComDroid analysis are sufficient to estimate compatibility and security.

An alternative approach to static analysis is to run the applications dynamically. The limitation here is that current dynamic Android analyzers do not achieve sufficient execution coverage. They may fail to trigger specific events required to construct and send an Intent, thereby losing outgoing message information. Also, to evaluate whether a message should be delivered internally or externally, the system would need to have knowledge of all possible receivers for all applications. Due to device resource constraints, it is impossible to install a large set of applications on the device at once. Alternatively, each pair of applications in a set could be installed iteratively, but this technique is slow. Our static approach achieves reasonable code coverage and is not limited by resource constraints.

Our compatibility evaluation is also limited by our choice of the most popular applications without regard to what applications have common developers. Since applications with a common developer are more likely to communicate with each other, considering them separately would increase the relevance of a future compatibility evaluation.

Alternative Implementations. In the case that Android is hesitant to push these changes to the platform, our proposed heuristic can be applied in other ways. We discuss the alternatives in order of decreasing security and increasing compatibility. First, the modified platform would be distributed as a third-party, custom ROM, like CyanogenMod [3]. This approach would maintain our compatibility rates, but only users who choose that platform would gain any security benefit. Second, Android Market and other markets could use our static analysis tool to identify applications where our changes could break compatibility. Then they could use our heuristics to selectively perform binary rewriting (to make Intents explicit and components private)

on only the vulnerable but compatible applications. This would increase compatibility and security on most applications. Third, anti-virus software could use our heuristics to identify intra-application communication. It could monitor communication, issue alerts for external communication with "internal" Intents or components, and ignore any known broadcast protocols.

Finally, our heuristics could be used in a lint tool to detect exposed intra-application communication. The lint tool could warn the developer and provide a recommended remediation (e.g., set the `exported` flag to false) any time our heuristics would treat a component differently than the Android platform does. The false positive rate would be low (the recommended remediation would be problematic for only 0.6%–7% of applications, as our compatibility evaluation shows), and if developers test their applications, false positives would likely be relatively harmless. This approach achieves full compatibility, but security gains would accrue gradually over time (as developers would have to opt-in).

Implications for Future Systems. Guessing developer intention is the primary difficulty in automatically patching intra-application communication vulnerabilities. Although our approach shows a low compatibility cost, one way to avoid the guesswork is to make developers declare their intentions upfront. We recommend that future systems require developers to make their intentions explicit.

As a lesson for future systems, we recommend that a system should not implicitly open holes in isolation and expose applications to possible attack without an explicit request by the developer. Isolation should be enabled by default, and system designers should be wary of complex mechanisms that make it tricky for developers to predict when their application might be exposed to attack. Android violates this principle: under certain conditions, it will treat a component as exported even if the `exported` flag has not been set. This has misled developers into using Intents in an unsafe way. Unfortunately, many applications have already been written assuming this behavior, so it cannot be easily changed. In this paper, we develop intricate measures to reduce the number of applications put at risk, while striving to maintain a high level of backward compatibility for existing applications. However, future systems could avoid this complexity and avoid compatibility problems by simply following this principle from the start. In particular, we recommend future systems provide different APIs to separate internal communication from external communication.

Android could follow this recommendation in a future API revision by making the `exported` flag mandatory and adding separate API calls for sending Intents to internal and external components.

6. RELATED WORK

Application Communication. Insecure application communication and exposure can lead to other attacks in addition to information leakage, information injection, and component hijacking. Maji et al. measure the robustness of the Intent system against malformed or unexpected Intents [19]. They build a tool, JJB, to fuzz test Android components. They find that input validation and exception handling are overlooked problems whose absence can result in crashes of the Android runtime system.

Unrestricted access to components can also lead to privilege escalation. Many researchers have examined this problem [13, 9, 8, 15]. Davi et al. discuss privilege escalation through the Android Scripting Environment [8] and Grace et al. present a static analysis tool to detect such attacks [15]. Felt et al. [13] and Dietz et al. [9] further propose runtime defenses. By making unintentionally exposed components private, our work can prevent some access by third-party Intents, thereby avoiding a portion of these problems.

