Modeling and Enhancing Android's Permission System

Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. Several works have recently shown that Android's security architecture cannot prevent many undesired behaviors that compromise the integrity of applications and the privacy of their data. This paper makes two main contributions to the body of research on Android security: first, it develops a formal framework for analyzing Android-style security mechanisms; and, second, it describes the design and implementation of SORBET, an enforcement system that enables developers to use permissions to specify secrecy and integrity policies. Our formal framework is composed of an abstract model with several specific instantiations. The model enables us to formally define some desired security properties, which we can prove hold on SORBET but not on Android. We implement SORBET on top of Android 2.3.7, test it on a Nexus S phone, and demonstrate its usefulness through a case study.

1 Introduction

Recent years have witnessed an explosion in the use of mobile computing thanks to the proliferation of feature-rich smartphones, and associated app stores and easy-to-install applications. Smartphones have powerful hardware, with many useful sensors (e.g., GPS, camera, microphone, accelerometer) exposed via rich APIs, and enough computing power to run complex applications. Applications take advantage of these rich APIs to perform convenient and useful, but potentially privacy-sensitive tasks such as accessing address-book or location information; accessing online banking and medical accounts; and controlling home security systems. App stores make it easy for users to install and run applications, while providing few guarantees about their provenance or behavior.

To protect sensitive resources from applications, and applications from each other, Android and other mobile OSes implement security mechanisms such as permission systems and strong isolation between applications. These mechanisms, however, have in practice proved insufficient, with an increasing number of malicious applications starting to target smartphones [15,23,16].

A number of works have investigated these weaknesses from various perspectives, including demonstrating how applications can communicate through covert channels [24,18], developing tools to detect information leaks [8,5,14], and implementing more powerful protection mechanisms (e.g., [22,20,7,2]).

This paper adds to the body of research on Android security in two main ways: first, by developing a formal framework for analyzing Android-style security mechanisms, including defining properties desired of those, and verifying

S. Foresti, M. Yung, and F. Martinelli (Eds.): ESORICS 2012, LNCS 7459, pp. 1–18, 2012.

whether these properties hold; and, second, by designing and implementing an enforcement system that provides application developers with simple language constructs to specify flexible secrecy and integrity policies, and provably exhibits desirable security properties. To remain practically relevant, we constrain our enforcement system, which we call SORBET, to be easily retrofittable into Android's current architecture. The design and implementation of SORBET improves existing Android permission system in the following aspects: (1) we formally state the properties that we wish our new mechanisms to achieve, and formally prove that our system design supports them; (2) we enhance Android's permission system to support coarse-grained secrecy and integrity policies; and (3) we provide more flexible support for fine-grained and scope-limited delegation of permissions.

Formal analysis. One of our main goals is to improve our understanding of the security properties that we desire of Android-like permission systems, and to verify that specific systems are capable of specifying and enforcing desired properties. We pursue this goal by building a generalized, abstract model of the Android permission system, and stating a set of desirable properties in terms of the model. We then develop instantiations of this model both for the current Android permission system and for SORBET. Based on this formal account, we study the properties of the current system; our investigation reveals both design and implementation flaws, which guide the design of SORBET. We also prove that SORBET's design is sufficient to support the properties that we have defined.

Coarse-grained secrecy and integrity policies. SORBET's key innovation is coarse-grained mechanisms that allow developers to protect their applications against privilege escalation and undesired information flows (e.g., [6,8]). Android's permission system only prevents applications that do not have the correct permissions from directly calling a protected component. This is inadequate to protect against a malicious application that reaches a protected component indirectly, via a chain of calls to innocent applications. To protect against such attacks, we enrich Android's permission system with the ability to specify information-flow constraints and explicit declassification permissions, and implement a light-weight calling-context tracking and checking mechanism. A key challenge here is to support *local* specification of *global* properties.

Flexible and fine-grained delegation. Run-time delegation of URI permissions is a key feature in Android, and allows applications to use third-party components (e.g., a viewer activity) to manipulate content that those components normally would not be permitted to access. On examination, we discovered that Android's implementation of permission delegation is plagued by a number of flaws and questionable design decisions. SORBET supports more flexible and principled permission delegation and revocation, and allows developers to specify constraints that limit the lifespan and redelegation scope of the delegated permissions. Developing a mechanism that correctly enforces lifetime and scope constraints turns out to be unexpectedly tricky, due to redelegation and the dynamic nature of Android applications and components, including application installation and uninstallation, and instantiation and termination of components.

Contributions and Roadmap. This paper makes the following contributions:

- We develop a formal model that generalizes Android-style permissions (§2.2). We show how Android's current permission system can be represented as an instantiation of our abstract model (§2.3).
- Building on this model, we define a set of security properties that one may desire of Android-style permission systems (§3.1). We show that Android currently obeys some of the desired security properties, but not others, and expose several design inconsistencies and implementation flaws (§3.2).
- We describe SORBET, a set of improvements to Android's permission system that supports developer-specified coarse-grained information-flow and privilege-escalation policies. We formalize SORBET as an instantiation of our model and show that it better supports the desired security properties (§4).
- Finally, we implement SORBET on top of Android 2.3.7, test it on a Nexus S phone, and demonstrate several new scenarios that it enables (§5).

2 Preliminaries

We first review the Android architecture as it pertains to permissions (§2.1). We then develop an abstract model of Android-style permission systems (§2.2), and an instantiation of it that captures details of Android's implementation (§2.3).

2.1 Android Overview

Android is a Linux-based open-source OS designed for smartphones. Android applications are written in Java and compiled to Dalvik bytecode. Each application is executed in a separate Dalvik Virtual Machine (DVM) instance.

Android applications are composed of four types of components:

Activities define the user interface. Only one activity interacts with the user at a time. Users typically interact with a sequence of activities to perform a task.

Services run in the background and have no user interface. Unlike activities, services remain active regardless of which application is in the foreground.

Broadcast receivers listen for system-wide broadcasts, and inform other application components upon the receipt of a broadcast.

Content providers store data and are the main way to share data between applications. Each provider exposes a public URI that uniquely identifies its data set. Components and applications can access or update the data via SQL queries.

