Language-Based Security on Android

Avik Chaudhuri
University of Maryland at College Park
avik@cs.umd.edu

Abstract

In this paper, we initiate a formal study of security on Android: Google's new open-source platform for mobile devices. Specifically, we present a core typed language to describe Android applications, and to reason about their data-flow security properties. Our operational semantics and type system provide some necessary foundations to help both users and developers of Android applications deal with their security concerns.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Access controls, Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control constructs

General Terms Security, Languages, Verification

Keywords data-flow security, hybrid type system, mobile code, certified compilation

1. Introduction

Android [3] is Google's new open-source platform for mobile devices. Designed to be a complete software stack, it includes an operating system, middleware, and core applications. Furthermore, it comes with an SDK [1] that provides the tools and APIs necessary to develop new applications for the platform in Java. Interestingly, Android does not distinguish between its core applications and new applications developed with the SDK; in particular, all applications can potentially interact with the underlying mobile device and share their functionality with other applications. This design is very encouraging for developers and users of new applications, as witnessed by the growing Android "market" [2]. At the same time, it can be a source of concern—*what do we understand about security on Android?*

Indeed, suppose that Alice downloads and installs a new application, developed by Bob, on her Android-based phone. Say this application, *wikinotes*, interacts with a core application, *notes*, to publish some notes from the phone to a wiki, and to sync edits back from the wiki to the phone. Of course, Alice would not like all her notes to be published, and would not like all her published notes to be edited; for instance, her notes may include intermediate results of her ongoing lab experiments. How does she know whether it is safe to run the application? Can she trust the application to safely access her data? If she cannot, is there still a way to safely run the application? These concerns are important for Alice, because she realizes that running a malicious application on her phone can be disastrous; for instance, it may compromise the records of her experiments. Conversely, it is Bob's concern to be able to convince Alice that his application can be run safely on her phone.

This paper initiates an effort to help Alice and Bob deal with these related concerns, through a unified formal understanding of security on Android. To this end, we envision a recipe inspired by PCC [11]: Bob constructs a safety proof for his application by some conservative analysis of the associated Java code, and Alice verifies the proof before installing the application. Such a recipe requires (at least) two ingredients: (1) a formal operational semantics for application code in an Android environment—this includes, in particular, formal specifications of the APIs provided by the SDK; (2) a static safety analysis for application code based on this semantics—in particular, this analysis may be formalized as a security type system for applications, and a soundness proof for such a system should provide the necessary safety proofs for well-typed applications. We take some initial steps towards composing such a recipe in this paper.

- We design a core formal language to describe and reason about Android applications abstractly. For now, we focus only on constructs that are unique to Android, while ignoring the other usual Java constructs that may appear in Android applications. This simplification allows us to study Android-specific features in isolation. Still, to reason about actual Android applications we must consider the usual Java features in combination with these features, and we plan to extend our language to include those features in the future.
- We present an operational semantics for our language. Our semantics exposes the sophisticated control flows

PLAS '09 June 15, Dublin, Ireland.

that underlie Android's constructs. Indeed, while the official documentation provides only a vague idea of what these constructs mean, a formal understanding of their semantics is crucial for reasoning correctly about the behavior of Android applications. We test our semantics by running our own applications on an Android emulator (included in the SDK).

- We present a type system for security in this language. Our system exploits the access control mechanisms already provided by Android, and enforces "best practices" for developing secure applications with these mechanisms. We develop some new technical concepts, including a special notion of *stack types*, for this purpose. The resulting guarantees include standard data-flow security properties for well-typed applications described in our language. We expect that these guarantees can be preserved without significant difficulty by extending our type system to handle other usual Java constructs.

As future work, we plan to extend our analysis so that it can handle application code in Java, and implement it using existing static analysis tools [10, 12, 7]. As envisioned above, certified installation of Android applications based on such an implementation should help both users and developers deal with their security concerns. More ambitiously, we believe that this setting provides an ideal opportunity to bring language-based security to the mainstream. Indeed, Android applications are usually small and structured, so we expect type inference to scale well on such applications; furthermore, the idea of certified installation should certainly be attractive to a growing and diverse Android community.

The rest of the paper is organized as follows. In Section 2 we present an overview of Android, focusing on the application model and security mechanisms. In Section 3 we present our formal language for Android applications. In Section 4, we present our security type system for this language, and outline key properties of typing. Finally, in Section 5, we discuss some related work and conclude.

2. Overview of Android

2.1 The application model

In Android's application model [1], an application is a package of *components*, each of which can be instantiated and run as necessary (possibly even by other applications). Components are of the following types:

Activity components form the basis of the user interface; usually, each window of the application is controlled by some activity.

