Lintent: Towards Security Type-Checking of Android Applications

Michele Bugliesi, Stefano Calzavara, Alvise Spanò

Lintent: towards security type-checking of Android applications

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 through 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 prevent privilege escalation attacks on well-typed components. Drawing on our abstract framework, we develop a prototype implementation of `Lintent`, a security type-checker for Android applications integrated with the Android Development Tools suite. We finally discuss preliminary experiences with our tool, which highlight real attacks on existing applications.

1 Introduction

Mobile phones have quickly evolved 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 [9], a series of papers have developed techniques to ensure various system-level information-flow properties, by means of data-flow analysis [13], runtime detection mechanisms [7] and changes to the operating system [12]. Other papers have applied similar techniques to the study of the intent-based communication model of Android and its interaction with the underlying permission system [5,2]. Somewhat surprisingly, typing techniques have instead received very limited attention, with few notable exceptions to date ([3], and more recently [1]). 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.

Contributions. Our analysis of the Android platform is targeted at the static detection of privilege escalation attacks, a vulnerability which exposes the framework to the risk of unauthorized permission usage by malicious applications. To

carry out our study, we introduce π-`Perms`, a simple formal calculus for reasoning about inter-component interaction in Android (Section 3). Albeit small and abstract, π-`Perms` captures the most relevant aspects of the Android message passing architecture and its relationships with the underlying permission system. Our formalization pays off, as it allows us to unveil subtle attack surfaces to the current Android implementation that had not been evaluated before.

We tackle the problem of programmatically preventing privilege escalation attacks inside π-`Perms`, by spelling out a formal definition of safety (Section 4) and proposing a sound security type system which statically enforces such notion, despite the best efforts of an opponent (Section 5). Providing the desired protection turns out to be challenging, since the inadvertent disclosure of sensitive data may enable some typically overlooked privilege escalation scenarios.

Based on our formal framework, we then develop a prototype implementation of `Lintent`, a type-based analyzer integrated with the Android Development Tools suite (Section 6). `Lintent` integrates our typing technique for privilege escalation detection within a full-fledged static analysis framework aimed at supporting a robust and more reliable development process. `Lintent` is the first type-based analyzer for Android applications and its implementation highlights a number of engineering challenges which should likely be tackled by any other type-based verification tool for Android. We discuss preliminary experiences with our tool, which highlight real attacks on existing applications (Section 7).

Enhancing the Android development process is increasingly being recognized as an urgent need [4,10,8,16,6]: `Lintent` represents a first step in that direction[1].

2 Android Overview

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., asynchronous messages providing an abstract description of an operation to be performed. Intents may be either *explicit* or *implicit*: the former specify their intended receiver by name and are always securely delivered to it; the latter, instead, do not mention any specific receiver and just require delivery to any application that supports a given operation (an *action*).

Components. Intents are delivered to application *components*, the essential building blocks of Android applications. There are four different types of components. 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: services can either be started with an intent, or expose a remote method invocation interface to a client by returning it a *binder* object. A *broadcast receiver* waits for intents sent to multiple applications. A *content provider* manages a shared set of persistent application

[1] Technical report and `Lintent` at `https://github.com/alvisespano/lintent`

data. Content providers are not accessed through intents, but through a CRUD (Create-Read-Update-Delete) interface reminiscent of SQL.

Protection mechanisms. The Android security model implements isolation and privilege separation on top of a simple permission system. Android permissions are identified by strings and can be defined by either the operating system or the applications. Permissions are assigned at installation time and are shared by all the components of the same application; if any of the requested permissions is not granted by the user, the application is not installed. 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 to them; on the other hand, only broadcast requests may specify a permission which a receiver must hold to get the message. A limited form of permission delegation is implemented in Android by special objects known as *pending intents*: we will return to this point later on.

3 π-Perms: a calculus for Android applications

We describe π-Perms, a simple formal calculus which captures the essence of inter-component communication in Android. We detail the connections between π-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 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 $\sqcap$ respectively.

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*. The syntax of expressions is defined in Table 1.