Activities, services, and broadcast receivers communicate via *intents*, asynchronous messages that deliver data and, if needed, cause a new instance of the recipient component to be created. The OS mediates both cross- and intra-application communications via intents. The recipient of an intent can be specified explicitly by its package and class name, or implicitly via the *action* the intent attempts to initiate. We will often write that a component *calls* another component in lieu of explaining that the communication is via an intent.

Static Constructs

Components	C	$::= C_{code} \mid C_{data}$
Code Components	C_{code}	$::= (name, \mathcal{A}, \varphi_{ckCallee}, \varphi_{ckCaller}, \mathcal{P}_{decl}, \mathcal{P}_{req}, \mathcal{P}_{grnt})$
Data Components	C_{data}	$::= (name, \varphi_{ckCaller}, \mathcal{P}_{decl})$
Component Groups	$\widehat{C}$	$::= (name, \varphi_{ckCallee}, \varphi_{ckCaller}, \mathcal{P}_{decl}, \mathcal{P}_{req}, \mathcal{P}_{grnt}, \{C_1, \cdots, C_n\})$

Run-time Constructs

Run-time Instances	Ins	$::= iC \mid i\widehat{C}$
Comp Instances	iC	$::= (name_r, C, \mathcal{P}_{grnt})$
Comp Group Instances	$i\widehat{C}$	$::= (name_r, \widehat{C}, \mathcal{P}_{grnt}, \{iC_1, \cdots, iC_n\})$
Principals	$Prin$	$::= Ins \mid user$
Targets	Tgt	$::= Ins \mid C \mid \widehat{C}$
Events	E	$::= x = E_1;\ E_2 \mid \mathsf{call}\ iC_1\ iC_2\ I \mid \mathsf{return}\ iC_1\ iC_2\ I \mid \mathsf{resolve}\ iC\ \varphi$ $\mid \mathsf{grant}\ Prin\ Tgt\ P\ \mathcal{F} \mid \mathsf{revoke}\ Prin\ (\{Tgt_1, \cdots, Tgt_n\})\ P$ $\mid \mathsf{checkguard}\ iC\ Tgt\ \varphi \mid \mathsf{exit}\ Ins \mid \mathsf{install}\ Prin\ \widehat{C} \mid \mathsf{uninstall}\ Prin\ \widehat{C}$

Fig. 1. Syntax of permission model

Android uses *(application) permissions* to protect components and sensitive APIs. Permissions are strings (e.g., android.permission.INTERNET) defined by the system or declared by applications. A component or API protected by a permission can be accessed only by applications that hold this permission. Applications acquire (application) permissions only at install time, with the user's consent.

Additionally, content providers can use *URI permissions* to grant ad-hoc access to specific pieces of data that they control (records, tables, or databases). URI permissions can be dynamically granted and revoked.

2.2 Abstract Model

To be able to formally state the properties desired of a permissions architecture, we develop an abstract, formal model of Android-style permissions systems. The model comprises: (1) static elements, which are the code and data we want to protect; (2) run-time elements, such as system events and component instances; and (3) a transition system that captures the behavior of the protection mechanisms. The model is more general than Android's implementation as its purpose is to encompass a wider design space of permission systems, including previously suggested extensions (e.g., [22]). We only sketch the model here; see our technical report [13] for details. Fig. 1 shows the model's static and run-time elements.

Static Constructs. Following Android, applications in our model are built from components. We distinguish between *code components* (C_{code}) and *data components* (C_{data}). Code components—activities, services, and broadcast receivers—may act both as callers and as callees; data components—content providers—are passive and only receive calls. A code component is comprised of a name ($name$), the actions $\mathcal{A}$ to which the component is willing to respond, permissions ($\mathcal{P}_{decl}$, $\mathcal{P}_{req}$, and $\mathcal{P}_{grnt}$), and guards ($\varphi_{ckCallee}$, $\varphi_{ckCaller}$).

In Android, calls to a component are guarded by a permission check. We generalize this check to an abstract guard modeled by a boolean function $\varphi_{ckCaller}$. For now, we specify only that $\varphi_{ckCaller}$ takes as arguments a component and

the calling context and returns true or false. A second general guard, $\varphi_{ckCallee}$, specifies when outgoing calls should be allowed.

We distinguish between permissions that are declared ($\mathcal{P}_{decl}$), requested from the user ($\mathcal{P}_{req}$), and granted ($\mathcal{P}_{grnt}$). This allows us to model behaviors such as dynamic delegation of permissions.

We model applications, $\widehat{C}$, as a set of components ($\{C_1, \cdots, C_n\}$) with guards and permissions that apply to all. This is consistent with Android, where permissions are typically declared, requested, and granted at the application level, but individual components can protect themselves with additional permissions.

Run-Time Constructs. It is important to distinguish static components from run-time instances, and run-time instances from each other. A static component C may have multiple run-time instances iC, composed of a unique identifier (e.g., pointer), $name_r$, and the permissions $\mathcal{P}_{grnt}$ granted to this instance. We similarly model run-time component groups $i\widehat{C}$ (e.g., a running application).

Principals *Prin* are entities that can grant and revoke permissions: run-time components and component groups, and the user (i.e., human who installs applications). Targets *Tgt* are the objects of such operations, and can be either run-time or static components or component groups.

Abstracting detail, we focus on system events that concern permissions, such as component communication via intents (call iC_1 iC_2 I), and granting (grant) and revoking permissions (revoke). We discuss these further in §2.3 and §4.1 when we focus on the Android and SORBET instantiation of the abstract model.

Transition System. We capture the dynamics of the model as a transition system. We model a system state Σ as a tuple composed of a set of entities (run-time and static) and auxiliary data structures Aux. We write $\mathcal{E}$ to denote a sequence of events to be processed by the system. We assume that each event is associated with a unique event ID n. The evolution of the system is a series of transitions ($\Sigma; \mathcal{E} \xrightarrow{o} \Sigma'; \mathcal{E}'$), where o records whether the evaluation of event n is successful ($o = \mathsf{ok}(n)$) or fails ($o = \mathsf{fail}(n)$). Evaluation of a call event will fail, for example, if the appropriate guards don't evaluate to true. A trace, denoted by $\mathcal{T}$, is a sequence of transitions: $\Sigma_0; \mathcal{E}_0 \xrightarrow{o_1} \Sigma_1; \mathcal{E}_1 \cdots \xrightarrow{o_k} \Sigma_k; \mathcal{E}_k$.