Service components run in the background, and remain active even if windows are switched. Services can expose interfaces for communication with other applications.

Receiver components react asynchronously to messages from other applications.

Provider components store data relevant to the application, usually in a database. Such data can be shared across applications.

Consider, *e.g.*, a music-player application for an Android-based phone. This application may include several components. There may be activities for viewing the songs on the phone, and for editing the details of a particular song. There may be a service for playing a song in the background. There may be receivers for pausing a song when a call comes in, and for restarting the song when the call ends. Finally, there may be a provider for sharing the songs on the phone.

Component classes and methods The Android SDK provides a base class for each type of component (`Activity`, `Service`, `Receiver`, and `Provider`), with methods (callbacks) that are run at various points in the life cycle of the associated component. Each component of an application is defined by extending one of the base classes, and overriding the methods in that class. In particular:

- The `Activity` class has methods that are run when some activity calls `this` activity, or returns to `this` activity.
- The `Service` class has a method that is run when some component binds to `this` service.
- The `Receiver` class has a method that is run when a message is sent to `this` receiver.
- The `Provider` class has methods to query and update the data stored by `this` provider.

2.2 Security mechanisms

As mentioned above, it is possible for an application to share its data and functionality across other applications, by letting such applications access its components. Clearly, these accesses must be carefully controlled for security. We now describe the key access control mechanisms provided by Android [1].

Isolation The Android operating system builds on a Linux kernel, and as such, derives several protection mechanisms from Linux. Every application runs in its own Linux process. Android starts the process when any of the application's code needs to be run, and stops the process when another application's code needs to be run. Next, each process runs on its own Java VM, so the application's code runs in isolation from the code of all other applications. Finally, each application is assigned a unique Linux UID, so the application's files are not visible to other applications.

That said, it is possible for several applications to arrange to share the same UID (see below), in which case their files become visible to each other. Such applications can also arrange to run in the same process, sharing the same VM.

Permissions Any application needs explicit permissions to access the components of other applications. Crucially, *such permissions are set at install time, not at run time*.

The permissions required by an application are declared statically in a manifest. These permissions are set by the package installer, usually via dialogue with the user. No further decisions are made at run time; if the application's code needs a permission at run time that is not set at install time, it blocks, and its resources are reclaimed by Android.

Enforcing permissions can prevent an application from calling certain activities, binding to certain services, sending messages to certain receivers, and receiving messages from other applications or the system, and querying and updating data stored by certain providers.

Signatures Finally, any Android application must be signed with a certificate whose private key is held by the developer. The certificate does not need to be signed by a certificate authority; it is used only to establish trust between applications by the same developer. For example, such applications may share the same UID, or the same permissions.

3. Language

We now proceed to design a core formal language to describe Android applications. To narrow our focus, we do not model general classes and methods; instead, we treat component classes and methods as primitive constructs. Furthermore, we ignore isolation and signatures, since permissions suffice to model the effects of those mechanisms in Android.

3.1 Syntax

We assume a lattice $\sqsupseteq$ of permissions (*e.g.*, PERMS). In Android, this lattice is implemented over sets. Components are identified by names. In Android, components are accessed through *intents*; an intent simply pairs the name of the component to access (*action*) and a value (*parameter*) to be passed to the component. Values include names n, variables x and a constant void. Our syntax of programs is as follows.

Program syntax

$v ::= n \mid x \mid \mathsf{void}$	value
$i ::= (n, v)$	intent
$t ::=$	code
$\mathsf{call}(i)$	call activity
$\mathsf{return}(v)$	return from activity
$\mathsf{bind}(i, \lambda x.t)$	bind to service
$\mathsf{register}(\mathtt{SEND}, \lambda x.t)$	register new receiver
$\mathsf{send}(\mathtt{RECEIVE}, i)$	send to receiver
$!n$	read from provider
$n := v$	write to provider
$\mathsf{let}\ x = t\ \mathsf{in}\ t'$	evaluate
$\lnot t$	fork
$t + t'$	choice
v	result

We describe the meanings of programs informally below; a formal operational semantics appears in Section 3.2.

A program runs in an *environment* that maps names to component definitions. In Android, such an environment is derived from the set of applications installed on the system.