$E ::=$	*expressions*	$D ::=$	*definitions*
$D \setminus E$	evaluation	$u(x \triangleleft \texttt{CALL}).E$	function def.
$\overline{u}\langle v \triangleright \texttt{RECV}\rangle$	invocation	$D \wedge D$	conjunction
$\mathsf{let}\ x = E\ \mathsf{in}\ E'$	let expr.		
$(\nu n)\, E$	restriction		
$[\texttt{PERMS}]\, E$	perm. assign.		
v	value		

Table 1. Syntax of π-Perms expressions

The expression $D \setminus E$ runs E in the pool of function definitions D. An invocation $\overline{u}\langle v \triangleright \texttt{RECV}\rangle$ tries to call function u, supplying v as an argument; the invocation succeeds only if the callee has at least permissions `RECV`. A let expression $\mathsf{let}\ x = E\ \mathsf{in}\ E'$ evaluates E to a name n and then behaves as E' with x substituted by n. A restriction $(\nu n)\, E$ creates a fresh name n and then behaves as E. The expression $[\texttt{PERMS}]\, E$ represents E running with permissions `PERMS`. A definition $u(x \triangleleft \texttt{CALL}).E$ introduces 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 definitions, "let" and ν are binding operators for variables and names, respectively: the notions of free names *fn* and free variables *fv* arise as expected.

The formal semantics of π-`Perms` is given by the small-step reduction relation $E \to E'$ defined in Table 2.

$$\text{(R-Call)}\quad \frac{\texttt{CALL} \sqsubseteq \texttt{PERMS} \qquad \texttt{RECV} \sqsubseteq \texttt{PERMS}'}{n(x \triangleleft \texttt{CALL}).[\texttt{PERMS}']\, E \setminus [\texttt{PERMS}]\, \overline{n}\langle m \triangleright \texttt{RECV}\rangle \to [\texttt{PERMS}']\, E\{m/x\}}$$

$$\text{(R-Return)}\quad \mathsf{let}\ x = [\texttt{PERMS}]\, n\ \mathsf{in}\ E \to E\{n/x\} \qquad \text{(R-Context)}\quad \frac{E \to E'}{\mathcal{C}[E] \to \mathcal{C}[E']} \qquad \text{(R-Struct)}\quad \frac{E \Rrightarrow E_1 \to E_2 \Rrightarrow E'}{E \to E'}$$

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

Table 2. Reduction semantics for π-`Perms`

Rule (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 invocation fails. Whenever the invocation is successful, the expression runs with the permissions of the callee. The other rules are essentially standard, we just note that (R-Struct) closes reduction under *heating*, an asymmetric variant of the standard structural congruence relation. The heating relation $E \Rrightarrow E'$ allows to syntactically rearrange E into E', for instance by exchanging the order of the function definitions and by extruding the scope of bound names (see the online technical report for a complete definition of the heating relation).

3.2 π-`Perms` vs Android

Intents. π-`Perms` can encode both implicit and explicit intents. Communication in π-`Perms` is non-deterministic, in that a function invocation $\overline{n}\langle m \triangleright \texttt{RECV}\rangle$ can trigger any function definition $n(x \triangleleft \texttt{CALL}).E$ in the same scope, provided that the permission checks are satisfied. Technically, this non-determinism is achieved

through the heating relation, which allows to liberally rearrange the pool of function definitions. Hence, communication in π-`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 component types are represented in π-`Perms` by means of function definitions. Activities in Android may be started by invoking the methods `startActivity` or `startActivityForResult`; in our calculus we treat the two cases uniformly, by having functions always return a result. 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 π-`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 our present security analysis.

Protection mechanisms. π-`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. The Android communication API only allows broadcast transmissions to be protected by permissions, namely requiring receivers to be granted specific permissions to get the intent. Function invocation in π-`Perms` accounts for the more general behaviour available to broadcast transmissions, since unprotected communication can be encoded simply by specifying $\bot$ as the permission required to the callee, as in $\overline{n}\langle m \triangleright \bot\rangle$.

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

$$D \triangleq s(x \triangleleft \mathtt{C}).[\mathtt{P}]\,(\nu b)\,(b(y \triangleleft \bot).[\mathtt{P}]\,\overline{a}\langle y \triangleright \bot\rangle \setminus b) \tag{1}$$

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

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

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; later, the client can perform an invocation to b (bound to z) to get access to the function a. The example unveils a 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`. We find this implementation potentially dangerous, since it is exposed to privilege escalation when binders are improperly disclosed.