The specific rules in the transition system depend on the concrete implementations being modeled. Here we show the rule schema for a successful call event. The call succeeds only if both guards evaluate to true.

CALL-T $(\Sigma; \mathcal{E}, n :: \mathsf{call}\ iC_1\ iC_2\ I) \xrightarrow{\mathsf{ok}(n)} (\Sigma'; \mathcal{E})$ where $\Sigma' = updateCall(\Sigma, \mathsf{call}\ iC_1\ iC_2\ I)$
if $iC_2.\varphi_{ckCaller}(\Sigma, iC_1) = \mathsf{true}$ and $iC_1.\varphi_{ckCallee}(\Sigma, iC_2) = \mathsf{true}$

A parallel rule, CALL-F, specifies that a call fails if either guard returns false.

2.3 Android Model

We instantiate our abstract model to describe the key behaviors of Android's permission system[1]. This has helped us to identify flaws in its implementation

[1] When we refer to Android, we mean version 2.3.7, which was the newest version available while we were carrying out our investigation. The behaviors we describe generally hold in 4.0 as well.

and peculiarities in its design. We omit a full description, but show example instantiations of guards ($\varphi^{uri}_{\mathcal{P}}$) and transition rules for granting permissions.

Guards. The guard $\varphi^{uri}_{\mathcal{P}}$ checks whether a component has the URI permissions specified in $\mathcal{P}$. $\varphi^{uri}_{\mathcal{P}}$ can be used as $\varphi_{ckCaller}$ when $\mathcal{P}$ is the set of URI permissions protecting a component.

We first define functions to look up the permissions associated with a run-time component from the current state. Function $\mathit{grantedByUsrPerm}(iC, \Sigma)$ returns permissions granted at install time, and function $\mathit{URIPerm}(iC, \Sigma)$ returns the URI permissions dynamically granted to iC; *URIPerm* in turn relies on a data structure $\mathcal{M}$ to track the URI permissions granted to each application. Then, we define $\varphi^{uri}_{\mathcal{P}}$ as follows.

$$\varphi^{uri}_{\mathcal{P}} \triangleq f(iC, \Sigma) = \mathcal{P} \subseteq \mathit{grantedByUsrPerm}(iC, \Sigma) \cup \mathit{URIPerm}(iC, \Sigma)$$

Granting Permissions. URI permissions can be granted temporarily, via an intent, or permanently, via grantUriPermission. We model the former as:

$\mathsf{grant}\ iC_1\ iC_2\ P\ \mathcal{F}_{tmp}; \mathsf{call}\ iC_1\ iC_2\ I.$

Here, iC_1 grants permission P with flag $\mathcal{F}_{tmp}$ to iC_2 before transferring control to iC_2. Granting permanently we model as $\mathsf{grant}\ iC_1\ \widehat{C}\ P\ \mathcal{F}_{prm}$. Flags $\mathcal{F}_{tmp}$ and $\mathcal{F}_{prm}$ constrain the lifetime of the delegation of P and the scope of its potential redelegation by iC_2. Mirroring Android, the lifetime of permissions granted with $\mathcal{F}_{tmp}$ is confined to the lifetime of the recipient (iC_2) of the grant operation. When granting with $\mathcal{F}_{prm}$, the recipient will have the permission until the system reboots or the permission is revoked. Neither flag restricts the scope of redelegation. The following rule shows how grant currently works in Android.

$$(\Sigma; \mathcal{E}, n :: \mathsf{grant}\ iC_1\ iC_2\ P\ \mathcal{F}_{tmp}) \xrightarrow{\mathsf{ok}(n)} (\Sigma'; \mathcal{E}) \quad \text{if } \varphi^{uri}_{\{P\}}(iC_1, \Sigma) = \mathsf{true}$$
$$\text{where } \Sigma' = \mathit{updateGrant}(\Sigma, iC_1, iC_2, P, \mathcal{F}_{tmp})$$

Granting succeeds only if the granter has permission P. Afterwards, *updateGrant* updates state, by recording in $\mathcal{M}$ that the enclosing application of iC_2 now has permission P with flag $\mathcal{F}_{tmp}$, and that the instance iC_2 has P in $\mathcal{P}_{grnt}$.

The rule for granting with $\mathcal{F}_{prm}$ (omitted here) differs only in its update function: $\mathcal{M}$ records that now $\widehat{C}$ has permission P with the flag $\mathcal{F}_{prm}$. These rules make explicit that Android does not distinguish between $\mathcal{F}_{tmp}$ and $\mathcal{F}_{prm}$ when deciding whether a component can grant permissions. This causes problems when components redelegate permissions, as we discuss in §3.2.

3 Security Properties

We define several properties that one might desire of an Android-style security architecture (§3.1) and investigate whether they currently hold (§3.2).

3.1 Specifying Desired Security Properties

We formulate the properties desired of Android's security architecture based on the resources that need protection. These are typically interfaces that allow

access to functionality that could cause harm or inconvenience (e.g., sending expensive text messages) and to sensitive data that should not leave the possession of components that legitimately require it (e.g., financial information in a banking application; location information). We abstractly define access-control properties that specify when and how a protected interface can be called and information-flow properties that specify when and what information can flow to or from a component. We also investigate lower-level, functional-correctness properties concerning granting and revoking permissions, since these directly affect the access-control and information-flow properties.

Local Properties. The following two properties state that the immediate restrictions specified by a component on its callers or callees are always obeyed.

PROPERTY 1. (Local callee protection) *If a component A is called by another component B, then A's guard* $\varphi_{ckCallee}$ *evaluates to* true.

PROPERTY 2. (Local caller protection) *If a component A calls another component B, then A's guard* $\varphi_{ckCaller}$ *evaluates to* true.

