Modeling and Enhancing Android's Permission System

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

November 30, 2011
(revised April 25, 2012)

CMU-CyLab-11-020

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **25 APR 2012**	2. REPORT TYPE	3. DATES COVERED **00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE **Modeling and Enhancing Android's Permission System**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,CyLab,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. 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.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**20**	

Modeling and Enhancing Android's Permission System

Elli Fragkaki Lujo Bauer Limin Jia David Swasey
Carnegie Mellon University

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 [14, 22, 15].

A number of works have investigated these weaknesses from various perspectives, including demonstrating how applications can communicate through covert channels [23, 17], developing tools to detect information leaks [8, 5, 13], and implementing more powerful protection mechanisms (e.g., [21, 19, 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 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.

Android uses *(application) permissions* to protect components and sensitive APIs. Permissions are strings defined by the system (e.g., android.permission.INTERNET) or declared by applications. A component (or API) protected by a permission can be accessed only by applications that hold this permission. An application can 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., [21]). 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, while 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.

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	$::=$	$x = E_1;\ E_2 \mid \mathsf{call}\ iC_1\ iC_2\ I \mid \mathsf{return}\ iC_1\ iC_2\ I \mid \mathsf{resolve}\ iC\ \varphi$
		$\mid$	$\mathsf{grant}\ Prin\ Tgt\ P\ \mathcal{F} \mid \mathsf{revoke}\ Prin\ (\{Tgt_1, \cdots, Tgt_n\})\ P$
		$\mid$	$\mathsf{checkguard}\ iC\ Tgt\ \varphi \mid \mathsf{exit}\ Ins \mid \mathsf{install}\ Prin\ \widehat{C} \mid \mathsf{uninstall}\ Prin\ \widehat{C}$

Figure 1: Syntax of permission model

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 allow the sequencing of two events using $x = E_1;\ E_2$. The event E_1 is executed first, and produces a result, which substitutes x in E_2.

The event call iC_1 iC_2 I models a component iC_1 attempting to call iC_2 with intent I. When iC_2 is a code component, this models the last step of starting an activity, service, or broadcast receiver. When iC_2 is a data component, this models querying or updating the data component. Similarly, iC_2 can be a Java or Android API method protected by a permission. We model a return from a call as return iC_1 iC_2 I, where I is the intent that conveys the value returned by iC_2 to iC_1.

The resolve iC φ event returns a run-time component that satisfies the specification φ. For instance, the following macro models the sequence of events generated by iC starting an activity by specifying an action A: $(x = \mathsf{resolve}\ iC\ \varphi_A;\ \mathsf{call}\ iC\ x\ I)$. The guard φ_A, defined in §2.3, returns true if the set of accepted actions declared by x includes A.

Event exit $Prin$ signals the termination of a run-time component or group.

Events grant and revoke model granting and revocation of permissions. A permission P is granted or revoked by a principal $Prin$ to or from a target Tgt or set of targets. Granting is additionally described by a flag $\mathcal{F}$, which constrains the lifespan and redelegation scope of the delegated permissions. We discuss these constraints further in §2.3 and §4.1 when we focus on the Android instantiation of the abstract model and our enhancements.

A component can also check whether another component (group) satisfies a specific guard φ using checkguard iC Tgt φ. This is a generalization of the permission-checking operation in Android.

Finally, we model the installation and uninstallation of an application via the install and uninstall events.

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 do not evaluate to true. A trace, denoted by $\mathcal{T}$, is a sequence of transitions: $\Sigma_0; \mathcal{E}_0 \xrightarrow{o_1} \Sigma_1; \mathcal{E}_1 \cdots \xrightarrow{o_k} \Sigma_k; \mathcal{E}_k$.

Most of the specific rules in the transition system depend on the concrete implementations being modeled. Here we show the rules for resolving an instance and calling a component, which are the same for both of the instantiations we model, except for detailed operations on the auxiliary data structures.