Android Vulnerability Discovery and Measurement. Researchers have also identified other vulnerabilities in Android applications. Felt et al. examine permission overprivilege that arise due to confusion with Android's permission system and present Stowaway, a static analysis tool to detect these bugs in applications [12]. Fuchs et al. examine the combination of permissions and databases, and they present SCanDroid, a static analysis tool that uses a data-centric approach to reason about the security properties of an application [14]. Enck et al. conduct a broad survey of vulnerabilities in Android, including standard Java and Android-specific security threats (e.g., information leakage through logs and messages) [11]. In contrast, we focus solely on vulnerabilities that result from exposing intra-application communication. Also, their work focuses on tools to help developers detect and fix vulnerabilities; in contrast, we focus on platform changes that can protect a large fraction of applications and reduce the risk of these kinds of vulnerabilities.

Hardening Android Applications. We are not the only ones to seek to strengthen the security and privacy of Android through platform modifications. Ongtang et al. present Saint, a modification of the Android platform for runtime enforcement of application provider policies [20]. It provides a means for application developers to set finer-grained policies on whom the application should trust and what it should require before interacting with other applications. Saint assumes developer knowledge in setting security policies and moves control over security decisions to the application developer. Our work assumes less developer expertise and moves the control to the platform.

Enck et al. present TaintDroid, a modification to the Android platform to provide dynamic taint tracking on sensitive data (e.g., location, contact lists, etc.) [10]. Hornyack et al. present AppFence, a tool to provide users with the option to either prevent data from leaving the phone or provide false shadow data in place of legitimate data [16]. These systems focus on protecting user privacy by limiting the behavior of grayware and malware. We focus on fixing vulnerable applications to prevent data leakage and injection.

7. CONCLUSION

Developer confusion on how to write Android applications correctly has rendered many applications vulnerable to attack. We describe an implementation of a better heuristic in the Android platform for detecting unintentional inter-application Intent messaging. We showed that our proposal reduces the number of such vulnerabilities. We evaluated both the security gain and the compatibility cost of our proposed changes, finding 99.4% and 93.0% of applications analyzed are compatible with our Intent exposure and component exposure changes, respectively. Our proposal fixes

31.4% of security flaws found in a previous study. Our work suggests that intra-application communication vulnerabilities in applications can be patched by the Android platform in a way that is reasonably backward-compatible with existing applications.

Acknowledgments

This material is based upon work supported by the Intel Science and Technology Center for Secure Computing and National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of Intel or the National Science Foundation.

8. REFERENCES

[1] Android Market. http://www.android.com/market/.

[2] Android security overview. http://source.android.com/tech/security/index.html#the-application-sandbox.

[3] CyanogenMod. http://www.cyanogenmod.com/.

[4] H. Barra. Android: momentum, mobile and more at Google I/O. http://googleblog.blogspot.com/2011/05/android-momentum-mobile-and-more-at.html, May 2011.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of the 15th ACM Conference on Computer and Communications Security*, October 2008.

[6] C. Bonnington. Google's 10 billion Android app downloads: By the numbers. http://www.wired.com/gadgetlab/2011/12/10-billion-apps-detailed/, December 2011.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services*, June 2011.

[8] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. *Information Security*, pages 346–360, 2011.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. of the 20th USENIX Security Symposium*, August 2011.

[10] W. Enck, P. Gilbert, B.-g. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. of the 20th USENIX Security Symposium*, August 2011.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the 18th ACM Conference on Computer and Communications Security*, October 2011.

[13] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. of the 20th USENIX Security Symposium*, August 2011.

[14] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical report, University of Maryland, 2009.

[15] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proc. of the 19th Annual Symposium on Network and Distributed System Security (NDSS)*, February 2012.

[16] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proc. of the 18th ACM Conference on Computer and Communications Security*, October 2011.

[17] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proc. of the 2nd IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, August 2006.

[18] H. Lee. Nationwide bus. http://www.androlib.com/android.application.net-hyeongkyu-android-incheonbus-Eqwq.aspx.

[19] A. Maji, F. Arshad, S. Bagchi, and J. Rellermeyer. An empirical study of the robustness of inter-component communication in Android. In *Proc. of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2012.

[20] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proc. of the 25th Annual Computer Security Applications Conference (ACSAC)*, December 2009.

[21] I. Paul. Android market tops 400,000 apps. http://www.pcworld.com/article/247247/android_market_tops_400000_apps.html, January 2012.

[22] E. Schonfeld. Android phones pass 700,000 activations per day, approaching 250 million total. http://techcrunch.com/2011/12/22/android-700000/, December 2011.