Taming the Android Permissions System, by Typing

Michele Bugliesi, Stefano Calzavara, and Alvise Spanò

Università Ca' Foscari Venezia

Abstract. The widespread adoption of Android devices has attracted the attention of a growing computer security audience. Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed, mostly based on techniques from data-flow analysis, runtime protection mechanisms, or changes to the operating system. This paper complements this research by developing a framework for the analysis of Android applications based on typing techniques. We introduce a formal calculus for reasoning on the Android inter-component communication API and a type-and-effect system to statically detect privilege escalation attacks on well-typed components. Drawing on our abstract framework, we develop a prototype implementation of a type-checker for real Android applications, integrated with the Android Development Tools suite.

1 Introduction

Mobile phones have quickly evolved, over the past few years, from simple devices intended for phone calls and text messaging, to powerful handheld PDAs, hosting sophisticated applications that manage personal data and interact on-line to share information and access (security-sensitive) services.

This evolution has attracted the interest of a growing community of researchers on mobile phone security, and on Android security in particular. Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed. Originated with the seminal work in [13], a series of papers have developed techniques to ensure various system-level information-flow properties, by means of data-flow analysis [19], runtime detection mechanisms [11] and changes to the operating system [18]. Other papers have applied those same techniques in the study of application-level properties associated with Android's intent-based communication model and its interaction with the underlying permission system [9,6].

Somewhat surprisingly, typing techniques have instead received very limited attention, with few notable exceptions to date ([7], and more recently [3]). As a result, the potential extent and scope of type-based analysis has been so far left largely unexplored. In the present paper, we make a step towards filling this gap, by developing a calculus to reason on the Android inter-component communication API, and a typing system to statically analyze and control the interaction between intent-based communication and the underlying permission system.

Contributions. Our analysis of the Android permission system is targeted at the static detection of privilege escalation attacks, a vulnerability which exposes the Android platform to the risk of permission usage by unauthorized applications. Though the problem has been studied before [16,6], we are the first to devise a static detection technique. To carry out our study, we introduce λ-Perms, a simple formal calculus for reasoning about the Android inter-component interaction. Albeit small and abstract, λ-Perms captures all the relevant aspects of the Android message passing architecture and its relationships with the underlying permission system. Interestingly, our approach pays off, as it allows us to unveil subtle attack surfaces to the current Android implementation that had not been observed by previous work.

We tackle the problem of programmatically preventing privilege escalation attacks inside λ-Perms, by spelling out a formal definition of safety and proposing a sound security type system which statically enforces such notion, despite the best efforts of an unprivileged opponent. Our safety definition is inspired by run-time mechanisms proposed in earlier [16], but more compact and effective for formal reasoning. Enforcing the desired protection turns out to be challenging, because the inadvertent disclosure of sensible data may enable some typically overlooked privilege escalation scenarios. Given that an opponent may actively try to fool well-typed components into revealing secret data, our type system must deal with both secrecy and authenticity to be proven sound. Our type discipline then provides formal assurance about some secure communication guidelines proposed in [8].

Based on our forma framework, we then develop a prototype implementation of `Lintent`, a type-based analyzer integrated with the Android Development Tools suite (ADT). `Lintent` integrates our typing technique for privilege escalation detection within a full-fledged static analysis framework that includes intent type reconstruction, manifest permission analysis, and a suite of other actions directed towards assisting the programmer in writing more robust and reliable applications. Enhancing the Android development process is increasingly being recognized as an urgent need [8,15,12,24,10]: `Lintent` represents a first step in that direction.

Plan of the paper. Section 2 reviews the basics of the Android architecture. Section 3 introduces λ-Perms and discusses its relationships with Android. Section 4 describes privilege escalation attacks. Section 5 presents a type-and-effect system to enforce protection against such attacks. Section 6 describes `Lintent` and reports practical remarks. Section 7 discusses related work. Section 8 concludes.

The full version of the paper is available at `http://...` . The `Lintent` tool is available for download from the same URL[1]..

1. add the URL

2 Android Overview

We review the most important aspects of the Android architecture and its security model, thus providing the necessary ingredients to understand the technical contents of the paper.

Intents Once installed on a device, Android applications run isolated from each other in their own security sandbox. Data and functionality sharing among different applications is implemented through a message-passing paradigm built on top of *intents*, i.e., passive data structures providing an abstract description of an operation to be performed and the associated parameters. For instance, an application can send an intent to an image viewer, requesting to display a given JPEG file, to avoid the need of reimplementing such functionality from scratch.

The most interesting aspect of intents is that they can be used for both *explicit* and *implicit* communication. Explicit intents specify their intended receiver by name and are always securely delivered to it; since the identity of the recipient is typically unknown to developers of third-party applications, explicit intents are particularly useful for intra-application communication. Implicit intents, instead, do not mention any specific receiver and just require delivery to any application that supports a desired operation. Elaborating on the previous example, a developer may specify the string `ACTION_VIEW` as the recipient of an implicit intent, thus enabling any image viewer registered on that string to get the message and perform the task. Implicit intents facilitate runtime binding among different applications, but are more difficult to secure [8].

Components Intents are delivered to application *components*, the essential building blocks of Android applications. There are four different types of components, serving different purposes:

– An *activity* represents a screen with a user interface. Activities are started with an intent and possibly return a result upon termination;
– A *service* runs in the background to perform long-running computations and does not provide a user interface. Services can either be started with an intent or expose a remote method invocation interface to a client upon establishment of a long-standing connection;
– A *broadcast receiver* waits for intents sent to multiple applications. Broadcast receivers typically act as forwarders of system-wide broadcast messages to specific application components;
– A *content provider* manages a shared set of persistent application data. Content providers are not accessed through intents, but through a CRUD (Create-Read-Update-Delete) interface reminiscent of SQL databases.

We refer to the first three component types as "intent-based" components. Any communication among such components can employ either explicit or implicit intents.

Protection Mechanisms The Android security model implements isolation and privilege separation on top of a simple permission system. Permissions are used both to secure (implicit) inter-component communication and to access privileged methods of the API.

Android permissions are identified by strings and can be defined by either the operating system or the applications. Permissions are granted at installation time, application-wise, and are thus shared by all the components of the same application. All permissions are assigned a protection level:

– A *normal* permission is granted to any requesting application;
– A *dangerous* permission is granted to any requesting application, provided that the user provides explicit consent;
– A *signature* permission is granted only if the requesting application is signed with the same key as the application defining the permission;

– A *signature-or-system* permission lifts the previous restriction, by also allowing a limited set of system applications to acquire the permission.

If any of the requested permissions is not assigned, the application is not installed. Permission checks may fail at runtime, whenever the granted permissions do not suffice to perform a privileged operation, leading to security exceptions.

The Android communication API offers various protection mechanisms to the different component types. In particular, all components may declare permissions which must be owned by other components requesting access; on the other hand, only by broadcasting a request one may specify permissions which a receiver must hold to handle the intent. This implies, for instance, that a programmer cannot restrict the set of receivers when invoking the method `startActivity` with an implicit intent.