Application syntax

$d ::=$	definition
$\mathsf{activity}(\mathtt{CALL}, \mathtt{PERMS}, \lambda x.t, \lambda x.t')$	activity
$\mathsf{service}(\mathtt{BIND}, \mathtt{PERMS}, \lambda x.t)$	service
$\mathsf{receiver}(\mathtt{SEND}, \mathtt{PERMS}, \lambda x.t)$	receiver
$\mathsf{provider}(\mathtt{READ}, \mathtt{WRITE}, v)$	provider
$D ::= \varnothing \mid D, n \mapsto d$	hash of definitions

Furthermore, a program runs with a permission (context). In general, the program may be run on a stack of windows (produced by calls to activities), or in a pool of threads (produced by forks). Exceptions are call or return programs, which can only be run on the stack.

- The program $\mathsf{call}((n, v))$ checks that n is mapped to an activity of the form $\mathsf{activity}(\mathtt{CALL}, \mathtt{PERMS}, \lambda x.t, \lambda x.t')$, and that the current context has permission CALL. A new window is pushed on the stack, and the program t is run with permission PERMS and with x bound to v.
- Dually, $\mathsf{return}(v)$ pops the current window off the stack and returns control to the previous activity; if that activity is of the form shown above, the program t' is run with permission PERMS and with x bound to v.
- The program $\mathsf{bind}((n, v), \lambda x.t')$ checks that n is mapped to a service of the form $\mathsf{service}(\mathtt{BIND}, \mathtt{PERMS}, \lambda x.t)$, and that the current context has permission BIND. The program t is run with permission PERMS and with x bound to v, and the result v' is passed back to the current context; then, t' is run with x bound to v' in the current context. In Android, v' is typically an interface to some functionality exposed by the service. Below, we encode away such services with receivers.
- The program $\mathsf{register}(\mathtt{SEND}, \lambda x.t)$ creates a fresh name n, maps it to the receiver $\mathsf{receiver}(\mathtt{SEND}, \mathtt{PERMS}, \lambda x.t)$ in the environment, and returns n; here, PERMS is the permission of the current context.
- Dually, $\mathsf{send}(\mathtt{RECEIVE}, (n, v))$ checks that n is mapped to a receiver of the form $\mathsf{receiver}(\mathtt{SEND}, \mathtt{PERMS}, \lambda x.t)$, that the current context has permission SEND, and that PERMS includes RECEIVE. The program t is run with permission PERMS, and with x bound to v.
- The program $!n$ checks that n is mapped to a provider of the form $\mathsf{provider}(\mathtt{READ}, \mathtt{WRITE}, v)$, and that the current context has permission READ; the value v is returned.
- Dually, $n := v$ checks that n is mapped to a provider of the form $\mathsf{provider}(\mathtt{READ}, \mathtt{WRITE}, v')$, and that the current context has permission WRITE; n is then mapped to $\mathsf{provider}(\mathtt{READ}, \mathtt{WRITE}, v)$ in the environment.
- The program $\mathsf{let}\ x = t\ \mathsf{in}\ t'$ evaluates t, and then evaluates t' with x bound to the result.
- The program $\lnot t$ forks a thread that runs t.
- The program $t + t'$ evaluates either t or t'. In Android, such choices arise in the user interface.

Local reduction $D; e; E \to D'; e'; E'$

$$\textbf{(Red let-distr)}\quad D; [\texttt{PERMS}]\ \mathsf{let}\ x = t\ \mathsf{in}\ t'; E \to D; \mathsf{let}\ x = [\texttt{PERMS}]\ t\ \mathsf{in}\ [\texttt{PERMS}]\ t'; E$$

$$\textbf{(Red let-eval)}\quad \frac{D; e; E \to D'; e'; E'}{D; \mathsf{let}\ x = e\ \mathsf{in}\ [\texttt{PERMS}]\ t; E \to D'; \mathsf{let}\ x = e'\ \mathsf{in}\ [\texttt{PERMS}]\ t; E'}$$

$$\textbf{(Red let-return)}\quad D; \mathsf{let}\ x = [\texttt{PERMS}']\ v\ \mathsf{in}\ [\texttt{PERMS}]\ t; E \to D; [\texttt{PERMS}]\ t\{v/x\}; E$$

$$\textbf{(Red fork)}\quad \frac{E' = E, [\texttt{PERMS}]\ t}{D; [\texttt{PERMS}]\ \curvearrowleft\ t; E \to D; [\texttt{PERMS}]\ \mathsf{void}; E'}$$

$$\textbf{(Red read)}\quad \frac{D(n) = \mathsf{provider}(\texttt{READ}, \_, v) \qquad \texttt{PERMS} \sqsupseteq \texttt{READ}}{D; [\texttt{PERMS}]\ !n; E \to D; [\texttt{PERMS}]\ v; E}$$