It is easy to show that Prop. 1 and 2 hold on any instantiation that includes rules like CALL-T and CALL-F (see §2.2).

Delegation and Revocation Properties.

PROPERTY 3. (Delegation) *A component A has a permission P if A owns P, or there is a delegation chain from a component B to A such that A satisfies the scope and lifetime constraints imposed by every component on the chain, and that every component on the chain also has P.*

Intuitively, Prop. 3 ensures that the use of a redelegated permission is confined by the lifetime and scope constraints specified by the original granter. For instance, if an email component gives to a viewer component the URI permission P for displaying an attachment, two sensible constraints are that P is confined to a specific instance of the viewer, and that the viewer cannot redelegate P.

PROPERTY 4. (Revocation) *If A revokes P from B, then there is a delegation chain from A to B, or A owns P.*

This is a basic correctness property for revocation. Allowing arbitrary components to revoke permissions is likely to be disruptive; hence, only the owner or granter should be allowed to revoke a permission.

Global Properties. The next two properties are simplified noninterference. We customize the general notion that secret inputs cannot affect public outputs and tainted inputs cannot affect endorsed outputs to fit the permission-based Android model.

PROPERTY 5. (Privilege escalation) *Given any component B protected by permission P, and any component A that does not have that permission, if* S_{AB} *is a system that contains A and B (and other components), and* S_B *is the same system without A, then a call chain ending with B exists in* S_{AB} *if and only if it exists in* S_B*. Additional call chains ending with B may exist in* S_{AB} *if explicitly allowed by policy.*

In other words, with respect to accessing B, a system with unprivileged component A should behave the same as a system without A. The only exception is if additional policy explicitly allows A to affect B. Without such exceptions, this property would likely be too restrictive.

For example, let B be the interface, guarded by permission P, for rebooting the phone. Suppose that component C has P (which allows it to call B), and a public interface, such that any calls to that interface will cause C to call B. Then, a component A that does not have P can indirectly cause B to be invoked by calling C. C's indiscriminate invocation of B is an example of the confused-deputy problem. Since a trace culminating in that invocation of B cannot exist in a system without A, Prop. 5 prohibits this behavior.

In the other direction, we may want to prevent sensitive information from being leaked, which permission systems typically cannot specify directly. We leverage permissions to state an undesired information flow as follows. Suppose that permission P_1 guards the source of some information and permission P_2 guards the sink. Then, an undesired information flow can be specified as a call chain from a component that uses P_1 to a component that uses P_2. A system that has no undesired information flows should then obey the following property.

PROPERTY 6. (Information flow) *Given an undesired information flow from a component A guarded by P_1 to a component B guarded by P_2, a call chain that ends with B exists in a system with A if and only if the same call chain exists in a system without A. Additional call chains ending with B may exist in the system with A only if explicitly allowed by policy.*

Without a more expressive policy specification language, these properties cannot be specified precisely.

3.2 Analyzing Android Permissions

We investigated the extent to which Android's current permission system, as represented by our model, supports the properties defined in §3.1.

Local Properties Hold. Android's permission system implements the CALL-T and CALL-F rules, and the guards specified by the components are checked at run time; hence, Prop. 1 and 2 hold. However, Prop. 2 holds trivially, because callers cannot state useful guards on callees.

Delegation and Revocation Properties Do Not Hold. Prop. 3 requires that a permission does not outlive the lifespan specified by its granter. Android's implementation, however, does not distinguish between $\mathcal{F}_{tmp}$ and $\mathcal{F}_{prm}$ when deciding whether a component can grant permissions. This violates Prop. 3 and causes several bugs (see our companion technical report [13]), e.g., a component that gained temporary permission can redelegate the permission permanently, including to itself.

Android's revokeURIPermission revokes a URI permission from all components to which it was dynamically granted, and can be called by any component that was granted the permission at install time. This violates Prop. 4, which requires

that a component A can revoke only from entities to which it granted permission (unless A owns the permission). Such violations can easily cause confusion, as unrelated applications can revoke each other's permissions.

Global Properties Do Not Hold. Previous work has pointed out that Android suffers from privilege-escalation flaws (e.g., [6]); i.e., Prop. 5 does not hold. Prop. 6 also does not hold, as Android does not have a mechanism for preventing, or even specifying, undesired information flows. An application can access any component for which it has the permission to do so, regardless of whether it had previously accessed protected information. Previous work has shown that this results in various specific undesired information flows [24,18,8].

Examining Android in light of these properties also revealed several design and implementation bugs, which we reported to Google. These include the ability of components that received a temporary permission to redelegate that permission permanently, and improper bookkeeping of granted permissions during application uninstallation and installation that can lead to privilege escalation. These flaws are discussed in more detail in our companion technical report [13].

4 Sorbet: Android Permissions++

Motivated by the properties of §3.1, we develop SORBET, an improved permission system that supports (1) developer-defined policies to mitigate undesired information flows and privilege-escalation attacks; and (2) well-behaved permission delegation and revocation. Our goals were to enable developers and users to specify richer policies on their applications without dramatically altering Android, and to construct an enforcement system that is provably well behaved.

Some of the mechanisms we use have been discussed previously [10,22,14,7]; we integrate these and other ideas into a system that we can formally show satisfies interesting security properties and enables new use cases.

4.1 New Features in Sorbet

Coarse-Grained Information-Flow Protection. SORBET extends Android's permission labels to make them suitable for specifying coarse-grained information-flow policies, and enforces such policies at component and application boundaries. By reusing permission labels, this approach requires little new syntax.

In SORBET, a component A guarded by P_1 (e.g., the contacts permission) can specify (in the application manifest) information-flow policies of the form $\mathsf{disallow\text{-}flow}(P_1, P_2)$. This indicates that any component B that made use of P_1 to access A cannot (including transitively) use permission P_2. A component can also request at install time the permission $\mathsf{allow\text{-}declassify}(P_1, P_2)$ to declassify sensitive information, i.e., to escape the restriction imposed by $\mathsf{disallow\text{-}flow}(P_1, P_2)$. We formalize this mechanism and the property it enforces in §4.2 and §4.3.