The resolve event updates the state to Σ', and returns a run-time instance iC_2 that satisfies the guard φ

under the updated state Σ'.

$$\textsc{resolve-t} \quad (\Sigma; \mathcal{E}, n :: (x = \mathsf{resolve}\ iC_1\ \varphi; E)) \xrightarrow{\mathsf{ok}(n)} (\Sigma'; \mathcal{E}, n' :: E[iC_2/x])$$
$$\text{where } \varphi(\Sigma', iC_2) = \mathsf{true} \text{ and } \Sigma' = \mathit{updateResolve}(\Sigma, \mathsf{resolve}\ iC_1\ \varphi)$$

For brevity, we abstract away the details of the function $\mathit{updateResolve}(\Sigma, \mathsf{resolve}\ iC_1\ \varphi)$. This update will include, if needed, creating the run-time instance iC_2. A similar rule describes the scenario when resolve fails and the output of the transition is $\mathsf{fail}(n)$.

Below is the rule schema for a successful call event. The call succeeds only if both guards evaluate to true.

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

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

$$\textsc{call-f} \quad (\Sigma; \mathcal{E}, n :: \mathsf{call}\ iC_1\ iC_2\ I) \xrightarrow{\mathsf{fail}(n)} (\Sigma; \mathcal{E})$$
$$\text{if } iC_2.\varphi_{ckCaller}(\Sigma, iC_1) = \mathsf{false} \text{ or } iC_1.\varphi_{ckCallee}(\Sigma, iC_2) = \mathsf{false}$$

A call event fails (rule CALL-F) if one of the guards specified by the caller and the callee fails (evaluates to 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 and peculiarities in its design. We discuss in detail the guards used for checking permissions, the transition rules for granting and revoking permissions, and how events such as application exit and phone reboot affect permissions.

Guards The Android permission system uses four guards, which we label φ_{true}, φ_A, $\varphi_{\mathcal{P}}$ and $\varphi_{\mathcal{P}}^{uri}$.

The guard φ_{true} always returns true, and is used when access is unrestricted, e.g., as $\varphi_{ckCallee}$ for components that do not broadcast messages (and hence need not restrict their recipients).

The guard φ_A is used by resolve when a call is described by an action (e.g., ACTION_CALL), rather than being sent to a specific component. $\varphi_A(iC, \Sigma)$ returns true if iC accepts action A.

The guard $\varphi_{\mathcal{P}}$ checks whether a component is granted at install time all permissions in $\mathcal{P}$. $\varphi_{\mathcal{P}}$ can be used as $\varphi_{ckCaller}$ when $\mathcal{P}$ is the set of permissions protecting a component, or as $\varphi_{ckCallee}$ by components that want to make sure that their sensitive broadcasts are not broadcast too widely. To define $\varphi_{\mathcal{P}}$, 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. Then, we can define $\varphi_{\mathcal{P}}$ as follows.

$$\varphi_{\mathcal{P}} \triangleq f(iC, \Sigma) = \mathcal{P} \subseteq \mathit{grantedByUsrPerm}(iC, \Sigma)$$

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

We define function $\mathit{URIPerm}(iC, \Sigma)$ returns the URI permissions dynamically granted to iC; $\mathit{URIPerm}$ in turn relies on a data structure $\mathcal{M}$ to track the URI permissions granted to each application. Then, we define $\varphi_{\mathcal{P}}^{uri}$ as follows.

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

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

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

[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.

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.

$$\textsc{grant-tmp-t}\quad (\Sigma; \mathcal{E}, n :: \mathsf{grant}\ iC_1\ iC_2\ P\ \mathcal{F}_{tmp}) \xrightarrow{\mathsf{ok}(n)} (\Sigma'; \mathcal{E}) \quad \text{if } \varphi^{uri}_{\{P\}}(iC_1, \Sigma) = \mathsf{true}$$
$$\text{where } \Sigma' = \mathit{updateGrant}(\Sigma, iC_1, iC_2, P, \mathcal{F}_{tmp})$$

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.

Revoking permissions Revocation in Android is coarse-grained: permissions can be revoked by their owner and by any applications that are granted the permissions at install time, and revocation indiscriminately removes the permission from all entities which had it. There is no way of revoking permission from just a specific application or component. We write $*$ to denote all run-time instances.

$$\textsc{revoke}\quad (\Sigma; \mathcal{E}, n :: \mathsf{revoke}\ iC_1\ *\ P) \xrightarrow{\mathsf{ok}(n)} (\Sigma'; \mathcal{E})$$
$$\text{if } \varphi_{\mathcal{P}}(iC_1, \Sigma) = \mathsf{true} \text{ where } \Sigma' = \mathit{updateRevoke}(\Sigma, *, P)$$

The function $\mathit{updateRevoke}(\Sigma, *, P)$ removes all entries related to P from $\mathcal{M}$, and removes P from the granted permission set of all run-time components.