3 λ-`Perms`: a calculus for the Android permission system

We introduce λ-`Perms`, a simple formal calculus which captures the essence of inter-component communication in Android. We detail the connections between λ-`Perms` and the Android platform in Section 3.2.

3.1 Syntax and Semantics

We presuppose disjoint collections of names m, n and variables x, y, z, and use the meta-variables u, v to range over *values*, i.e., both names and variables. We denote permissions with typewriter capital letters, as in $\mathtt{PERMS}$, and assume they form a complete lattice with partial order $\sqsubseteq$, top and bottom elements $\top$ and $\bot$ respectively, and join and meet operators $\sqcup$ and $\sqcap$.

An *expression* represents a sequential program, which runs with a given set of assigned permissions and may return a value. As part of its computation, an expression may perform function calls from a pool of *function definitions*, i.e., named expressions ready to input an argument and run. The syntax of expressions is reported in Table 1.

$D ::=$	*Definitions*
$\mathsf{def}\ u = \lambda(x \triangleleft \mathtt{CALL}).E$	Function definition (scope of x is E)
$D \wedge D$	Conjunction
$E ::=$	*Expressions*
$D \setminus E$	Evaluation
$\overline{u}\langle v \triangleright \mathtt{RECV}\rangle$	Function invocation
$\mathsf{let}\ x = E\ \mathsf{in}\ E'$	Let (scope of x is E')
$(\nu n)\, E$	Restriction (scope of n is E)
$[\mathtt{PERMS}]\, E$	Permissions assignment
v	Value

Table 1. Syntax of expressions

$D \setminus E$ runs expression E in the pool of function definitions D. $\overline{u}\langle v \triangleright \texttt{RECV}\rangle$ tries to call function u, supplying v as an argument; the invocation succeeds only if the callee has at least permissions RECV. $\mathsf{let}\ x = E\ \mathsf{in}\ E'$ evaluates E to a name n and then behaves as E' with x substituted by n. $(\nu n)\, E$ creates a fresh name n and then behaves as E. $[\texttt{PERMS}]\, E$ represents E running with permissions PERMS. $\mathsf{def}\ u = \lambda(x \triangleleft \texttt{CALL}).E$ defines a function u: only callers with at least permissions CALL can invoke this function, supplying an argument for x. Multiple function definitions can be combined into a pool with the $\wedge$ operator. Function abstractions, "let" and ν are binding operators for variables and names, respectively: the notions of free names *fn* and free variables *fv* arise as expected, according to the scope defined in Table 1.

The formal semantics of λ−`Perms` is given by the small-step reduction relation $E \rightsquigarrow E'$ defined in Table 2. Reduction contexts $\mathcal{C}[\cdot]$ are defined as follows:

$$\mathcal{C}[\cdot] ::= \ \cdot \ \mid\ \mathsf{let}\ x = \mathcal{C}[\cdot]\ \mathsf{in}\ E \ \mid\ (\nu n)\, \mathcal{C}[\cdot] \ \mid\ D \setminus \mathcal{C}[\cdot]$$

Notice that permission assignments do not constitute a reduction context: indeed, although the syntax of expressions is liberal, such constructs are not intended to be nested.

(R-CALL)

$$\frac{\texttt{CALL} \sqsubseteq \texttt{PERMS} \qquad \texttt{RECV} \sqsubseteq \texttt{PERMS}'}{\mathsf{def}\ n = \lambda(x \triangleleft \texttt{CALL}).[\texttt{PERMS}']\, E \setminus [\texttt{PERMS}]\, \overline{n}\langle m \triangleright \texttt{RECV}\rangle \rightsquigarrow [\texttt{PERMS}']\, E\{m/x\}}$$

(R-RETURN)

$$\mathsf{let}\ x = [\texttt{PERMS}]\, n\ \mathsf{in}\ E \rightsquigarrow E\{n/x\}$$

(R-CONTEXT)

$$\frac{E \rightsquigarrow E'}{\mathcal{C}[E] \rightsquigarrow \mathcal{C}[E']}$$

(R-STRUCT)

$$\frac{E \Rrightarrow E_1 \qquad E_1 \rightsquigarrow E_2 \qquad E_2 \Rrightarrow E'}{E \rightsquigarrow E'}$$

Table 2. Reduction

(R-CALL) implements the security "cross-check" between caller and callee, which we discussed earlier: if either the caller is not assigned permissions CALL, or the callee is not granted permissions RECV, then the function invocation fails. Whenever the invocation is successful, the expression runs with the permissions of the callee. The other rules are essentially standard: (R-RETURN) allows the execution to proceed after complete evaluation to a name n of an expression $[\texttt{PERMS}]\, E$ inside the reduction context of a let; (R-CONTEXT) states that the reduction relation is contextual; (R-STRUCT) closes reduction under *heating*, an asymmetric version of structural congruence, which we define as the smallest preorder closed under the rules in Table 3. We write $E \equiv E'$ if and only if $E \Rrightarrow E'$ and $E' \Rrightarrow E$.

(H-EXTR-1) and (H-EXTR-2) formalize scope extrusion, much in the same spirit as in the pi-calculus. (H-FLIP-1) and (H-FLIP-2) perform some house-keeping needed

(H-CONTEXT)
$$\frac{E \Rrightarrow E'}{\mathcal{C}[E] \Rrightarrow \mathcal{C}[E']}$$

(H-EXTR-1)
$$\frac{n \notin \mathit{fn}(E')}{\mathsf{let}\ x = (\nu n)\, E\ \mathsf{in}\ E' \Rrightarrow (\nu n)\, (\mathsf{let}\ x = E\ \mathsf{in}\ E')}$$

(H-EXTR-2)
$$\frac{n \notin \mathit{fn}(D)}{D \setminus (\nu n)\, E \Rrightarrow (\nu n)\, (D \setminus E)}$$

(H-FLIP-1)
$$[\mathtt{PERMS}]\, (\nu n)\, E \Rrightarrow (\nu n)\, [\mathtt{PERMS}]\, E$$

(H-FLIP-2)
$$[\mathtt{PERMS}]\, (D \setminus E) \Rrightarrow D \setminus [\mathtt{PERMS}]\, E$$

(H-COMM)
$$(D_1 \wedge D_2) \setminus E \equiv (D_2 \wedge D_1) \setminus E$$

(H-ASSOC)
$$(D_1 \wedge D_2) \wedge D_3 \setminus E \equiv D_1 \wedge (D_2 \wedge D_3) \setminus E$$

(H-CONJ)
$$D_1 \setminus (D_2 \setminus E) \equiv (D_1 \wedge D_2) \setminus E$$

(H-MOVE)
$$D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') \equiv \mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E'$$

(H-DISTR)
$$[\mathtt{PERMS}]\ \mathsf{let}\ x = E\ \mathsf{in}\ E' \Rrightarrow \mathsf{let}\ x = [\mathtt{PERMS}]\, E\ \mathsf{in}\ [\mathtt{PERMS}]\, E'$$

Table 3. Heating

to export new names and functions dynamically created by a running expression. (H-COMM) and (H-ASSOC) are used in combination with (H-CONJ) to liberally rearrange a pool of function definitions. (H-MOVE) is needed both to perform function calls inside the reduction context of a let expression (when read from left to right) and to export new function definitions (when read from right to left). (H-DISTR) is borrowed from [7]. (H-EXTR-1) and (H-MOVE) are adapted from the concurrent object calculus [22].