Pending intents. π-`Perms` can naturally encode the simple form of permission delegation enabled by 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)" [15]. This informal description perfectly fits the previous encoding of binders in π-`Perms`, in that any component exposed to the binder b is allowed to invoke the corresponding function and let it run 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 inadvertently 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 [5]. To illustrate, consider three applications A, B and C. 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, if B can be freely accessed from A, then it may possibly act as a proxy between A and C.

We formalize a notion of safety against privilege escalation based on the IPC Inspection mechanism proposed by Felt *et al.* to dynamically prevent privilege escalation attacks on Android [11]. The idea behind IPC Inspection is simple: when an application receives a message from another application, a centralized reference monitor lowers the privileges of the recipient to the intersection of the privileges of the two interacting applications. A patched Android system implementing IPC Inspection is therefore protected against privilege escalation attacks "by design": we then take such a system as a reference specification and state a simulation-based notion of safety on top of it. As we discuss at the end of this section, the resulting definition provides an effective proof technique for the characterization of privilege escalation safety based on non-interference in [12].

To formalize the semantics of the IPC inspection mechanism, we first annotate each function definition of a given expression with a distinct label ℓ drawn from a denumerable set $\mathcal{L}$, disjoint from the set of values. The annotations make it possible to univocally identify the function triggered in response to each call, and hence trace the call chain. The IPC inspection semantics is then rendered formally by the labelled reduction relation $E \xrightarrow{\alpha}_i E'$ in Table 3, where α ranges uniformly over the set of annotation labels and the distinguished symbol $\cdot \notin \mathcal{L}$.

Note that, while the labelled transitions help tracking the dynamics of the call chains, the labels themselves do not have any import at runtime: in fact, function invocations do not mention labels at all and the semantics is still non-deterministic. We similarly label the original semantics in Table 2.

Let now $E_1 \asymp E_2$ denote two expressions that are syntactically equal but for their granted permissions (see the online technical report for a formal definition).

Definition 1 (IPC-Simulation). *A binary relation $\mathcal{R}$ contained in $\asymp$ is an* IPC-simulation *if and only if whenever $E_1 \mathcal{R} E_2$ and $E_1 \xrightarrow{\alpha} E_1'$ there exists E_2'*

(R-Call-IPC)

$$\frac{\texttt{RECV} \sqsubseteq \texttt{PERMS}' \qquad \texttt{CALL} \sqsubseteq \texttt{PERMS}}{n^{\ell}(x \triangleleft \texttt{CALL}).[\texttt{PERMS}']\, E \setminus [\texttt{PERMS}]\, \overline{n}\langle m \triangleright \texttt{RECV}\rangle \xrightarrow{\ell}_i [\texttt{PERMS} \sqcap \texttt{PERMS}']\, E\{m/x\}}$$

(R-Return-IPC)

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

(R-Context-IPC)

$$\frac{E \xrightarrow{\alpha}_i E'}{\mathcal{C}[E] \xrightarrow{\alpha}_i \mathcal{C}[E']}$$

(R-Struct-IPC)

$$\frac{E \Rrightarrow E_1 \xrightarrow{\alpha}_i E_2 \Rrightarrow E'}{E \xrightarrow{\alpha}_i E'}$$

Table 3. Reduction semantics for π-`Perms` under IPC Inspection

such that $E_2 \xrightarrow{\alpha}_i E_2'$ *with* $E_1' \mathcal{R} E_2'$. *We say that* E_1 *is* IPC-simulated *by* E_2 *(written* $E_1 \preccurlyeq_{IPC} E_2$*) iff there exists an IPC-simulation* $\mathcal{R}$ *such that* $E_1 \mathcal{R} E_2$.

The requirement $E_1 \asymp E_2$ guarantees that the labels that annotate the function definitions occurring in the two expressions are consistent (i.e., the same function bears the same label in E_1 and E_2) while disregarding any difference in the assigned permissions introduced upon reduction (cf. (R-Call) against (R-Call-IPC)). Given the previous definition, our notion of safety is immediate: an expression E is safe if and only if all its possible executions are oblivious to IPC Inspection being enabled or not.

Definition 2 (Safety). *An expression* E *is* safe *against privilege escalation if and only if* $E \preccurlyeq_{IPC} E$.

Though our definition is inspired by IPC Inspection, it reveals an important aspect which was never discussed before. Namely, we notice 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. This is precisely the case of example (1) where b exercises permissions `P`, but can be disclosed to any component which is granted permissions `C`. A sample Android application suffering of a similar flaw is given in the online technical report.