Exit and Reboot When a component exits, the URI permissions that it was granted are removed.

$$\textsc{exit}\quad (\Sigma; \mathcal{E}, n :: \mathsf{exit}\ iC) \xrightarrow{\mathsf{ok}(n)} (\Sigma', \mathcal{E}) \quad \text{where } \Sigma' = \mathit{updateExit}(\Sigma, iC)$$

updateExit performs two tasks: it removes the run-time instance from the state; and removes from $\mathcal{M}$ the mapping from the name of the enclosing application of iC to a permission and flag pair $(P, \mathcal{F}_{tmp})$ for all permissions P that were granted to iC. Note that it does not remove permissions associated with the permanent flag $\mathcal{F}_{prm}$.

When the system reboots, all the run-time components are deleted, and the auxiliary data structure is reset to be an empty map.

$$\textsc{reboot}\quad ((\mathcal{M}, \widehat{C}_1, \cdots, \widehat{C}_n, i\widehat{C}_1, \cdots, i\widehat{C}_n); \mathcal{E}, n :: \mathsf{reboot}) \xrightarrow{\mathsf{ok}(n)} ((\emptyset, \widehat{C}_1, \cdots, \widehat{C}_n); \emptyset)$$

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.*

1. *A owns P or A is granted P by the user; or*
2. *(a) there exists a delegation chain that ends with A s.t. for all B in the chain (where $B \neq A$):*
    - *i. B has P, and*
    - *ii. A satisfies the scope and lifetime constraints imposed by B;*

   (b) and P has not been revoked from A.

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 Appendix A), 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 [23, 17, 8].

Examining Android in light of these properties also revealed several implementation bugs (see Appendix A), which we reported to Google.

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, 21, 13, 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.

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

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

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 disallow-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 allow-declassify(P_1, P_2) to declassify sensitive information, i.e., to escape the restriction imposed by disallow-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.

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 describe Sorbet as an instantiation of the abstract model.

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.

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

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. $\mathit{updFlow}(\mathcal{N}, iC, Fl)$ returns a new mapping $\mathcal{N}'$, where $\mathcal{N}'(iC) = \mathcal{N}(iC) \cup Fl$. $\mathit{updDeclassify}(\mathcal{N}, iC, \mathsf{allow\text{-}declassify}(P_1, P_2))$ 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 $\mathcal{N}''' = \mathit{updDeclassify}(\mathcal{N}'', iC_2, \mathit{getDeclassify}(iC_2))$.

Since Android cannot mediate all communication between components within the same application, a cross-application call needs to conservatively assume that components within an application have communicated. Hence, we treat such 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 $\mathit{guardPA}(\Sigma, iC)$ 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 $\mathit{guardPA}$ takes the place of guardP. This means that if any component in an application has accessed private data protected by $\mathsf{disallow\text{-}flow}(P_1, P_2)$, 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.

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:

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

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.

Given the definitions of $\mathcal{T}$ and $\mathcal{T}_S$, we define functions $\mathit{checkPF}(\mathcal{T}_S, P)$ and $\mathit{checkP}(\mathcal{T}, P)$ recursively. $\mathit{checkPF}(\mathcal{T}_S, P)$ returns true if for each individual call stack in $\mathcal{T}_S$, it is the case that 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. Similarly, $\mathit{checkP}(\mathcal{T}, P)$ returns true if the above condition holds for the call tree $\mathcal{T}$. We show the definitions below.

$$\frac{}{\mathit{checkPF}([], P) = \mathsf{true}} \qquad \frac{\mathit{checkPF}(\mathcal{T}_S, P) = \mathsf{true} \qquad \mathit{checkP}(\mathcal{T}, P) = \mathsf{true}}{\mathit{checkPF}(\mathcal{T}_S @ [\mathcal{T}], P) = \mathsf{true}}$$

$$\frac{P \in P_C \qquad \text{and } \mathit{checkPF}(\mathcal{T}_S, P) = \mathsf{true} \qquad \text{or } \hat{P} \in P_C}{\mathit{checkP}((\mathcal{T}_S, (iC, P_C)), P) = \mathsf{true}}$$