3.2 λ-`Perms` vs Android

Though λ-`Perms` is a small calculus, it is expressive enough to capture all the most important aspects of the Android platform of interest for our present concerns.

Intents. λ-`Perms` can encode both implicit and explicit intents. Communication in λ-`Perms` is non-deterministic, in that a function invocation $\overline{u}\langle v \triangleright \mathtt{RECV}\rangle$ can trigger any function definition def $u = \lambda(x \triangleleft \mathtt{CALL}).E$ in the same scope, provided that all permission checks are satisfied. Technically, this non-determinism is enforced by the heating relation in Table 3, hence communication in λ-`Perms` naturally accounts for implicit intents, which represent the most interesting aspect of Android communication. Explicit intents can be recovered by univocally assigning each function definition with a distinct, unique permission: explicit communication is then encoded by requiring the callee to possess at least such permission.

Components. All of Android's intent-based active component types are represented in λ-`Perms` by means of function abstractions. Activities may be started through invocations to either `startActivity` or `startActivityForResult`; in λ-`Perms` we treat the two cases uniformly, by having functions return a result, which may simply be discarded by the caller. Services may either be started by `startService` or become the end-point of a long-running connection with a client through an invocation to `bindService`. The former behaviour is modelled directly in λ-`Perms` by a function call, while the latter is subtler and its encoding leads to some interesting findings (see below). Broadcast communication can be captured by a sequence of function invocations: this simple treatment suffices for security analysis. Finally, there is no λ-`Perms` counterpart of content providers, as they are passive entities, which are not accessed through a message-passing paradigm, but through a sophisticated CRUD interface reminiscent of SQL; hence, their security analysis is orthogonal to our setting.

Protection Mechanisms. λ-`Perms` is defined around a generic complete lattice of permissions. In Android this lattice is built over permission sets, with set inclusion as the underlying partial order. In our security analysis we collapse normal and dangerous permissions to $\bot$, since they do not provide any strong protection; all remaining permissions are intended to have signature(or system) protection, with different signatures and system permissions are represented by distinct (and incomparable) points in the lattice[2]. As to permission checking, the Android communication API only allows broadcast transmissions to be protected by permissions, namely requiring receivers to be granted specific privileges to get the message. Function invocation in λ-`Perms` just accounts for the more general behaviour available to broadcast transmissions, since unprotected communication can be simply encoded by specifying $\bot$ as the permission required to the callee, as in $\overline{u}\langle v \triangleright \bot\rangle$.

2. OK?

Binders. In Android a component can invoke `bindService` to establish a connection with a service and retrieve an `IBinder` object, which transparently dispatches method calls from the client to the remote service. This behavior is captured in λ-`Perms` by relying on its provision for dynamic component creation. To illustrate, let D contain the following service definition:

$$D \triangleq \mathsf{def}\ s = \lambda(x \triangleleft \mathtt{C}).[\mathtt{P}]\ (\nu b)\ (\mathsf{def}\ b = \lambda(y \triangleleft \bot).[\mathtt{P}]\ \ldots \setminus b) \tag{1}$$

and consider the λ-`Perms` encoding of a component binding to service s:

$$D \setminus [\mathtt{C}]\ \mathsf{let}\ z = \overline{s}\langle n \triangleright \bot\rangle\ \mathsf{in}\ \ldots$$

Service s runs with permissions `P` and requires permissions `C` to establish a connection. When a connection is successfully established, the service returns a fresh binder b, encoded as a function granted the same permissions `P` as s. The example unveils a subtle, and potentially dangerous, behaviour of the current Android implementation of `IBinder`'s: notice in particular that the function b may be invoked with no constraint, even though binding to s was protected by permissions `C`. In Android's current implementation, in fact, the permissions checks made when binding to a service are not repeated upon method invocations over the returned `IBinder` object; we find this implementation potentially dangerous, since it is exposed to privilege escalation attacks when binders are disclosed inadvertently.

Pending intents and delegation. Android introduces a form of delegation to relax the tight restrictions imposed by permissions checking. The mechanism is implemented through special objects known as *pending intents*: "by giving a `PendingIntent` to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity)" [21]. This informal description perfectly fits the previous encoding of binders in λ-`Perms`, in that any component exposed to the binder b is allowed to invoke the corresponding function running with permissions `P`, hence pending intents can be modelled in the very same way as binders, and are exposed to the same weaknesses whenever they are improperly disclosed.

4 Privilege escalation, formally

Davi *et al.* first pointed out a conceptual weakness in the Android permission system, showing that it is vulnerable to privilege escalation attacks [9]. The problem is best illustrated with an example. Consider three applications A, B and C, each consisting of a single component. Application A is granted no permission; application B, instead, is granted permission `P`, which is needed to access C. Apparently, data and requests from A should not be able to reach C; on the other hand, since B can freely be accessed from A, then it may possibly act as a proxy between A and C (see Figure 1 below).

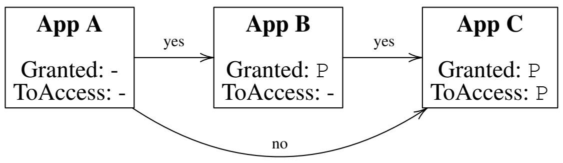

Fig. 1. Example of privilege escalation

Defining a formal notion of safety against privilege escalation attacks is an interesting task. We start from the IPC Inspection mechanism proposed by Felt *et al.* to dynamically prevent privilege escalation attacks on Android [16]. The idea behind IPC Inspection is remarkably simple: when an application receives a message from another application, a centralized runtime reference monitor lowers the privileges of the recipient to the intersection of the privileges of the two interacting applications. Since a patched Android system implementing IPC Inspection is protected against privilege escalation attacks "by design", our proposal is to consider such a system as a reference specification and state an equivalence-based notion of safety on top of it. Intuitively, an expression E is safe against privilege escalation attacks when its execution is completely oblivious of the fact that IPC Inspection is enabled or not.