Our mechanism can be used by programmers to strengthen their own code by separating trusted information that should remain internal to an application from untrusted flows that may be communicated to the outside, thereby decreasing the chance of the application being misused by malicious ones. The mechanism can also be used to defend against malicious applications or developers, by specifying policies that should hold between applications.

Coarse-Grained Privilege-Escalation Protection. To mitigate the confused-deputy problem, SORBET tracks the permissions of all components on the call stack. When a component A is called, and A is protected by permission P, SORBET checks if every component on the call stack has P. However, this is too restrictive for practical use; e.g., an email app, which needs to use the INTERNET permission to send email, could do so only when started by applications that have the INTERNET permission. To address this, SORBET allows components to request a privileged permission $\hat{P}$. When a component B has the permission $\hat{P}$, it is permitted to call A even when other components on its call stack do not have P. $\hat{P}$ is similar to the *enable privilege* operation in Java stack inspection. Other works have also tracked the call stack for similar purposes (e.g., [7]); SORBET's novelty here is in allowing developers to specify policies, and in enabling proofs that this and other design features allow the system to exhibit desired properties.

As with information flow, SORBET protects against privilege escalation at both component level and application level. To account for Android's inability to completely mediate communication (e.g., via public static fields) between components within an application, the policy enforced at the application level assumes that component boundaries within an application are not respected.

Flag	Recipient	Redelegation scope	Lifetime
$\mathcal{F}_{comp}$	activity	no redelegation	activity exit
$\mathcal{F}_{task}$	activity	activities in the same task	activity exit
$\mathcal{F}_{appTmp}$	activity	activities in the same app	activity exit
$\mathcal{F}_{allTmp}$	activity	any component	activity exit
$\mathcal{F}_{app}$	app	no redelegation	app uninstall
$\mathcal{F}_{all}$	app	unrestricted	app uninstall

Fig. 2. Flags for constraining delegation. Columns show the recipient scope, the scoping constraints of redelegation, and the lifetime of the granted permission.

Principled Redelegation and Revocation. SORBET also addresses Android's problems with indiscriminate redelegation. The challenge here is to design a (correct) mechanism to allow programmers to predictably control delegation lifetime and redelegation scope. Building on Android's notion of temporary and persistent permissions, we enable the grant operation to precisely convey the intended scope of the recipient (a component or an application), the scope of redelegation (none, components in the same task, components in the same application, and unrestricted), and the lifetime of the permission (until the recipient activity exits, or is uninstalled). For simplicity, we converge on six combinations of these constraints (summarized in Fig. 2), which the programmer can specify via flags passed as arguments to grant. The enforcement mechanism enforces the transitive properties that the constraints implicitly require.

SORBET allows a component A to revoke a permission P from component B only if A granted P to B (or A owns P). In other words, the act of delegating creates a new link in a delegation chain, and revocation removes that link.

4.2 Implementation of Improvements in Abstract Model

We now briefly describe SORBET as an instantiation of the abstract model. We focus on mechanisms for enforcing information flow, and briefly discuss privilege escalation. Delegation and revocation are discussed in our technical report [13].

Information-Flow Protection. To enforce information-flow policies specified by $\mathsf{disallow\text{-}flow}(P_1, P_2)$ and $\mathsf{allow\text{-}declassify}(P_1, P_2)$, we augment the model with an auxiliary data structure $\mathcal{N}$, which keeps track of information-flow constraints. More concretely, $\mathcal{N}$ maps a component instance iC to the set of information-flow constraints that includes all such policies specified by components in the call chain before and including iC.

We define $\mathit{forbidP}(\mathcal{N}, iC)$ to return the set of permissions that are forbidden from being used by constraints in $\mathcal{N}(iC)$. For instance, if $\mathcal{N}(iC) = \{\mathsf{disallow\text{-}flow}(P_1, P_2)\}$, then $\mathit{forbidP}$ returns $\{P_2\}$. Function $\mathit{guardP}(\Sigma, iC)$ returns the set of permissions that guards the calls to component iC. A successful call between components in the same group can now be defined as follows.

$$\begin{aligned}\textsc{call-t}\ (\Sigma; \mathcal{E}, n :: \mathsf{call}\ iC_1\ iC_2\ I) &\xrightarrow{\mathsf{ok}(n)} (\mathit{updateCall}(\Sigma, \mathsf{call}\ iC_1\ iC_2\ I); \mathcal{E}) \\ &\text{if } iC_2.\varphi_{ckCaller}(\Sigma, iC_1) = \mathsf{true} \text{ and } iC_1.\varphi_{ckCallee}(\Sigma, iC_2) = \mathsf{true} \\ &\text{and } \mathit{guardP}(\Sigma, iC_2) \cap \mathit{forbidP}(\mathcal{N}, iC_1) = \emptyset\end{aligned}$$

The last line is the added check for information-flow policies. The call succeeds only if the permission required to access the callee is not forbidden by the policy.

If the call succeeds, information will flow from the caller to the callee, and constraints need to be similarly propagated. In addition, the callee has its own constraints that need to be incorporated in $\mathcal{N}$. For this, we define two new functions. $\mathit{updFlow}(\mathcal{N}, iC, \mathit{Fl})$ returns a new mapping $\mathcal{N}'$, where $\mathcal{N}'(iC) = \mathcal{N}(iC) \cup \mathit{Fl}$. $\mathit{updDeclassify}(\mathcal{N}, iC, \mathsf{allow\text{-}declassify}(P_1, P_2))$ returns a new mapping $\mathcal{N}'$, which removes $\mathsf{disallow\text{-}flow}(P_1, P_2)$ from $\mathcal{N}$ for iC. Hence, after a declassification permission $\mathsf{allow\text{-}declassify}(P_1, P_2)$ is encountered, the constraint that forbade access to components guarded by P_2 is lifted. E.g., if the user explicitly allows access to the Internet after private data is read, then this will be allowed.