$$\textbf{(Red write)}\quad \frac{D(n) = \mathsf{provider}(\texttt{READ}, \texttt{WRITE}, \_) \qquad \texttt{PERMS} \sqsupseteq \texttt{WRITE} \qquad D' = D[n \mapsto \mathsf{provider}(\texttt{READ}, \texttt{WRITE}, v)]}{D; [\texttt{PERMS}]\ n := v; E \to D'; [\texttt{PERMS}]\ \mathsf{void}; E}$$

$$\textbf{(Red register)}\quad \frac{n\ \text{fresh} \qquad D' = D, n \mapsto \mathsf{receiver}(\texttt{SEND}, \texttt{PERMS}, \lambda x.t)}{D; [\texttt{PERMS}]\ \mathsf{register}(\texttt{SEND}, \lambda x.t); E \to D'; [\texttt{PERMS}]\ n; E}$$

$$\textbf{(Red send)}\quad \frac{D(n) = \mathsf{receiver}(\texttt{SEND}, \texttt{PERMS}', \lambda x.t) \qquad \texttt{PERMS} \sqsupseteq \texttt{SEND} \qquad \texttt{PERMS}' \sqsupseteq \texttt{RECEIVE}}{D; [\texttt{PERMS}]\ \mathsf{send}(\texttt{RECEIVE}, (n, v)); E \to D; [\texttt{PERMS}']\ t\{v/x\}; E}$$

$$\textbf{(Red choice-l)}\quad D; [\texttt{PERMS}]\ t + t'; E \to D; [\texttt{PERMS}]\ t; E$$

$$\textbf{(Red choice-r)}\quad D; [\texttt{PERMS}]\ t + t'; E \to D; [\texttt{PERMS}]\ t'; E$$

Global reduction $D; S; E \longrightarrow D'; S; E'$

$$\textbf{(Red local)}\quad \frac{D; e; E \to D'; e'; E'}{D; \langle e, \lambda x.e'' \rangle :: S; E \longrightarrow D'; \langle e', \lambda x.e'' \rangle :: S; E'}$$

$$\textbf{(Red call)}\quad \frac{D(n) = \mathsf{activity}(\texttt{CALL}, \texttt{PERMS}', \lambda x.t, \lambda x.t') \qquad \texttt{PERMS} \sqsupseteq \texttt{CALL} \qquad S' = \langle [\texttt{PERMS}]\ \mathsf{void}, \lambda x.e \rangle :: S}{D; \langle [\texttt{PERMS}]\ \mathsf{call}((n, v)), \lambda x.e \rangle :: S; E \longrightarrow D; \langle [\texttt{PERMS}']\ t\{v/x\}, \lambda x.[\texttt{PERMS}']\ t' \rangle :: S'; E}$$

$$\textbf{(Red return)}\quad \frac{S = \langle [\texttt{PERMS}']\ \mathsf{void}, \lambda x.e' \rangle :: S'}{D; \langle [\texttt{PERMS}]\ \mathsf{return}(v), \lambda x.e \rangle :: S; E \longrightarrow D; \langle e'\{v/x\}, \lambda x.e' \rangle :: S'; E}$$

$$\textbf{(Red thread)}\quad \frac{D; e; E \to D'; e'; E'}{D; S; E, e \longrightarrow D'; S; E', e'}$$

Figure 1. Small-step operational semantics

3.2 Semantics

We now present a formal small-step operational semantics.

Since services can be encoded with receivers (as follows), we do not consider services any further in our development.

Encodings

$$\begin{aligned} \mathsf{service}(\texttt{BIND}, \texttt{PERMS}, \lambda x.t) &\triangleq \mathsf{receiver}(\texttt{BIND}, \texttt{PERMS}, \lambda x.t) \\ \mathsf{bind}((n, v), \lambda x.t) &\triangleq \mathsf{let}\ x = \mathsf{send}(\bot, (n, v))\ \mathsf{in}\ t \end{aligned}$$

Next, we introduce some internal syntactic categories to describe intermediate states of an Android system. Recall that a program runs with a permission, and may be run on a stack of windows or in a pool of threads.

An *expression* denotes code running with a particular permission (context). A thread is simply an expression. A window is of the form $\langle e, \lambda x.e' \rangle$, where e is the expression currently running in the window, and $\lambda x.e'$ is the callback invoked when a window returns control to this window.