Formally, let $E \rightsquigarrow_{\text{spec}} E'$ be the reduction relation obtained from the rules in Table 2 by substituting each occurrence of the symbol $\rightsquigarrow$ with the symbol $\rightsquigarrow_{\text{spec}}$ and by replacing the rule (R-CALL) with the new rule (R-CALL-SPEC) defined below:

(R-CALL-SPEC)

$$\frac{\texttt{RECV} \sqsubseteq \texttt{PERMS}' \qquad \texttt{CALL} \sqsubseteq \texttt{PERMS}}{\mathsf{def}\ n = \lambda(x \triangleleft \texttt{CALL}).[\texttt{PERMS}']\, E \setminus [\texttt{PERMS}]\, \overline{n}\langle m \triangleright \texttt{RECV}\rangle \rightsquigarrow_{\text{spec}} [\texttt{PERMS} \sqcap \texttt{PERMS}']\, E\{m/x\}}$$

The new reduction relation $\rightsquigarrow_{\text{spec}}$ formalizes inter-component communication in an Android system patched to support IPC Inspection.

Definition 1 (Simulation). *A binary relation $\mathcal{R}$ is a* simulation *if and only if, for any pair of expressions E_1, E_2 such that $E_1 \mathcal{R} E_2$, whenever $E_1 \rightsquigarrow E_1'$ we have $E_2 \rightsquigarrow_{spec} E_2'$ and $E_1' \mathcal{R} E_2'$. We say that E_1 is* simulated *by E_2 (written $E_1 \preccurlyeq E_2$) if and only if there exists a simulation $\mathcal{R}$ such that $E_1 \mathcal{R} E_2$.*

Given Definition 1, our notion of safety is immediate.

Definition 2 (Safety). *An expression E is* safe *against privilege escalation attacks if and only if $E \preccurlyeq E$.*

Although our definition draws inspiration from IPC Inspection, it clarifies an important aspect which was not previously discussed. Namely, we acknowledge that improper disclosure of some specific data, such as binders or pending intents, may lead to the development of applications which are unsafe according to Definition 2. Consider for instance the following adaptation of example (1): Reference later

$$D \triangleq \mathsf{def}\ s = \lambda(x \triangleleft \bot).[\texttt{P}]\, (\nu b)\, (\mathsf{def}\ b = \lambda(y \triangleleft \bot).[\texttt{P}]\, \overline{a}\langle y \triangleright \bot\rangle \setminus b) \tag{2}$$

and consider an unprivileged component interacting with s:

$$(\mathsf{def}\ a = \lambda(x \triangleleft \texttt{P}).[\texttt{P}]\, E) \wedge D \setminus [\bot]\ \mathsf{let}\ z = \overline{s}\langle n \triangleright \bot\rangle\ \mathsf{in}\ \overline{z}\langle n \triangleright \bot\rangle$$

Service s can be freely invoked by the unprivileged component, but it returns a pending intent b, which grants access to the component a protected by permissions $\texttt{P}$. As such, the system does allow to escalate privileges and maliciously supply arguments to the privileged component a through the pending intent b.

Being equivalence-based, our notion of safety is already a rather strong property, but we target a more ambitious goal: we desire protection despite the best efforts of an arbitrary opponent. In our model an opponent is a malicious, but unprivileged, Android application installed on the same device.

Definition 3 (Opponent). *A function definition O is an* opponent *if and only if each type annotation within O is Un and each permission assignment within O is $\bot$.*

The restriction on the type annotations is a standard technical device, which does not constrain the behaviour of opponents, as we discuss in Section 5. We conclude this section with the definition of *robust* safety, which is our true property of interest.

Definition 4 (Robust Safety). *An expression E is* robustly safe *against privilege escalation if and only if $O \setminus E$ is safe against privilege escalation for all opponents O.*

A very recent paper by Fragkaki et al. proposes a formal definition of protection against privilege escalation attacks inspired by the classic notion of non-interference for information flow control [18]. Their definition essentially demands that any call chain ending in a "high" (permission-protected) component exists in a system only if it exists in a variant of same system, where the "low" (unprivileged) components have been pruned away. We conjecture that their definition is equivalent to ours, but we do not have a formal equivalence result at the time of writing.

5 Preventing privilege escalation, by types and effects

We present a static type-and-effect system which allows to enforce robust protection against privilege escalation attacks. Designing a sound type discipline is subtle, mainly due to the presence of sensitive data like binders and pending intents, which the opponent may actively try to get under its control by deceiving well-typed components.

Types and Typing Environments We consider a minimal syntax for types, given below.

$$\tau ::= \mathsf{Un} \mid \mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$$

Type Un is the base type, which is used both as a building block for function types and to encompass all the data which are under the control of the opponent. Types of the form $\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$ are inhabited by functions which input arguments of type τ and return results of type τ'. Functions with this type can be invoked only by callers which are granted at least permissions $\mathtt{CALL}$, and should only be disclosed to components running with at least permissions $\mathtt{SECR}$. We define the *secrecy level* of a type τ, written $\mathcal{L}(\tau)$, as expected, by having $\mathcal{L}(\mathsf{Un}) = \bot$ and $\mathcal{L}(\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}) = \mathtt{SECR}$.

A typing environment Γ is a finite map from values to types. The *domain* of a typing environment Γ, written $dom(\Gamma)$, is the set of the values on which Γ is defined.

Typing Values The typing rules for values are simple, and given below.

$$\text{(T-PROJ)}\quad \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \qquad\qquad \text{(T-PUB)}\quad \frac{\Gamma \vdash v : \tau \qquad \mathcal{L}(\tau) = \bot}{\Gamma \vdash v : \mathsf{Un}}$$

(T-PROJ) is standard, while (T-PUB) makes it possible to treat all public data as "untyped", since they may possibly be disclosed to the opponent. We discuss the type rules for opponent code in the next section.

Typing Expressions The typing rules for expressions are in Table 4. The main judgement $\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{PERMS}'$ is read as "expression E, running with permissions $\mathtt{PERMS}$, has type τ in Γ and exercises at most permissions $\mathtt{PERMS}$ throughout its execution". We also define an auxiliary judgement $\Gamma \vdash_{\mathtt{PERMS}} D$ to be read as "definition D, with granted permissions $\mathtt{PERMS}$, is well-formed in Γ". The two judgement forms are mutually dependent.