We define a function $\mathit{PermOf}(iC, \Sigma)$ to return the set of permissions of a run-time component iC in state Σ. We define $\mathcal{MT}(iC, \Sigma)$ to return the call tree rooted at iC. $\mathcal{MT}(iC, \Sigma) = (\mathcal{MT}_S(iC), (iC, \mathit{PermOf}(iC, \Sigma)))$. We define the rule for a successful call below.

$$\begin{array}{ll} \textsc{call-t} & (\Sigma; \mathcal{E}, n :: \mathsf{call}\ iC_1\ iC_2\ I) \xrightarrow{\mathsf{ok}(n)} (\mathit{updateCall}(\Sigma, \mathsf{call}\ iC_1\ iC_2\ I); \mathcal{E}) \\ & \quad \text{if } iC_2.\varphi_{ckCaller}(\Sigma, iC_1) = \mathsf{true},\ iC_1.\varphi_{ckCallee}(\Sigma, iC_2) = \mathsf{true}, \\ & \quad \text{and } \forall P \in \mathit{guardP}(\Sigma, iC_2),\ \mathit{checkP}(\mathcal{MT}(iC_1, \Sigma), P) = \mathsf{true} \end{array}$$

The last line contains the additional checks for potential privilege escalation. The call will succeed only if all the components on the call stack have the permission P required to access the callee. We assume that checking whether C_1 has the permissions that C_2 requires is specified in $\varphi_{ckCaller}$.

If a call is successful, we need to update the call forest mapping $\mathcal{MT}_S$ of the callee. We define a function $\mathit{updCallF}(\mathcal{MT}_S, iC_1, iC_2)$ to update the call forest map when iC_1 calls or returns to iC_2. It will return a new map $\mathcal{MT}'_S$ derived from $\mathcal{MT}_S$ except that $\mathcal{MT}'_S(iC_2) = \mathcal{MT}_S(iC_2) @ [(\mathcal{MT}(iC_1, \Sigma))]$, where @ is the list-concatenation operation.

The *updateCall* function will call the *updCallF* function. As before, a return is treated as a call in the opposite direction. We omit the definitions here.

For cross-application calls, as with information-flow protection, we need to consider the call history of all components in the callee's application. We define $\mathcal{MT}_{SA}$ to return the concatenation of all the call trees $\mathcal{MT}(iC')$, where iC' and iC belong to the same application. We define $\mathcal{MT}_A(iC, \Sigma)$ to return the call tree rooted at iC and the call forest contains the call forest $\mathcal{MT}_A(iC, \Sigma)$: $\mathcal{MT}_A(iC, \Sigma) = (\mathcal{MT}_{SA}(iC) @ \mathcal{MT}_S(iC), (iC, \mathit{PermOf}(iC, \Sigma)))$. In the cross-application call, $\mathcal{MT}$ is replaced with $\mathcal{MT}_A$.

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 ($\mathsf{disallow\text{-}flow}$, $\mathsf{allow\text{-}declassify}$, and $\hat{P}$), and proof sketches of Prop. 6 and Prop. 5.

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* $\mathsf{disallow\text{-}flow}(P_1, P_2)$*, 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* $\mathsf{allow\text{-}declassify}(P_1, P_2)$*.*

We formally define a call graph G to capture all the call history as the system executes. Each node in the graph is a pair of a component's run-time instance iC, and relevant data structures Δ at the time the node is created. The edges of the graph capture all three kinds of information sharing: direct call via intent, instance sharing; and unmonitored communication between components from the same application.

Proof sketch of Prop. 5 We instantiate Δ in each node (C, Δ) of G as the tuple $(R, H, \mathcal{T}_S)$, where R is the set of permissions that C requires the caller to have, H is the set of permissions that C has, and $\mathcal{T}_S$ is a call forest such that $\mathcal{T}_S = \mathcal{MT}_S(C)$ at the time of the call.

Given an indirect call chain t in S_{AB} that ends in B, we construct the call graph G_B, which is the subgraph of G such that there is a path from every node to a node $v_B = (B, \Delta)$. By definition, all components that could have influenced the call to B are included in this graph. If G_B does not contain A then the call graph is also a valid call graph in system S_B, and Prop. 5 holds. Otherwise, let p_{AB} be the path from v_A to v_B, where $v_A = (A, \Delta_A)$. It must be the case that p_{AB} overlaps with t, since v_B is a shared node.

Let p_B be the longest suffix of p_{AB} such that all the nodes in p_B belong to the same application.