Our notion of safety is already a strong property, but we target a more ambitious goal: we desire protection despite the best efforts of an active opponent. In our model an opponent is a malicious, but unprivileged, Android application installed on the same device. Notice that the term "unprivileged" is loosely used here: we are not assuming that the opponent is granted no permission at all, but rather that it is not assigned any sensitive permission beforehand (in that case, it would have no reason in escalating privileges). In a typical security analysis, one can single out all the permissions under the control of the opponent (e.g., `INTERNET`) and identify the set of these permissions with $\bot$.

Definition 3 (Opponent). *A definition* O *is an* opponent *if and only if each permission assignment in* O *is* $\bot$.

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

Privilege escalation and non-interference. As we anticipated, a recent paper by Fragkaki *et al.* [12] proposes a definition of safety against privilege escalation inspired by the classic notion of non-interference for information flow control. 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 can rephrase their notion in our setting and prove that our definition implies, and hence may be employed as a proof technique for, theirs.

Let $|E|_\ell$ denote the expression obtained from E by erasing all the function definitions labelled with $\ell' \neq \ell$ and which are granted permissions $\mathtt{P} \sqsubset \mathtt{CALL}$, where $\mathtt{CALL}$ are the permissions required to invoke the function identified by ℓ.

Definition 5 (NI-Safety). *An expression E is* NI-safe *if and only if, for every ℓ occurring in E and for every reduction sequence $E \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$, there exist $E'_1, \ldots, E'_{n+1}$ such that $|E|_\ell \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} E'_n \xrightarrow{\ell} E'_{n+1}$.*

Proposition 1 (Safety vs NI-safety). *Safety implies NI-safety.*

Proof. Let $E \preccurlyeq_{IPC} E$ and assume $E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$. Since $E \preccurlyeq_{IPC} E$, we know that $E \xrightarrow{\alpha_1}_i E'_1 \xrightarrow{\alpha_2}_i \ldots \xrightarrow{\alpha_n}_i E'_n \xrightarrow{\ell}_i E'_{n+1}$ for some $E'_1, \ldots, E'_{n+1}$ such that $E_1 \asymp E'_1, \ldots, E_{n+1} \asymp E'_{n+1}$. By definition of the semantics $\xrightarrow{\alpha}_i$, we know that all the functions invoked in the call chain identified by $\alpha_1, \ldots, \alpha_n$ must be granted at least the permissions $\mathtt{CALL}$ needed to invoke ℓ. Hence, such function definitions are present also in $|E|_\ell$ and we can mimic the very same trace there.

We can thus confirm that the IPC Inspection mechanism enforces a reasonable semantic security property and justify further our choice of taking it as the building block for our safety notion. With respect to NI-safety, our notion has the important advantage of enabling a powerful form of coinductive reasoning, which is central to proving our main result (Theorem 2 below).

A still open question is if the two notions of safety are actually equivalent. We notice that for non-deterministic transition systems (bi)simulation-based equivalences are typically finer than trace equivalences, but at the time of writing we were not able to identify a counterexample in our setting.

5 Preventing privilege escalation by types and effects

Types and typing environments. A type τ may be either Un or a function type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$. 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 $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$ 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 SECR. We define the *secrecy level* of a type τ, written $\mathcal{S}(\tau)$, as expected, by having $\mathcal{S}(\mathsf{Un}) = \bot$ and $\mathcal{S}(\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}) = \mathtt{SECR}$. A typing *environment* Γ is a finite map from values to types. The *domain* of 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 in Table 4.

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ (\text{T-Proj}) \qquad \frac{\Gamma \vdash v : \tau \qquad \mathcal{S}(\tau) = \bot}{\Gamma \vdash v : \mathsf{Un}}\ (\text{T-Pub})$$

Table 4. Typing rules for values

Rule (T-Proj) is standard, while rule (T-Pub) makes it possible to treat all public data as "untyped", since they may possibly be disclosed to the opponent.

Typing expressions. The typing rules for expressions are in Table 5. The main judgement $\Gamma \vdash_{\mathtt{P}} E : \tau \blacktriangleright \mathtt{Q}$ is read as: expression E, running with permissions P, has type τ in Γ and exercises at most permissions Q throughout its execution. We also define an auxiliary judgement $\Gamma \vdash D$ to be read as: definition D is well-formed in Γ. The two judgement forms are mutually dependent.