We define function $\mathit{flowP}(\Sigma, iC)$ to return the set of information-flow constraints that guard the calls to iC, and $\mathit{getDeclassify}(iC)$ to return the set of declassification permissions of iC. The function $\mathit{updateCall}$ first computes $\mathcal{N}' = \mathit{updFlow}(\mathcal{N}, iC_2, \mathit{flowP}(\Sigma, iC_1))$, then $\mathcal{N}'' = \mathit{updFlow}(\mathcal{N}', iC_2, \mathcal{N}(iC_1))$, and finally $\mathcal{N}''' = \mathit{updDeclassify}(\mathcal{N}'', iC_2, \mathit{getDeclassify}(iC_2))$.

Android does not mediate all communications between components within the same application (e.g., via shared static fields). SORBET conservatively assumes that components within an application have communicated, and treats cross-application calls differently. We write $\mathcal{N}_A(iC)$ to be the union of sets of information-flow constraints $\mathcal{N}(iC')$, for each iC' that is in the same application as iC. We define $\mathit{forbidPA}(\mathcal{N}, iC) = \mathcal{N}_A(iC)$. We define function

guardPA(Σ, iC) to return the set of permissions that guards the calls to all components in the same application as component iC. In the rule for cross-application calls, $\mathcal{N}_A$ takes the place of $\mathcal{N}$, and *guardPA* takes the place of *guardP*. This means that if any component in an application has accessed private data protected by disallow-flow(P_1, P_2), then no component in that application can use permission P_2. The update function similarly accumulates all constraints in the entire application, rather than just one component.

Returns are treated similarly to calls, with the caller and callee designations switched. We omit the definitions here for space reasons.

Privilege-Escalation Protection. To prevent privilege escalation, we use auxiliary tree-like data structures to keep track of the full call history. We define a call forest $\mathcal{T}_S$ as a list of call trees $\mathcal{T}$, as follows:

Call Forest $\mathcal{T}_S ::= [\mathcal{T}_1, \cdots, \mathcal{T}_n]$ *Call Tree* $\mathcal{T} ::= (\mathcal{T}_S, (iC, \mathcal{P}))$

We use $\mathcal{MT}_S$ to denote a mapping from run-time components to call forests. Each call tree represents a call chain, and the root of the tree is the last component on the call chain. The child of the root is a call forest, which is a list of call chains, each representing a past call chain to the root component. If component A (which has permissions P_A) calls B (with permissions P_B), and C (with permissions P_C) also calls B, and B has only one run-time instance, then $\mathcal{MT}_S(B) = [([\,], (A, P_A)), ([\,], (C, P_C))]$. In other words, each call tree in the call forest $\mathcal{MT}_S(B)$ records the full context of the call stack. If B now calls D, the call tree $([([\,], (A, P_A)), ([\,], (C, P_C))], (B, P_B))$ will be stored in $\mathcal{MT}_S(D)$.

A call from component A to component B is allowed only when for any permission P that guards the access to B, either A has $\hat{P}$; or A has P and for every call chain recorded in $\mathcal{MT}_S(A)$, either (1) all the components have permission P; or (2) there exists a component C that has permission $\hat{P}$, and all the components in the call stack after C have P.

As with information flow, the rule for cross-application calls assumes that all components within an application have communicated with each other.

4.3 Properties

We prove Sorbet obeys Prop. 1–6. Here we show only the more concrete restatements of Prop. 5 and 6 made possible by Sorbet's new policy statements (disallow-flow, allow-declassify, and $\hat{P}$). For brevity, details and proof sketches are relegated to our companion technical report [13].

We first define an indirect call chain.

Definition 1. (Indirect call chain) *Given components A and B, there exists an* indirect call chain *from A to B if there exist*

1. *components $D_1, \cdots, D_k$; and*
2. *call chains from A to D_1, from D_1 to D_2, $\cdots$, and from D_k to B.*

We say that a component A can *influence* another component B if there is an indirect call chain from A to B. For example, A can affect the behavior of

B (i.e., the intents that B sends) if either (1) A is part of a call chain to B, or (2) A appears in a call chain to some component D, and this chain shares a component with a different call chain to B. The shared component carries A's influence to B.

PROPERTY 5*. (Privilege escalation (2)) *Given a component B protected by permission P, and a component A that does not have P and belongs to a different application than B, if S_{AB} is a system that contains A and B (and other components), and S_B is the same system without A, then a (possibly indirect) call chain that ends in B exists in S_{AB} if and only if it exists in S_B. Additional (possibly indirect) call chains may exist in S_{AB} only if each such chain has a common suffix with a (possibly indirect) call chain from A to B, and there exists a component between A and B that has permission $\hat{P}$; or there is a component B' between A and B, the path between B and B' contains components of the same application, and B' is not protected by permission P but communicated to B via unmonitored channels.*

PROPERTY 6*. (Information flow (2)) *Suppose a component A is guarded by permission P_1 and an information-flow policy* disallow-flow(P_1, P_2)*, and a component B is guarded by P_2, and A and B belong to different applications. Then, a (possibly indirect) call chain that ends with B, in a system with A, exists if and only if the same call chain exists in a system without A. Additional (possibly indirect) call chains may exist in the system with A only if each such chain has a common suffix with a (possibly indirect) call chain from A to B, and there exists a component between A and B that has permission* allow-declassify(P_1, P_2).

5 Implementing and Evaluating Sorbet

We implemented SORBET on top of Android 2.3.7. This section describes the most salient implementation details, including the syntactic additions for expressing SORBET's policies, and a case study that illustrates SORBET's features.

Syntactic Additions. We extended Android's manifest file syntax to support information-flow and integrity policies. The component protected by P_1 can specify disallow-flow(P_1, P_2) by adding `android:forbiddenPermissions=["`P_2`"]` to the permissions by which this component is protected. allow-declassify(P_1, P_2) is specified as `<declassified-info source=["`P_1`"] destination=["`P_2`"]/>`. A permission is labeled as privileged $\hat{P}$ by the addition of a "privileged" attribute to its declaration: `<uses-permission android:name="`P`" android:privileged="true"/>`.