We first show that the call forest ($\mathcal{T}_S$) in the first node in p_B contains a path p'_{AB} that contains node v_A, and all the nodes in p_{AB} that have at least one direct neighbor in p_{AB} that belongs to a different application. By the definition of cross-application communication, all the edges in p'_{AB} are mediated by the activity manager. Such a path p'_{AB} exists because the rule for cross-application communication takes all the call forests of the components from the caller's application as the caller component's call forest, which in effect assumes that there is a direct call edge from every component in that application to the caller component.

Next we show by induction on the length of p_B that either there is a component between v_A and v_B that contains $\hat{P}$, or there exists a B' such that the edge between B' and B is unmonitored, and B' is not protected by P.

In the base case, when $|p_B| = 1$, B is the only node in p_B. By the reasoning above, we know that there exists p'_{AB} in the call forest of B such that A is the first node. By the definitions of rules CALL-T, CALL-F, RET-T and RET-F, if the call to B succeeded, then it must be the case that $\mathit{checkPF}(\mathcal{T}_S, P)$ returned true, where $\mathcal{T}_S$ was the call forest at node v_B. This means that there is a component C on the path p'_{AB} that has $\hat{P}$; otherwise the check would have failed. Hence, Prop. 5 holds.

In the inductive case, $|p_B| = n$. There are two subcases. In the first subcases, p_B does not contain any unmonitored edges. In this case, the call forest stored at B would have included p'_{AB}. Using similar reasoning as above, there is a component C on the path p'_{AB} that has $\hat{P}$; otherwise the check would have failed.

In the second subcase, let B' be the closest node to B on p_B such that the edge between B' and B is unmonitored. If B' is not protected by P, then conclusion holds. Otherwise, the length of the prefix of p_B ending with B' is smaller than n, and we can apply induction hypothesis directly.

Therefore, Prop. 5 holds.

Proof sketch of Prop. 6 Here the data structure Δ at each node stores forbidden permissions at the time the node is created. Given the call graph G, we can identify the subgraph G_B containing all the (indirect) call chains that reach B. If G_B does not contain A, then this subgraph is also valid in the system without A. Hence, the conclusion holds. If G_B contains A, there must be a path from A to B. In this case, we prove two statements inductively: (1) If there is an indirect call chain from A to X, X's enclosing application is XA, and some of the components in XA use P_2, then P_2 cannot be in the forbidden permission set of any of the components in XA on this path, and there must be a component that has $\mathsf{allow\text{-}declassify}(P_1, P_2)$ between A

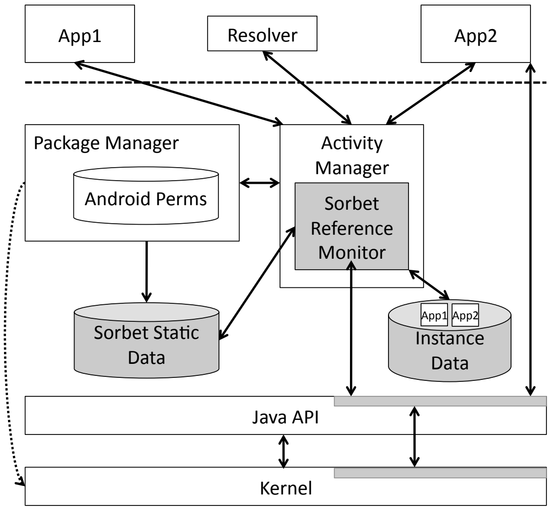

Figure 3: Sorbet architecture: Sorbet's additions to Android are shaded; arrows indicate interactions between components; and dashed arrows are present in Android but unneeded in Sorbet.

and X (not including X); (2) If there is an indirect call chain from A to X, X's enclosing application is XA, and P_2 is not in the forbidden permissions of any of the components in XA; then there must be a component between A and X (inclusive) that has allow-declassify(P_1, P_2). When proving these two statements, we apply mutual induction on the length of the trace. The main idea is that the unmonitored communication will not change the invariant of the conditions, since we make a uniform assumption about components in the same application. In proving (1), the rule for cross-application calls ensures that the intersection of the sets of permissions guarding components of the callee's application and the set of forbidden permissions of the caller's application is empty. This condition allows us to apply the I.H. of (2) right away.