We first notice that our effect system discriminates between *granted* and *exercised* permissions. For instance, the expression $a(x \triangleleft \bot).[\mathtt{P}]\,\overline{b}\langle n \triangleright \bot\rangle \setminus E$ could either be well-typed or not, even though the function a is publicly known, but is granted permissions $\mathtt{P} \sqsupset \bot$. The crux here is if the permissions P must be actually exercised or not to perform the invocation to b.

Apparently, we could enforce protection against privilege escalation by simply checking for each function definition that the privileges exercised by the function body are at most equal to the privileges required to invoke the function. However, since binders and pending intents allow indiscriminate access to potentially privileged components, our type system must also assign an appropriate secrecy level to these sensitive data and prevent their inadvertent disclosure. It turns out that in rule (T-Def) we must actually check that the permissions Q exercised by the function body must be at most equal to the join between the permissions CALL, needed to pass the security runtime checks upon invocation, and the permissions SECR, needed to learn the name of the function.

Interestingly, the opponent can play an active role in trying to get binders and pending intents under its control. In particular, by using rules (T-Def-Un) and (T-Call-Un), it can define arbitrary new functions and invoke existing ones, completely disregarding the restrictions enforced by typing. Protecting well-typed components requires then some care: for instance, in rule (T-Def) we must type-check public functions under the additional assumption that their input parameter is provided by the opponent with type Un; of course, in this case

(T-Def)
$$\frac{\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \quad \Gamma, x : \tau \vdash_\top E : \tau' \blacktriangleright \mathtt{Q} \qquad \mathtt{Q} \sqsubseteq \mathtt{CALL} \sqcup \mathtt{SECR} \quad \mathtt{CALL} \sqcup \mathtt{SECR} = \bot \Rightarrow \Gamma, x : \mathsf{Un} \vdash_\top E : \mathsf{Un} \blacktriangleright \bot \qquad x \notin dom(\Gamma)}{\Gamma \vdash u(x \triangleleft \mathtt{CALL}).E}$$

(T-Conj)
$$\frac{\Gamma \vdash D_1 \qquad \Gamma \vdash D_2}{\Gamma \vdash D_1 \wedge D_2}$$

(T-Eval)
$$\frac{\Gamma \vdash D \qquad \Gamma \vdash_{\mathtt{P}} E : \tau \blacktriangleright \mathtt{Q}}{\Gamma \vdash_{\mathtt{P}} D \setminus E : \tau \blacktriangleright \mathtt{Q}}$$

(T-Call)
$$\frac{\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \qquad \Gamma \vdash v : \tau \quad \bot \sqsubset \mathtt{RECV} \sqcup \mathtt{SECR} \qquad \mathtt{CALL} \sqcup \mathtt{SECR} \sqsubseteq \mathtt{P}}{\Gamma \vdash_{\mathtt{P}} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \tau' \blacktriangleright \mathtt{CALL} \sqcup \mathtt{SECR}}$$

(T-Val)
$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash_{\mathtt{P}} v : \tau \blacktriangleright \mathcal{S}(\tau)}$$

(T-Fail)
$$\frac{\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \quad \Gamma \vdash v : \tau'' \quad \mathtt{RECV} \sqcup \mathtt{SECR} = \bot \Rightarrow \mathcal{S}(\tau'') = \bot \quad \mathtt{CALL} \not\sqsubseteq \mathtt{P}}{\Gamma \vdash_{\mathtt{P}} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \mathsf{Un} \blacktriangleright \mathtt{P}}$$

(T-Perms)
$$\frac{\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{R} \qquad \mathtt{Q} \sqsubseteq \mathtt{P}}{\Gamma \vdash_{\mathtt{P}} [\mathtt{Q}]\, E : \tau \blacktriangleright \mathtt{R}}$$

(T-Let)
$$\frac{\Gamma \vdash_{\mathtt{P}} E : \tau \blacktriangleright \mathtt{Q} \qquad \Gamma, x : \tau \vdash_{\mathtt{P}} E' : \tau' \blacktriangleright \mathtt{R} \qquad x \notin dom(\Gamma)}{\Gamma \vdash_{\mathtt{P}} \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau' \blacktriangleright \mathtt{Q} \sqcup \mathtt{R}}$$