(T-DEF)

$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Fun}(\texttt{CALL}, \tau \to \tau')^{\texttt{SECR}} \\ \Gamma, x : \tau \vdash_{\texttt{PERMS}} E : \tau' \blacktriangleright \texttt{PERMS}' \qquad \texttt{PERMS}' \sqsubseteq \texttt{CALL} \sqcup \texttt{SECR} \\ \texttt{CALL} \sqcup \texttt{SECR} = \bot \Rightarrow \Gamma, x : \mathsf{Un} \vdash_{\texttt{PERMS}} E : \mathsf{Un} \blacktriangleright \bot \qquad x \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\texttt{PERMS}} \mathsf{def}\ u = \lambda(x \triangleleft \texttt{CALL}).E}$$

(T-CONJ)

$$\frac{\Gamma \vdash_{\texttt{PERMS}} D_1 \qquad \Gamma \vdash_{\texttt{PERMS}} D_2}{\Gamma \vdash_{\texttt{PERMS}} D_1 \wedge D_2}$$

(T-EVAL)

$$\frac{\Gamma \vdash_{\texttt{PERMS}} D \qquad \Gamma \vdash_{\texttt{PERMS}} E : \tau \blacktriangleright \texttt{PERMS}'}{\Gamma \vdash_{\texttt{PERMS}} D \setminus E : \tau \blacktriangleright \texttt{PERMS}'}$$

(T-CALL)

$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Fun}(\texttt{CALL}, \tau \to \tau')^{\texttt{SECR}} \qquad \Gamma \vdash v : \tau \\ \bot \sqsubset \texttt{RECV} \sqcup \texttt{SECR} \qquad \texttt{CALL} \sqcup \texttt{SECR} \sqsubseteq \texttt{PERMS}\end{array}}{\Gamma \vdash_{\texttt{PERMS}} \overline{u}\langle v \triangleright \texttt{RECV}\rangle : \tau' \blacktriangleright \texttt{CALL} \sqcup \texttt{SECR}}$$

(T-VAL)

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash_{\texttt{PERMS}} v : \tau \blacktriangleright \bot}$$

(T-FAIL)

$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Fun}(\texttt{CALL}, \tau \to \tau')^{\texttt{SECR}} \qquad \Gamma \vdash v : \tau'' \\ \texttt{RECV} \sqcup \texttt{SECR} = \bot \Rightarrow \mathcal{L}(\tau'') = \bot \\ \texttt{CALL} \not\sqsubseteq \texttt{PERMS}\end{array}}{\Gamma \vdash_{\texttt{PERMS}} \overline{u}\langle v \triangleright \texttt{RECV}\rangle : \mathsf{Un} \blacktriangleright \texttt{PERMS}}$$

(T-PERMS)

$$\frac{\begin{array}{c}\Gamma \vdash_{\texttt{PERMS}'} E : \tau \blacktriangleright \texttt{PERMS}'' \\ \texttt{PERMS}' \sqsubseteq \texttt{PERMS}\end{array}}{\Gamma \vdash_{\texttt{PERMS}} [\texttt{PERMS}']\, E : \tau \blacktriangleright \texttt{PERMS}''}$$

(T-LET)

$$\frac{\begin{array}{c}\Gamma \vdash_{\texttt{PERMS}} E : \tau \blacktriangleright \texttt{PERMS}' \\ \Gamma, x : \tau \vdash_{\texttt{PERMS}} E' : \tau' \blacktriangleright \texttt{PERMS}'' \qquad x \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\texttt{PERMS}} \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau' \blacktriangleright \texttt{PERMS}' \sqcup \texttt{PERMS}''}$$

(T-RESTR)

$$\frac{\begin{array}{c}\Gamma, n : \tau \vdash_{\texttt{PERMS}} E : \tau' \blacktriangleright \texttt{PERMS}' \\ n \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\texttt{PERMS}} (\nu n : \tau)\, E : \tau' \blacktriangleright \texttt{PERMS}'}$$

(T-DEF-UN)

$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Un} \\ \Gamma, x : \mathsf{Un} \vdash_{\bot} E : \mathsf{Un} \blacktriangleright \bot \\ x \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\texttt{PERMS}} \mathsf{def}\ u = \lambda(x \triangleleft \texttt{CALL}).E}$$

(T-CALL-UN)

$$\frac{\Gamma \vdash u : \mathsf{Un} \qquad \Gamma \vdash v : \mathsf{Un}}{\Gamma \vdash_{\bot} \overline{u}\langle v \triangleright \texttt{RECV}\rangle : \mathsf{Un} \blacktriangleright \bot}$$

Table 4. Typing rules for definitions and expressions

We first note that our effect system discriminates between *granted* permissions and *exercised* permissions. For instance, the expression:

$$\mathsf{def}\ a = \lambda(x \lhd \bot).[\mathtt{P}]\ \overline{b}\langle n \rhd \bot\rangle \setminus ...$$

could either be well-typed or not, even though the function a is publicly available, but runs with strong permissions $\mathtt{P}$. The crux here is if the permissions $\mathtt{P}$ are indeed necessary to perform the invocation to b or not. We take advantage of the information tracked by our effect system in a number of type rules, as well as to perform additional helpful checks in our tool (see Section 6). Below, we comment on the most interesting (aspects of the) rules.

We consider rule (T-DEF) first. The third condition is central to enforce protection against privilege escalation. Namely, invoking a function of type $\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$ requires both permissions $\mathtt{CALL}$, to pass the security runtime checks, and permissions $\mathtt{SECR}$, to learn the name of the function; this implies that $\mathtt{CALL} \sqcup \mathtt{SECR}$ is a lower bound for the permissions granted to any caller of the function. Therefore, if the permissions exercised by the function itself are bounded above by $\mathtt{CALL} \sqcup \mathtt{SECR}$, no caller can escalate privileges upon invocation. As a practical remark, recall that both binders and pending intents enable indiscriminate access to a given application component c, hence our type system forces to assign to such values a secrecy level which is at least as high as the permissions exercised by c, to prevent their inadvertent disclosure. For instance, in example (2), we would give b a type of the form $\mathsf{Fun}(\bot, \tau_b \rightarrow \tau_b')^{\mathtt{P}}$.

Continuing with rule (T-DEF), the fourth condition is needed to account for interactions with the opponent. Since a function of type $\mathsf{Fun}(\bot, \tau \rightarrow \tau')^{\bot}$ is public and can be invoked by anyone, the body of such function must be type-checked also under the assumption that the input parameter is provided by the opponent (with type Un). Of course, in such case no privilege must be exercised by the function. A similar treatment is enforced by security type systems including cryptography to handle asymmetric decryption, since messages encrypted under a public key may actually come from the opponent [17,2].

We now focus on rule (T-CALL). Its first two conditions are standard, while the third one is needed to rule out as ill-typed the invocation $\overline{u}\langle v \rhd \bot\rangle$ when u is public. This is a very subtle case, since function invocation is non-deterministic in λ-`Perms`, hence the previous call, which does not constrain at all the choice of the callee, may run either a function defined by the opponent or a piece of trusted code. In the first case we should consider Un as the return type, while in the second case we should expect some value of type τ'. It turns out that both choices are unsound: the first one could break the secrecy of the return value upon interaction with trusted code; the second one would give the strong type τ' to some tainted data returned by the opponent. The implication for the Android platform is that any call to `startActivityForResult` or to `bindService` should employ explicit intents to be deemed as well-typed.

The last condition of rule (T-CALL) is specifically designed to prevent privilege escalation attacks. Indeed, recall that a function of type $\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$ can exercise at most privileges $\mathtt{CALL} \sqcup \mathtt{SECR}$ by rule (T-DEF), hence it can be safely invoked only by a caller granted with at least permissions $\mathtt{PERMS} \sqsupseteq \mathtt{CALL} \sqcup \mathtt{SECR}$. This interplay between rules (T-CALL) and (T-DEF) implements a rely-guarantee mechanism common to the modular analysis performed by most type systems.