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. disallow-flow(P_1, P_2) is specified at the component that is protected by P_1 by including android:forbiddenPermissions=["P_2"] in the list of permissions by which a 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.

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 $\widehat{\text{Internet}}$	✓	✓

Figure 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).

A particular challenge in implementing Sorbet was to capture operations that are 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 placed in the correct groups at installation time by the Package Manager. To mediate access to these operations, we used TOMOYO Linux [20], 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.

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.

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.

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

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 straightforward specification of useful policies that go 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 their manifest files, added sending functionality to some, and added a private content provider to Open Manager. The impact of Sorbet on performance 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., [19, 21]). Many works studied the attack surface against Android (e.g., [18]), including attacks using covert and overt channels [23], DoS [1] and web attacks [16], and unauthorized application repackaging [26].

Several works have pointed out flaws of the current Android 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 runtime and has the capacity to mitigate a wide range of privilege escalation attacks [2]. Our enforcement mechanism has many similarities to theirs. However, 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 for provenance tracking to mitigate the confused deputy problem [7]. Our goals are similar, though Sorbet's mechanism differs in several ways: we do not track full provenance information, but instead focus on flexible policy specification based on permissions, and we rely on the Android runtime for bookkeeping, rather than using digital signatures. Another proposal for mitigating unintended application collusion is through domain isolation. Bugiel et al. assigned trust levels to applications, allowing applications to communicate only if they are at the same level [3]. They focus on defining policy for a set of applications that consist a trust level, whereas we let applications define policy individually.

Several works have identified the problem of privacy leaks in Android [8, 23, 4, 9]. We provide a formal infrastructure which allows these and other flaws to be seen as violations of desired security properties. Projects such as TaintDroid [8] and AppFence [13] aim to automatically detect and prevent dangerous private information leaks in Android. Our work is in several ways complementary to these. 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 enforcement mechanism soundly enforces desired high-level security properties.

Formal analysis of Android-related security issues has received less attention. Shin et al. [24] 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 [25]. In contrast, our work develops a more abstract model suitable for reasoning about extensions to Android's permission system.

[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.

7 Conclusion

In addition to developing a framework for formally analyzing Android-style permission systems, this paper shows that it is possible 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 developing our system 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 component-level protection would be enhanced by stronger underlying abstractions.

Acknowledgments

This research was supported in part 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] M. M. A. Armando, A. Merlo and L. Verderame. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In *Proc. 27th IFIP International Information Security and Privacy Conference (SEC 2012)*, 2012.

[2] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. NDSS*, 2012.

[3] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *Proc. 1st ACM Workshop on Security and Privacy in Mobile Devices (SPSM)*, 2011.

[4] A. Chaudhuri. Language-based security on Android. In *PLAS Workshop*, 2009.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, 2011.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proc. ISC*, volume 6531 of *Lecture Notes in Computer Science*, 2010.

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

[8] W. Enck, P. Gilbert, B. gon 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. USENIX OSDI*, 2010.

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

[10] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *Proc. CCS*, 2009.

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

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

[13] 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. CCS*, 2011.

[14] A. Lineberry, D. L. Richardson, and T. Wyatt. These aren't the permissions you're looking for. `http://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf`, 2010. [accessed 10-Apr-2012].

[15] J. Loftus. DefCon dings reveal Google product security risks. `http://gizmodo.com/5828478/defcon-dings-reveal-google-product-security-risks`, 2011. [accessed 10-Apr-2012].

[16] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In *Proc. ACSAC '11*, 2011.

[17] C. Marforio, A. Francillon, and S. Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, April 2011.

[18] A. Mylonas, S. Dritsas, B. Tsoumas, and D. Gritzalis. Smartphone security evaluation - the malware attack case . In *SECRYPT'11*, 2011.

[19] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. ASIACCS*, 2010.

[20] NTT Data Corporation. TOMOYO Linux. `http://tomoyo.sourceforge.jp/`, 2012. [accessed 10-Apr-2012].

[21] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel. Semantically rich application-centric security in Android. In *Annual Computer Security Applications Conference*, 2009.

[22] P. Passeri. One year of Android malware (full list). `https://paulsparrows.wordpress.com/2011/08/11/one-year-of-android-malware-full-list/`, 2011. [accessed 10-Apr-2012].

[23] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proc. NDSS*, 2011.