(T-Restr)
$$\frac{\Gamma, n : \tau \vdash_{\mathtt{P}} E : \tau' \blacktriangleright \mathtt{Q} \qquad n \notin dom(\Gamma)}{\Gamma \vdash_{\mathtt{P}} (\nu n)\, E : \tau' \blacktriangleright \mathtt{Q}}$$

(T-Def-Un)
$$\frac{\Gamma \vdash u : \mathsf{Un} \quad \Gamma, x : \mathsf{Un} \vdash_\bot E : \mathsf{Un} \blacktriangleright \bot \qquad x \notin dom(\Gamma)}{\Gamma \vdash u(x \triangleleft \mathtt{CALL}).E}$$

(T-Call-Un)
$$\frac{\Gamma \vdash u : \mathsf{Un} \qquad \Gamma \vdash v : \mathsf{Un}}{\Gamma \vdash_\bot \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \mathsf{Un} \blacktriangleright \bot}$$

Table 5. Typing rules for definitions and expressions

no privilege must be exercised. Similarly, in rule (T-Call) we cannot trust the return type of a function when the invocation can be dispatched to the opponent: this justifies the third premise of the rule.

Rule (T-Fail) allows to provide an argument of arbitrary type to any function which will never be invoked at runtime, since the caller is granted permissions P, but the function requires permissions $\mathtt{CALL} \not\sqsubseteq \mathtt{P}$ to be invoked. Again, the information CALL in the function type can be trusted only when the function is not defined by the opponent, hence some additional care is needed to prevent secrecy violations in that case (see the third premise of the rule). Note that, due to such a possible interaction with the opponent, the exercised permissions are conservatively assumed to be P, i.e., all the permissions granted 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 Defini-

tion 2 and its robust variant. Our implementation, however, features a number of escape hatches based on Java annotations to keep programming practical, much in the spirit of the declassification/endorsement constructs customary to the literature on information-flow control. We discuss this point further in Section 6.

Example type-checking. We briefly discuss how example (1) is deemed as ill-typed according to our type discipline. We first note that, since function a requires permissions $\mathtt{P}$ to be called, the invocation $\overline{a}\langle y \triangleright \bot\rangle$ is assigned at least the effect $\mathtt{P}$ by (T-Call). Hence, the only possible way to type-check the function definition $b(y \triangleleft \bot).[\mathtt{P}]\,\overline{a}\langle y \triangleright \bot\rangle$ through (T-Def) is by assigning b a function type τ such that $\mathcal{S}(\tau) = \mathtt{P}$. Assuming that the service s is a public component, this implies that the function definition $s(x \triangleleft \mathtt{C}).[\mathtt{P}]\,(\nu b)\ \ldots \setminus b$ is ill-typed by (T-Def), since the effect $\mathtt{P}$ assigned to the service body b by (T-Val) is not lesser or equal to the permissions $\mathtt{C}$ required to invoke the service s.

Formal results. 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 the Subject Reduction theorem: full details can be found in the online technical report.

Theorem 1 (Type Safety). *If $\Gamma \vdash_\top E : \tau \blacktriangleright \mathtt{P}$ for any $\mathtt{P}$, then $E \preccurlyeq_{IPC} E$.*

The next result states that our type system does not constrain the opponent. Its proof follows by a simple structural induction.

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

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

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

6 Implementation

We have implemented the type system as a tool (`Lintent`) designed as a plug-in for Android `Lint`, the widely popular utility distributed with Android's ADT.

`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. Additionally, `Lintent` infers and records the types of data injected into and extracted from intents, while tracking the flow of inter-component message passing. This is needed to prevent privilege escalation attacks exploiting improper disclosure of binders

or pending intents, and at the same time proves very effective in detecting common programming errors related to misuse of intents [16].

`Lintent` analyzes Java source code: in principle, the same analysis could be performed on the Java bytecode, though reasoning about types at the bytecode level is arguably more demanding than at source level [14]. Below, we give a brief overview of the main features of the tool and of the the main challenges we had to face during its development.