The opponent counterparts for rules (T-DEF) and (T-CALL) are rules (T-DEF-UN) and (T-CALL-UN) respectively. By using these rules, the opponent can define arbitrary new functions and invoke existing ones, completely disregarding the restrictions enforced by typing. These rules are needed only for technical reasons, namely allowing us to prove Theorem 2 below; as such, they are not included in our implementation.

Finally, we discuss rule (T-FAIL). This rule is tricky and it is not strictly needed for soundness, but just to make type-checking more precise. To illustrate, consider the invocation $\overline{u}\langle v \triangleright \mathtt{RECV}\rangle$ performed by a caller endowed with permissions $\mathtt{PERMS}$ and assume that u has type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$. We can distinguish two cases: either u is defined by trusted code through rule (T-DEF), or u is defined by the opponent using rule (T-DEF-UN). In the first case, the information $\mathtt{CALL}$ annotated on the function type is consistent with the runtime permission enforcement, thus, since $\mathtt{CALL} \not\sqsubseteq \mathtt{PERMS}$, we are guaranteed that the invocation will actually fail at runtime and we can give an arbitrary type τ'' to the argument v. Otherwise, suppose that u was defined by the opponent: in this case the invocation might actually take place, since the opponent can disregard the type of u. Anyway, if the invocation happens, we are guaranteed that $\mathtt{RECV} \sqcup \mathtt{SECR} \sqsubseteq \bot$, since the opponent has no privileges and learns only public data; we must then enforce the condition $\mathcal{L}(\tau'') \sqsubseteq \bot$ to protect the secrecy of the argument v. Note that, due to such a possible interaction with the opponent, the exercised permissions are conservatively assumed as $\mathtt{PERMS}$, i.e., all the permissions entitled to the caller.

We conclude the description of the type system with an important remark on expressiveness. Some of the constraints imposed by our typing rules are rather restrictive for practical use, but are central to enforcing the conditions of Definition 2 and its robust variant. Our implementation, however, features a number of escape hatches based on Java annotations to keep programming practical, much in spirit of the declassification/endorsement constructs customary to the information-flow literature [25]. We discuss this point further in Section 6.3.

Formal Results. We can prove that the previous type discipline enforces the expected security properties. The safety result below follows by a "simulation-aware" variant of a standard Subject Reduction theorem for our type system, which captures the step-by-step relationships between the standard semantics and our reference semantics. The proof relies on a co-inductive argument enabled by such theorem.

Theorem 1 (Type Safety). *If $\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{P}$, then $E \preccurlyeq E$.*

The next result states that our type system does not constrain the opponent in any way.

Lemma 1 (Opponent Typability). *Let O be an opponent and let $\Gamma \vdash u : \mathsf{Un}$ for all $u \in fnfv(O)$, then $\Gamma \vdash_{\mathtt{PERMS}} O$ for every $\mathtt{PERMS}$.*

By combining the two previous results, we can prove our main theorem.

Theorem 2 (Robust Safety). *Let $\mathcal{L}(\tau) = \bot$ for every u such that $\Gamma(u) = \tau$. If $\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{PERMS}'$, then E is robustly safe against privilege escalation attacks.*

6 Implementation

Our implementation is a tool (`Lintent`) designed as a plug-in for Android `Lint`, the official static analysis utility distributed within the Android Development Tools (ADT).

`Lintent` analyzes Java source code rather than bytecode because it is has been developed within a larger research project aimed at type-based verification techniques for Android applications. In principle, the same analysis could be performed on the bytecode, though reasoning about types at the bytecode level level is arguably more demanding than at source level [20].

The main highlights of `Lintent` may be summarized as follows.

ADT Lint Integration. Android `Lint` is a very useful ADT component, as it can detect a wide range of anomalies and defects within the source code and related meta-data (manifest file, resource files, etc.) that the Java compiler alone would not be able to spot out. `Lint` is very popular within the development community, therefore deploying our tool as a `Lint` plug-in appears to be the natural choice to ease a wide adoption.

Security Verification. The Java compiler is completely oblivious of the Android permission system, since all permission information is encoded in terms of string literals used within the Java code and declared in the manifest. `Lintent` performs a number of static checks over permissions usage, analyzing the application source code and the manifest permission declarations, and eventually warning the developer in case of potential attack surfaces for privilege escalation scenarios. As a byproduct of its analysis, `Lintent` is able to detect over-privileged or under-privileged applications, and suggest fixes.

Intent and Component Type Reconstruction. The typing of intents and component supported by the Java compiler is rather loose and uninformative: in fact, the Java type system does not keep track of any type information about either the contents of Intent objects or the data a component sends and expects to receive. This seriously hinders any form of type-based analysis, including the one discussed in the paper, and makes Android programming very error-prone. `Lintent` infers and records the types of data injected into and extracted out of intents while tracking the flow of inter-component message passing for reconstructing the incoming requests and outgoing results of each component. This is needed to prevent improper disclosure of binders or pending intents, but it proves helpful also to detect common programming errors related to misuse of intents [24].

6.1 Architecture

The `Lintent` architecture is described in Figure 2 below.

As anticipated, the tool is a `Lint` plug-in acting as a front-end for an engine program running as a separate process. The plug-in is written in Java and takes advantage of the built-in Java parser offered by `Lint`, which produces an Abstract Syntax Tree (AST) based on `Lombok JavaC AST` [27]. Once parsing ends successfully, the engine process is spawned and starts receiving data from a pipe formerly created by the

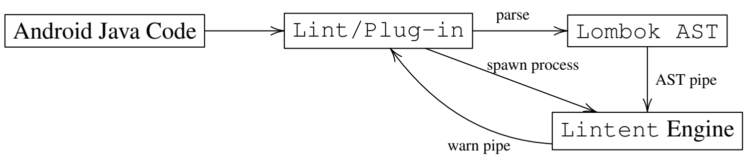

Fig. 2. `Lintent` architecture

plug-in itself for interprocess communication. Our plug-in AST visitor simply serializes the program tree through the pipe and then waits for feedback from the engine process, hanging on a second pipe aimed at receiving warnings and messages to be eventually shown as issues by the `Lint` UI. The engine program is written in F# and does the real job: after deserializing the input program tree acquired from the AST pipe, it creates its own custom representation of the AST and performs the analysis.

The first phase consists in reconstructing the types of intents and components by means of a hybrid type-inference/partial-evaluation algorithm; the second pass eventually checks permissions usage and validates security-related properties of the input program. Throughout the analysis, the engine communicates back with the `Lint` plug-in through the warn pipe, feeding back any issue worth to be prompted to the user.

6.2 Challenges

Analyzing Android applications is a complex and demanding activity, which involves a number of non-trivial inter-related tasks.