[24] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A formal model to analyze the permission authorization and enforcement in the Android framework. In *SocialCom/PASSAT*, 2010.

[25] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the Android permission scheme. *IEEE Workshop on Policies for Distributed Systems and Networks*, 2010.

[26] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proc. CODASPY '12*, 2012.

Appendix

A Flaws Discovered in Android

To the best of our knowledge only Flaw 4 had previously been reported.

Flaws related to delegation and revocation

FLAW 1: A component that has gained temporary permission can redelegate the permanent version of the permission to another application, or itself.

For example, a JPEG viewer given temporary permission to read an email attachment is able to grant itself permanent permission to read the attachment.

FLAW 2: The revocation function revokeURIPermission revokes a specific URI permission from all components that were dynamically granted this permission. It can be called by components that own the permission or were granted the permission at install time. This may lead to confusion.

Consider the scenario where the Contacts application delegates a URI permission to the email client in order to send a contact through email. Before the email client completes this request, another application that has static permission (permission.READ_CONTACTS) to read the Contacts application's content provider can revoke the permission from the email client. This is an unexpected behavior for both the Contacts application and the email application and can cause the email application to crash.

FLAW 3: If a component protected with permission P is initiated by the Resolver activity then its caller is not constrained to have permission P.

The Resolver is a system activity that presents the user with a list of applications that can complete an action. For example, if the SEND action can be performed by multiple applications—like email, messaging and social network applications—the Resolver shows all of them to the user and the user chooses which one to use. However, if any of these activities was protected by permission P, and a call that will cause that activity to start is intercepted by the Resolver, the caller is not constrained to have P.

We speculate that this behavior is allowed because the user's action of selecting the target for the call implies the user's consent to the call even if the caller does not have the required permission. In this situation the user is being asked, without this being made explicit, to make a policy decision that may conflict with his install-time decision to allow only specific applications access to a protected component. This behavior of the Resolver is not documented, and may lead to a vulnerability if the user is not careful enough.

Flaws due to implementation error Along with the high-level correctness properties of §3.1, the implementation should obey several low-level correctness properties. The abstract model has helped us to identify these properties, and prompted us to examine the Android implementation and discover several flaws.

The first correctness property concerns the naming scheme for permissions: if permissions P and P' are deemed the same by the implementation, then they must be the same permission. In Android, permissions are represented as strings, and any two permissions with the same name string are treated as equivalent, even if they belong to unrelated applications. We will show vulnerabilities partially caused by this later.

The second correctness property requires system state to be properly updated when components exit or are uninstalled. More concretely, when an application declaring a permission P is uninstalled, P should be revoked from other applications that use P. Similarly, when an application containing a content provider is uninstalled, any permissions for accessing this content provider should be revoked. Android does not implement these clean-up functions properly. Combined with the inadequate identification of permissions, this results in the following vulnerabilities.

FLAW 4: Permissions declared by uninstalled applications are not revoked.

This leads to a potential attack previously identified by Shin et al. [25]. Suppose that a user installs a malicious application A that declares the permission P, and a malicious application B that uses it. Afterwards, the user uninstalls A and installs an innocent application that declares a permission with the same name. Now, the malicious application B can access the innocent application, as it has already been granted the permission. This scenario is particularly problematic because an attacker can easily target well-known applications.

This flaw that enabled the attack was claimed to have been addressed [25], but we were still able to reproduce it in Android 2.3.7.

FLAW 5: When an application A containing a content provider is uninstalled, any dynamically granted URI permissions for accessing this content provider are not revoked.

If the application A is reinstalled, it can be accessed by the applications that were previously granted these permissions. Additionally, URI permissions may refer to rows inside database tables, which they do by row index. Even if application A is willing to share the same data with the same applications after being reinstalled, old permissions that include row indices may now point to non-existent rows or rows that hold different data. This can cause confusion on the part of the permission holder, and can lead to an unintended information leak by the content provider.

Flaw 6: When an application that holds a URI permission is uninstalled, the permission is not revoked; if the application is reinstalled it still holds the URI permission. To decide whether a new application is the same as a previously uninstalled one, Android relies on the application's self-declared package name.

These two design decisions can be leveraged by an attacker to perform the following attack. Suppose that an application A grants permission to an innocent application B, which is then uninstalled. A malicious application C is then installed, but has the same (self-declared) package name as B. When it is installed, C gains all URI permissions that were granted to B.