Type reconstruction. The hardest challenge for the implementation is related to the widespread use of "untyped" coding patterns supported by the current Android API. Consider, for instance, a simple scenario of intent usage with multiple data types:

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

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

On the recipient side, intent "extras" are retrieved by freely accessing the intent as if it was a dictionary, so the receiver may actually retrieve data of unexpected type and fail at runtime, or disregard altogether some keys provided by the sender [16].

```
class ReceiverActivity extends Activity {
    static class WS implements Serializable { ... }

    void onCreate(Bundle savedInstanceState) {
        Intent i = getIntent();
        String k1 = i.getStringExtra("k1"); // run-time type error!
        WS o = (WS)i.getSerializableExtra("k3"); // dynamic cast fails!
        // data associated to k2 is never extracted!
    }
}
```

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 each k_i is a string constant and each T_i is a Java type. This enforces a much stronger discipline on data passing between components, which is consistent with our type system, where a function type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$ constrains the caller in providing an argument of type

τ and the callee in returning a result of type τ'. A similar discipline is crucial in Android applications to protect the secrecy of binders and pending intents.

Notice that, since the `putExtra` method is overloaded to 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. As a valuable byproduct of this analysis, `Lintent` is able to warn the user in case of intents misuse.

Partial evaluation. As noted above, each piece of data put into an intent must be bound to a key, hence an intent object can be seen 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. Whether they happen to be string literals or the result of 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 invocations: 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. Partial evaluation is required for reconstructing the intent record type labels described above.

API signatures and permissions. Implementing the rules of the type system for π-`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 several different patterns to implement inter-component communication. Moreover, many 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[2], which are thus updatable with no need to rebuild the tool. Finally, `Lintent` must perform type resolution for third-party libraries: access to `jar` files must be granted to the tool to let it inspect the contents of imported packages and classes through the `javap` disassembler.

Java annotations support. We rely on Java annotations to provide some escape hatches from the tight discipline imposed by `Lintent`. Several privileged components intentionally expose functionalities, thus we define annotations of the form `@priv{endorse="P"}` to mark methods such as `onCreate()` with a set of permissions `P` that the type-checker will disregard. More precisely, if the method exercises the permissions set `Q`, the associated component is deemed well-typed as long as it is protected with at least permissions `Q`$\setminus$`P`. A similar treatment is implemented for pending intents based on the annotation `@priv{declassify="P"}`, to lower the secrecy level of such objects computed by `Lintent`.

7 `Lintent`: typing experiments and findings

At the time of writing `Lintent` is able to type-check activities, started services and broadcast receivers. The current prototype should be considered in alpha

[2] Currently such permission map files are those distributed with Stowaway [10].

stage, as we are currently performing tests, fixing bugs and adding support for some missing Java language features. Still, we were able to analyze some existing open-source applications from the Google Play store and identify previously unknown privilege escalation attacks on them. In our case studies we performed a code refactoring to avoid the usage of some Java features which are still unsupported by `Lintent`, like reflective calls and nested classes. However, our findings are confirmed by running the original applications on a Nexus device.

The first case study we consider is `APN-Switch`, a widget that allows to enable and disable the device data connection with a click. Of course, these network operations are sensitive, hence the application requires the permission `CHANGE_NETWORK_STATE` to be installed. Unfortunately, `APN-Switch` is exposed to privilege escalation attacks: an unprivileged malicious application can forge an intent to the action string `ch.blinkenlights.android.apnswitch.CLICK` and simulate a click of the user on the widget, thus enabling (or disabling) the device data connection as if it were granted the `CHANGE_NETWORK_STATE` permission.

Our second case study is `Wifi Fixer`, a small application aimed at fixing several problems with the Android wifi. Also `Wifi Fixer` suffers of privilege escalation attacks, since it requires the permission `CHANGE_WIFI_STATE` to toggle on and off the wifi connection, but any unprivileged application can send an intent to the action string `org.wahtod.wififixer.ACTION_WIFI_OFF` to disconnect the wifi. Interestingly, the widget handling the wifi connection is declared as an internal component, hence it cannot receive intents from third-party applications; however, a public broadcast receiver in the application can act as a proxy to the widget, thus allowing to escalate privileges.

Both `APN-Switch` and `Wifi Fixer` are released on the official Google Play store, hence available to a wide audience. We argue that `Lintent` can prove helpful not only in detecting malicious code lying within existing source programs, but also in assisting well-meaning developers in identifying potential attack surfaces for privilege escalation and many other common programming mistakes, way before their applications reach the Google Play store.

8 Related work

The literature on Android application security is substantial, as reported in a recent survey by Enck [6].