Detecting API Patterns. Implementing the rules from the abstract type system for λ-`Perms` requires a preliminary analysis to detect the corresponding patterns in the Android source code. The analysis is far from trivial given the complexity of the Android communication API, which offers various different patterns to implement inter-component communication. For example, the developer guide describes at least three different ways to implement bound services, with different degrees of complexity, and a local inspection of the instructions alone does not suffice to reconstruct enough information to support verification. Partial evaluation techniques combined with type inference are needed where syntactic pattern matching of code templates would be too naive.

Delocalized Information. Permissions in Android are meta-information which is not included in Java sources, but in the application Manifest file. This is an XML file containing, among other information, the permissions each application component requires for being accessed and what permissions are requested by the application itself. Several Android API calls require non-empty permission sets and must be detected and tracked by our tool. `Lintent` retrieves a set of mappings between API method signatures and permissions from a set of external files[1], which are thus updatable with no need to rebuild the tool. All this information is needed to implement our effect system and is central to type-checking.

[1] Currently such permission map files are those distributed along with Stowaway [14].

Type reconstruction. Arguably the hardest challenge arising during the implementation is related to a number of "untyped" programming conventions which are enabled by the current Android API. Consider, for instance, a simple scenario of intent usage with multiple data types:

```
class MySenderActivity extends Activity {
   static class MySer implements Serializable { ... }

   void mySenderMethod() {
      Intent i = new Intent(this, TargetActivity.class);
      i.putExtra("k1", 3);
      i.putExtra("k2", "some_string");
      i.putExtra("k3", new MySer());
      startActivityForResult(i, 0);
   }
}
```

Since the `putExtra` method is overloaded for different types, the type of the second argument of each call must be reconstructed in order to keep track of the actual type of the value bound to each key. On the recipient side, intent "extras" are retrieved by freely accessing the intent as it was a dictionary, so the receiver may actually retrieve data of unexpected type and fail at runtime, or disregard altogether some fields provided by the sender.

```
class MyRecipientActivity extends Activity {
    static class WrongSer implements Serializable { ... }

    void onCreate(Bundle savedInstanceState) {
        Intent i = getIntent()
        // run-time type error: k1 was an int!
        String k1 = i.getStringExtra("k1");
        // dynamic cast fails!
        WrongSer o = (WrongSer)i.getSerializableExtra("k3");
        // forgets to extract "k2": might be unwanted!
    }
}
```

The example highlights a total lack of static control over standard intents manipulation operations: with these premises, no type-based analysis can be soundly performed. For this reason, intents are treated in `Lintent` as record types of the form $\{k_1 : T_1, \ldots, k_n : T_n\}$, where k_i is a string constant and T_i is a Java type. This enforces a much stronger discipline on data passing between components - i.e., on the injection and extraction of "extras" into and from intents. Notably, the same type reconstruction applies to objects of type `Bundle` as well, and `Bundle` objects possibly put within Intents or other `Bundle`'s are recursively typed as sub-records. Our treatment is consistent with our type system, in that a function type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$ constrains the caller in providing an argument (i.e., an intent) of type τ and the callee in returning a result of type τ'. Enforcing the same discipline in Android applications is crucial for protecting the secrecy of binders and pending intents. As a byproduct of this analysis, our tool is able to warn the user in case of ill-typed or dangerous manipulations of the intent.

Partial Evaluation Recall from the previous discussion that every data an user puts into an intent must be bound to a key, hence an intent object can be thought as a dictionary of the form $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$. Unfortunately, the dictionary keys are run-time string objects and therefore plain expressions in Java – they are not first-class language identifiers. Whether they happen to be string literals or complex method calls computing a string object is irrelevant: in any case they belong to the run-time world. The very same problem arises for result codes and Intent constructor invocation: both the sender component and the recipient class object supplied as arguments could be results of computations, and the same holds true for action strings in case of implicit intent construction. Consider for instance the following code snippet:

```
static class Const {
    public static final label = "LABEL";
    public static final name = name + "_NAME";
    public static final age = name + "_AGE";
}

Class<?> getRecp() { return SomeClass.class; }

Activity getSelf() { return this; }

void mySenderMethod() {
    int code = 3;
    final String base = "PERSON_" + Const.label;
    Intent i = new Intent(getSelf(), getRecp());
    i.putExtra(base + name + 1, "John");
    i.putExtra(base + name + 2, "Smith");
    i.putExtra(base + name, 23);
    startActivityForResult(i, code);
}
```

Da qui a fine paragrafo va rilavorato oppure togliamo/accorciamo

The type representation of an intent therefore must be enriched with further static information: the result code value bound to it at call-time and the recipient component class object specified at construction-time, leading to something more than a record: $(|T_c|^R, m)\{k_1 : T_1..k_n : T_n\}$, where $|T_c|^R$ is the reifiable [2] type of the component c having type T_c, and m is an integer constant. Partial evaluation of Java expression is required for determining k_i keys, the n result code and the T_c component class object. We argue that in most cases programmers factor their code in such a way that key names, result codes and either explicit recipient component class objects or action strings for implicit intents are bound to language constants[3] or simple computations,

[2] Since some type information is erased during compilation, in Java not all types are available at run time. Types that are completely available at run-time are known as reifiable types [26].

[3] Defining `static final` attributes as constants is a pretty common practice in Android development. Should partial static evaluation fail in determining a constant literal or object

hence the need not to perform a full-featured data-flow analysis - which would not anyway provide more meaningful information in case of truly undecidable code such as loops etc.

Interaction with third party libraries. Typically applications rely on external libraries offering a number of services to the programmer. From the point of view of Java code, such libraries are collections of compiled classes linked into one or more `jar` files: their source code is therefore not available at analysis time. Import declarations on top of compilation units simply carry information on package names and class paths, but do not specify class member signatures or other details. Type resolution is a tricky task for a tool that does not have the same information the compiler is given by command line arguments, therefore types that are inferred as external must be treated in some special way: access to `jar` files must be granted to `Lintent` to let it inspect the contents of imported packages and classes.

6.3 Explicit Java annotations support

Starting from Java 1.5, Java programs can be enriched with additional meta-information known as *annotations*. We rely on Java annotations to provide a number of escape hatches from the tight discipline imposed by our type rules. Several privileged components intentionally expose functionalities, hence we define a special annotation of the form `@priv{endorse="P"}` to mark methods as `onCreate` with a set of permissions which can be dispensed by the type-checker. Namely, if the method exercises the permissions set Q, its containing component is deem as well-typed if it is protected with at least permissions $Q \setminus P$. A similar treatment is implemented for pending intents through the usage of the annotation `@priv{declassify="P"}`, which allows to reduce the secrecy level of such objects computed by our type-checker, and enable a controlled form of delegation.

6.4 Limitations and Extensions

At the moment the tool supports only activities and started services, while support for bound services is still under heavy development and in a very preliminary stage. We plan to identify calls to API methods as `checkCallingPermission` to make our static analysis more precise. We are also investigating the possibility of developing a frontend to a decompiler as `smali` [1] to support the analysis of third-party applications.