Implementation Overview. SORBET's keystone is a reference monitor built on top of Android's ActivityManager (Fig. 3). ActivityManager already mediates inter-component communication, which includes preventing calls that are illegal by Android's policy; SORBET modifies it so that mediation of relevant calls is handled by SORBET instead of by the legacy parts of ActivityManager. Enforcing SORBET's policies also requires additional bookkeeping, including of instance data (e.g., to recognize that a particular application has accessed a resource protected by a "forbidden" permission), and richer static policy specified

in application manifests. Hence, a significant component of SORBET's implementation is the data structures that implement this bookkeeping. The bookkeeping includes keeping track of individual files accessed by applications; for enforcement purposes, these are treated as components.

The most challenging part in implementing SORBET was to identify not just which application invoked a protected resource (which Android typically already does) but which specific component instance was responsible for the call; we accomplished this by enhancing Android's IPC data structures to carry more information about the caller. Another challenge was to capture operations not mediated by ActivityManager, such as opening a socket or a file. Android enforces permission-based policies on such operations by Linux-level checks based on the (Linux) group ID of the calling application; applications are assigned group IDs at installation time by the package manager. To mediate access to these operations, we used TOMOYO Linux [21], a set of Linux kernel patches that replaces scattered, ad-hoc access-control checks with centralized ones.[2] We further extended TOMOYO Linux so that access attempts for which policy was enforced at Linux level (e.g., to open a socket or a file) trigger a call to SORBET's reference monitor. This also allows SORBET to mediate security-relevant behaviors implemented in native code that may be included in Android applications.

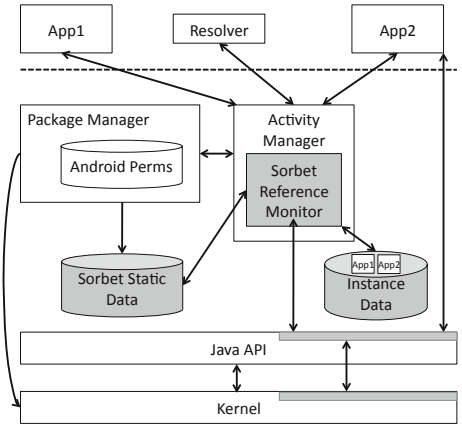

Fig. 3. SORBET architecture: additions to Android are shaded; arrows indicate interactions between system components

Case Study. To test SORBET and illustrate its usefulness, we used it to implement several policies; some that can be implemented (sometimes partially) by previously proposed mechanisms (e.g., [2,7]), and some that require SORBET's features. Our main case study involves four applications: a file manager for storing and manipulating private files (e.g., a diary or list of account numbers); a text editor; an encryption application; and an email application. The high-level policy we focus on is to prevent private files from being leaked on the Internet, but to allow them to be manipulated by various applications at the user's behest (e.g., by using the private file manager to launch an editor). Private files are kept in a content provider implemented by the file manager, and protected by separate permissions that allow read and write access. Applications can access private files only when dynamically delegated the appropriate permissions by the file manager. We next describe several specific scenarios (summarized in Fig. 4) that examine variants of this policy and show how they could be implemented.

[2] TOMOYO Linux has similarly been used by other researchers [2].

Scenario			Private File Manager	Editor	Encryption App	Email App	PE	IF
1	Private files cannot be sent over the network	a	protected by R/W perms	–	–	–	–	–
		b	protected by R/W perms	use Internet	use Internet	use Internet	✓	–
		c	protected by R/W perms forbid Internet	use Internet	use Internet	use Internet	–	✓
2	Private files sent over network only via email	a	protected by R/W perms	use Internet	use Internet	use $\widehat{\text{Internet}}$	✓	–
		b	protected by R/W perms forbid Internet	use Internet	use Internet	use Internet declassify R/W→Internet	–	✓
3	Private files sent over network only via email and if encrypted		protected by R/W perms forbid Internet	use Internet	use Internet declassify R/W→Internet	use $\widehat{\text{Internet}}$	✓	✓

Fig. 4. Three scenarios from our case study. Columns indicate the permissions assigned to each application, and whether enforcement is via protection from privilege escalation (PE), or information flow prevention (IF).

Scenario 1. We start from a base case in which private files must not be sent over the network (Fig. 4, Scenario 1). In Android, the only way to prevent one of these applications from leaking files to the network is to avoid granting any of the applications the Internet permission (Scenario 1a). In SORBET, this policy can be enforced by either the mechanism that prevents privilege escalation or the one that prevents undesired information flows. In the first case, all other applications can be granted the Internet permission, but will no longer be able to use it if the file manager, which does not have this permission, is on the call stack (Scenario 1b). In the second case, the file manager declares the Internet permission as forbidden, with the same effect (Scenario 1c).
Scenario 2. We now extend the desired policy to allow only the email client to send a private file (an activity that the user explicitly initiates), while other applications can use the Internet for other purposes. This cannot be implemented in stock Android, but can still be done with either of SORBET's protection mechanisms. For enforcement via the privilege-escalation mechanism, the email app must declare and be granted the privileged version of the Internet permission. To enforce the same policy via SORBET's information-flow mechanism, the file manager would declare the Internet permission as forbidden (as in Scenario 1), and the email would declare the permission to declassify from R/W to Internet.
Scenario 3. Finally, we extend the policy from Scenario 2 to allow emailing private files only if they are encrypted. Enforcing this without limiting reasonable uses of the email app requires both the information-flow and privilege-escalation mechanisms. As in Scenario 2a, the email app is given the privileged Internet permission, so that it can send email even if indirectly invoked by the file manager, which does not have the Internet permission. In addition, the file manager declares the Internet permission forbidden, and the encryption app is allowed to declassify. Now, the only path to emailing private files is via the encryption app, which is trusted to invoke the email app only with encrypted data.

The last scenario shows that SORBET allows easy specification of useful policies significantly beyond what Android offers. Our case study used minimally modified off-the-shelf applications: Open Manager v2.1.8, Qute Text Editor v0.1, Android Privacy Guard v1.0.9, Email v2.3.4. We modified manifest files, added sending functionality to some, and added a content provider to Open Manager. SORBET's overhead was sufficiently small to be unobservable by the user.[3]

6 Related Work

Researchers have analyzed the security of Android's permission system [5,10], developed analysis tools for Android applications [11], and proposed new protection mechanisms (e.g., [20,22]). Many works studied Android's attack surface (e.g., [19]), including covert channels [24], DoS [1] and web attacks [17], and unauthorized application repackaging [27].