Android permissions. Davi *et al.* [5] were the first to point out the weaknesses of the Android permission system with respect to privilege escalation attacks. Later, Felt *et al.* proposed IPC Inspection as a possible runtime protection mechanism [11]. Though effective, IPC Inspection may induce substantial performance overhead, as it requires to keep track of different application instances to make the protection mechanism precise. In a recent paper, Bugiel *et al.* describe a sophisticated runtime framework for enforcing protection against privilege escalation attacks [2]. Notably, their solution comprises countermeasures against colluding applications, an aspect which is neglected by both IPC Inspection and

`Lintent`. Providing such guarantees, however, requires a centralized solution built over the operating system. Our approach is complementary: runtime protection is useful against malicious applications which reach the Android market, while static analysis techniques can prove helpful for well-meaning developers who wish to assess the robustness of their applications. Finally, Felt *et al.* proposed Stowaway, a tool for detecting overprivilege in Android applications [10]. In our implementation we take advantage of their permission map, which relates API method calls to their required permissions.

Android communication. Chin *et al.* [4] were the first to study the threats related to the Android message-passing system. They provide also a tool, ComDroid, which is able to detect potential vulnerabilities in the usage of intents. ComDroid 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 and we are able to confirm many previous observations 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 fuzzy testing techniques, exposing some interesting findings [16]. Their empirical methodology, however, does not provide any clear understanding of the correct programming patterns for communication.

Formal models. π-`Perms` is partly inspired by a core formal language proposed by Chaudhuri [3]. With respect to Chauduri's model, π-`Perms` provides a more thorough treatment of the Android system, including implicit communication, runtime registration of new components, service binding and pending intents. 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 [13]. Another work by Fragkaki *et al.* discusses a number of enhancements over the Android permission system and validates their effectiveness in an abstract model [12] (cf. Section 4). The focus of the work remains on runtime protection mechanisms, however, as opposed to static analysis. The paper also discusses some issues related to controlled delegation, but it does it independently from privilege escalation. Finally, Armando *et al.* proposed a formal model of the Android operating system and a verification technique based on history expressions [1]. However, any specific security analysis is left for future work and no implementation is provided.

9 Conclusions

We have proposed a sound type-based analysis technique targeted at the static detection of privilege escalation attacks on Android, and developed `Lintent`, a prototype security type-checker which implements our analysis. Our tool addresses a number of engineering challenges which are central to the practical development of any sound type-checker for Android applications. We showed the effectiveness of our tool by unveiling real attacks on existing applications.

As part of our future work, we want to focus on the study of robust declassification and endorsement programming patterns in our formal framework, to assess the impact on security of the Java annotations discussed in Section 6. On the practical side, we want to further develop `Lintent` and add support for many features of the Android platform which are still missing. We also plan to integrate `Lintent` with a frontend to a decompiler as `ded` [8] to support the analysis of third-party applications.

Acknowledgements Work partially supported by MIUR PRIN Project "CINA: Compositionality, Interaction, Negotiation and Autonomicity", and conducted in cooperation with SMC Treviso s.r.l. The third author was supported by a EU-Regione Veneto funded fellowship within the POR FESR 2007 – 2013 Program, Action 1.1.3.

References

1. Armando, A., Costa, G., Merlo, A.: Formal modeling and verification of the Android security framework. In: TGC (2012)
2. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: NDSS (2012)
3. Chaudhuri, A.: Language-based security on Android. In: PLAS. pp. 1–7 (2009)
4. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys. pp. 239–252 (2011)
5. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: ISC. pp. 346–360 (2010)
6. Enck, W.: Defending users against smartphone apps: Techniques and future directions. In: ICISS. pp. 49–70 (2011)
7. 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)
8. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: USENIX Security Symposium (2011)
9. Enck, W., Ongtang, M., McDaniel, P.D.: Understanding Android security. IEEE Security & Privacy 7(1), 50–57 (2009)
10. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: CCS. pp. 627–638 (2011), `http://www.android-permissions.org/`
11. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: Attacks and defenses. In: USENIX Security Symposium (2011)
12. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: ESORICS. pp. 1–18 (2012)
13. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of Android applications (2009), Technical report, University of Maryland.
14. Gagnon, E., Hendren, L.J., Marceau, G.: Efficient inference of static types for Java bytecode. In: SAS. pp. 199–219 (2000)
15. Google Inc: Reference documentation for `android.app.PendingIntent`. `http://developer.android.com/reference/android/app/PendingIntent.html`
16. 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 (2012)