7 Related work

There exists a huge literature on Android application security, as recently reported in an interesting survey by Enck [10]. Below, we discuss the works most closely related to ours.

reference, a fallback unique value is calculated case by case according to contextual information such as the program location, the shape of the expression that couldn't be evaluated, ecc.

Android permissions. The deficiencies of the Android permission system with respect to privilege escalation attacks were first pointed out by Davi *et al.* [9]. The paper presents a proof of concept attack, but does not discuss any possible solution to the problem. Felt *et al.* instead propose a runtime mechanism called IPC inspection to provide protection against privilege escalation attacks on Android [16]. The solution is reminiscent of Java stack inspection and it inspired our definition of safety, as we discussed in Section 4. We find the implementation design very competent, but we also notice that IPC inspection may induce substantial performance overhead, since it requires keeping track of different application instances to make the protection mechanism precise, and avoid impacting heavily on the user's experience. In a more recent work, Bugiel *et al.* describe a fairly sophisticated runtime framework for enforcing protection against privilege escalation attacks on Android [6]. Notably, their solution comprises countermeasures also against colluding applications, which maliciously collaborate to escalate privileges, an aspect which is neglected by both IPC inspection and our type system. Providing such guarantees, however, requires a centralized solution built over low-level operating system mechanisms. We aim at being complementary to such proposal: enforcing runtime protection is fundamental against malicious applications which reach the market, while static analysis techniques can be helpful for well-meaning developers who desire to validate, and possibly certify, their code. Finally, Felt *et al.* propose Stowaway, a static analysis tool for detecting overprivilege in Android applications [15]. In our implementation we take advantage of their permission map, which relates API method calls to their required permissions.

Android communication. The threats related to the Android message-passing system were first studied by Chin *et al.* [8]. Their paper provides an interesting overview of the intent-based attack surfaces and discusses guidelines for secure communication. The authors provide also a tool, ComDroid, which is able to detect potential vulnerabilities in the usage of intents. However, the paper does not provide any formal guarantee about the effectiveness of the proposed secure communication guidelines; in our work, instead, we reason about intents usage in a formal calculus, hence we are able to confirm many of their findings as sound programming practices. ComDroid does not address the problem of detecting privilege escalation attacks. The robustness of inter-component communication in Android has been studied also by Maji *et al.* through the usage of fuzzy testing techniques, exposing some interesting findings [24]. Their empirical methodology, however, does not provide a clear understanding of the correct programming patterns for communication.

Formal models. λ-`Perms` is partially inspired by a core formal language proposed by Chaudhuri [7]. With respect to such formalism, however, λ-`Perms` provides a more thorough treatment of a number of Android peculiarities. First, it provides support for implicit communication and runtime registration of new components over action strings. Second, it introduces a scoping construct, which is useful to model both service binding and pending intents; more in general, a restriction operator in the style of process algebras typically proves useful for formal security reasoning. In later work, Fuchs *et al.* build on the calculus proposed by Chaudhuri to implement SCanDroid, a provably sound static checker of information-flow properties of Android applications [19]. As we mentioned, the work by Fragkaki *et al.* [18] proposes a formal definition of protection against privilege escalation attacks inspired to the classic notion of non-interference: we leave a formal comparison with our approach to our plans of future work. Shin *et al.* introduce a mechanized model of the Android permission system and validate some expected security properties using Coq [28]. Language support for privilege-based software systems has been studied by Jagadeesan *et al.* [23] and Braghin *et al.* [5].

8 Conclusions

We performed a first step towards supporting Android programmers with type-based analysis techniques, by providing a sound static analysis framework for detecting potential privilege escalation attacks. The implementation of our techniques identifies a number of challenges which are likely central to the practical development of any type-checker for Android applications. As a future work, we plan to study robust declassification and endorsement programming patterns in our formal framework, taking advantage from existing research results [4]. The on-going development of our tool, instead, will probably lead to some further interesting technological challenge.

References

1. Smali: An assembler/disassembler for android's dex format. `http://code.google.com/p/smali/`
2. Abadi, M., Blanchet, B.: Secrecy types for asymmetric communication. Theor. Comput. Sci. 3(298), 387–415 (2003)
3. Armando, A., Costa, G., Merlo, A.: Formal modeling and verification of the android security framework. In: TGC2012. pp. xx–xx (2012), to Appear
4. Askarov, A., Myers, A.: A semantic framework for declassification and endorsement. In: ESOP. pp. 64–84 (2010)
5. Braghin, C., Gorla, D., Sassone, V.: A distributed calculus for role-based access control. In: CSFW. pp. 48–60 (2004)
6. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: NDSS (2012), to appear
7. Chaudhuri, A.: Language-based security on Android. In: PLAS. pp. 1–7 (2009)
8. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys. pp. 239–252 (2011)
9. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: ISC. pp. 346–360 (2010)
10. Enck, W.: Defending users against smartphone apps: Techniques and future directions. In: ICISS. pp. 49–70 (2011)
11. Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI. pp. 393–407 (2010)
12. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: USENIX Security Symposium (2011)
13. Enck, W., Ongtang, M., McDaniel, P.D.: Understanding android security. IEEE Security & Privacy 7(1), 50–57 (2009)

14. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Stowaway - android permissions demystified. `http://www.android-permissions.org/`
15. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: ACM Conference on Computer and Communications Security. pp. 627–638 (2011)
16. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)
17. Focardi, R., Maffei, M.: Types for security protocols. Tech. Rep. CS-2010-3, University of Venice (2010), available at `http://www.lbs.cs.uni-saarland.de/resources/types-security.pdf`
18. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: ESORICS. Lecture Notes in Computer Science, vol. 7459, pp. 1–18 (2012), to appear.
19. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications (2009), Technical report, University of Maryland.
20. Gagnon, E., Hendren, L.J., Marceau, G.: Efficient inference of static types for java bytecode. In: SAS. pp. 199–219 (2000)
21. Google Inc: Reference documentation for `android.app.PendingIntent`. `http://developer.android.com/reference/android/app/PendingIntent.html`
22. Gordon, A.D., Hankin, P.D.: A concurrent object calculus: Reduction and typing. Electr. Notes Theor. Comput. Sci. 16(3), 248–264 (1998)
23. Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: Lambda-rbac: Programming with role-based access control. Logical Methods in Computer Science 4(1) (2008)
24. Maji, A.K., Arshad, F.A., Bagchi, S., Rellermeyer, J.S.: An empirical study of the robustness of inter-component communication in android. In: DSN. pp. 1–12 (2012)
25. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: POPL. pp. 228–241 (1999)
26. Oracle/Sun Microsystems: Java language reference. `http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.7`
27. Project Lombok: Reference documentation for `lombok.javac` abstract syntax tree. `http://projectlombok.org/api/lombok/javac/package-summary.html`
28. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the android framework. In: SocialCom/PASSAT. pp. 944–951 (2010)