Several works have pointed out flaws in Android's permission system. One weakness is the lack of global properties: Android's permission system does not prevent privilege escalation or information leakage. Davi et al. [6] and Felt et al. [12] have studied privilege-escalation attacks in detail. Bugiel et al. developed a system that monitors interactions between applications at run time and mitigates a wide range of privilege-escalation attacks [2]. Our mechanism has many similarities, but we focus on allowing developers to specify policies on a per-application basis, and emphasize formal analysis of mechanisms. Dietz et al. proposed a framework, Quire, for provenance tracking to mitigate the confused deputy problem [7]. Our goals overlap, but SORBET differs in several ways: We do not track full provenance information, but instead focus on flexible, application-level policy specification based on permissions; we rely on the Android runtime for bookkeeping, rather than using digital signatures. We also support declassification, and formally investigate SORBET's properties. Another approach to mitigating application collusion is through domain isolation. Bugiel et al. assigned trust levels to applications, allowing them to communicate only if they are at the same level [3]. They focus on defining policy for a set of applications at the same trust level, whereas we let applications define policy individually.

Several works have investigated privacy leaks in Android [8,24,4,9]. We provide a formal framework that allows such flaws to be seen as violations of desired security properties. Projects such as TaintDroid [8] and AppFence [14] aim to automatically detect and prevent dangerous information leaks. Our work is in several ways complementary. TaintDroid and AppFence operate at a much finer granularity, tracking tainting at the level of variables, and enforce fixed policies. In contrast, our enforcement is at the component level, and allows developers to specify policies, including, e.g., declassification, which is key to enabling applications that have legitimate reason to send tainted data to operate. We also formally prove that our design enforces desired high-level security properties.

[3] We ran microbenchmarks, but, as common in this setting, the small changes—and sometimes improvements—in latency were dwarfed by the variances between runs.

Systems such as Saint [22] and Apex [20] also improve Android's permission system, e.g., by protecting callers with guards that consider context beyond just permissions, while staying generally close to the original design. We focus on deeper revisions to the permission model and enforcing transitive properties.

Formal analysis of Android-related security issues has received less attention. Shin et al. [25] developed a formal model in order to verify functional correctness properties of Android, which revealed a flaw in the naming scheme for permissions and a possible attack [26]. In contrast, our work develops a more abstract model suitable for reasoning about extensions to Android's permission system.

7 Conclusion

This paper develops a framework for formally analyzing Android-style permission systems, and shows how to enhance Android's permission system to support rich policies while maintaining convenient, application-centric policy specification. We have proved the design of our enforcement system satisfies a set of security properties, showed its feasibility by implementing and running it on a Nexus S phone, and demonstrated its usefulness through a case study. In doing so, we discover that Android's inability to provide strong isolation between components constrains the expressiveness of our system and complicates its implementation. Our system successfully provides both application- and component-level protections, but it would need to resort to application-level protection less often if Android's component-level abstractions were more robust.

Acknowledgments. This research was supported by NSF grants 0917047 and 1018211; by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and W911NF-09-1-0273 from the Army Research Office; and by a gift from KDDI R&D Laboratories Inc.

References

1. Armando, A., Merlo, A., Verderame, M.M.: Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In: Proc. IFIP SEC (2012)
2. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: Proc. NDSS (2012)
3. Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.R., Shastry, B.: Practical and lightweight domain isolation on Android. In: Proc. SPSM (2011)
4. Chaudhuri, A.: Language-based security on Android. In: PLAS Workshop (2009)
5. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proc. MobiSys (2011)
6. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege Escalation Attacks on Android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
7. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight provenance for smart phone operating systems. In: Proc. USENIX Security (2011)

8. Enck, W., Gilbert, P., Gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. USENIX OSDI (2010)
9. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: Proc. USENIX Security (2011)
10. Enck, W., Ongtang, M., McDaniel, P.D.: On lightweight mobile phone application certification. In: Proc. CCS (2009)
11. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proc. CCS (2011)
12. Felt, A.P., Wang, H., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: Proc. USENIX Security (2011)
13. Fragkaki, E., Bauer, L., Jia, L.: Modeling and enhancing Android's permission system. Tech. Rep. CMU-CyLab-11-020, CyLab, Carnegie Mellon University (2011)
14. 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: Proc. CCS (2011)
15. Lineberry, A., Richardson, D.L., Wyatt, T.: These aren't the permissions you're looking for (2010), `www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf` (accessed April 10, 2012)
16. Loftus, J.: DefCon dings reveal Google product security risks (2011), `gizmodo.com/5828478` (accessed April 10, 2012)
17. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on WebView in the Android system. In: Proc. ACSAC (2011)
18. Marforio, C., Francillon, A., Čapkun, S.: Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Tech. Rep. 724, ETH Zurich (2011)
19. Mylonas, A., Dritsas, S., Tsoumas, B., Gritzalis, D.: Smartphone security evaluation – The malware attack case. In: Proc. SECRYPT (2011)
20. Nauman, M., Khan, S., Zhang, X.: Apex: extending Android permission model and enforcement with user-defined runtime constraints. In: Proc. ASIACCS (2010)
21. NTT Data Corporation: TOMOYO Linux (2012), `tomoyo.sourceforge.jp/` (accessed April 10, 2012)
22. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.D.: Semantically rich application-centric security in Android. In: Proc. ACSAC (2009)
23. Passeri, P.: One year of Android malware (full list) (2011), `hackmageddon.com/2011/08/11/one-year-of-android-malware-full-list/` (accessed June 20, 2012)
24. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound Trojan for smartphones. In: Proc. NDSS (2011)
25. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the Android framework. In: Proc. SocialCom/PASSAT (2010)
26. Shin, W., Kwak, S., Kiyomoto, S., Fukushima, K., Tanaka, T.: A small but non-negligible flaw in the Android permission scheme. In: Proc. POLICY (2010)
27. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party Android marketplaces. In: Proc. CODASPY 2012 (2012)