Internal syntax

$e ::=$	expression
$\quad [\texttt{PERMS}]\ t$	encapsulate
$\quad \mathsf{let}\ x = e\ \mathsf{in}\ e'$	evaluate
$E ::= \varnothing \mid E, e$	pool of threads
$S ::= \epsilon \mid \langle e, \lambda x.e' \rangle :: S$	stack of windows

Now, a *state* is a tuple $D; S; E$, where D is an environment, S is a stack of windows, and E is a pool of threads.

Figure 1 shows reduction rules that formalize the semantics explained in Section 3.1. A global reduction relation, $\longrightarrow$, describes the reduction of states. This relation depends on a local reduction relation, $\to$, that describes the reduction of expressions under an environment and a pool of threads. In particular, call and return programs reduce under $\longrightarrow$, and all other code reduces under $\to$. Of specific interest are **(Red let-return)**, **(Red send)**, **(Red call)**, and **(Red return)**, that can cause data flows across contexts.

A typical initial configuration of the system is of the form $D; \langle e, \lambda x.e \rangle; \varnothing$, where D is the environment defined by the set of installed applications, x is fresh, and e is of the form $[\top]\ t$; for example, t may be a choice of calls to the main activities of the installed applications, modeling code running in the "home" window of an Android-based phone.

4. Type system

Next, we present a system of security types for our language.

Types

$\tau ::=$	data type
$\mathsf{Any}(\mathtt{READ}, \mathtt{WRITE})$	any
$\mathsf{Activity}(\mathtt{CALL}, \tau \to \tau' \Rightarrow \tau'')$	activity
$\mathsf{Receiver}(\mathtt{SEND}, \tau \to T)$	receiver
$\mathsf{Provider}(\mathtt{READ}, \mathtt{WRITE})$	provider
Stuck	stuck
$T ::=$	type
τ	data type
$\tau \Rightarrow \tau'$	stack type

Roughly, the type $\mathsf{Any}(\mathtt{READ}, \mathtt{WRITE})$ is given to data that may flow from contexts with at most permission WRITE to contexts with at least permission READ. This type is the basis for security specifications in the language (see Theorem 4.2). For example, $\mathsf{Any}(\top, \top)$ types secret trusted data, $\mathsf{Any}(\bot, \top)$ types public trusted data, and $\mathsf{Any}(\bot, \bot)$ types public tainted data.

Next, we have types for each component class, that are given to names bound in the environment. The meanings of these types are given in the rules for well-formed environments below. Besides typing information, these types record the permissions required to access the associated components, so that programs that block due to access control can be identified. Such programs are vacuously safe, and are given the type Stuck. The treatment of Stuck closely follows previous work [8], and we omit the details in the sequel.

Finally, we introduce *stack types*. A stack type $\tau \Rightarrow \tau'$ is given to an expression running at the top of a stack; values returned by any window to this window have type τ, and values returned by this window have type τ'. Note that the code run by an activity must have a stack type, since it is always run on a stack; in contrast, the code run by a receiver may or may not have a stack type.

4.1 Typing rules

Let Γ be a list that associates names and variables to unique typing hypotheses, such as $n : \tau$ or $x : \tau$. We have the following subtyping rule, that captures safe data flows. (Other subtyping rules are not shown here.)

Subtyping $\Gamma \vdash T <: T'$

$$\textbf{(Sub any)}\quad \frac{\mathtt{READ}' \sqsupseteq \mathtt{READ} \qquad \mathtt{WRITE} \sqsupseteq \mathtt{WRITE}'}{\Gamma \vdash \mathsf{Any}(\mathtt{READ}, \mathtt{WRITE}) <: \mathsf{Any}(\mathtt{READ}', \mathtt{WRITE}')}$$

Well-typed code $\Gamma \vdash_{\mathtt{PERMS}} t : T$

$$\textbf{(Typ read)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} n : \mathsf{Provider}(\mathtt{READ}, \mathtt{WRITE})}{\Gamma \vdash_{\mathtt{PERMS}} !n : \mathsf{Any}(\mathtt{READ}, \mathtt{WRITE})}$$

$$\textbf{(Typ write)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} n : \mathsf{Provider}(\mathtt{READ}, \mathtt{WRITE}) \quad \Gamma \vdash_{\mathtt{PERMS}} v : \mathsf{Any}(\mathtt{READ}, \mathtt{WRITE})}{\Gamma \vdash_{\mathtt{PERMS}} n := v : \mathsf{Any}(\bot, \top)}$$

$$\textbf{(Typ register)}\quad \frac{\Gamma, x : \tau \vdash_{\mathtt{PERMS}} t : T \quad T' = \mathsf{Receiver}(\mathtt{SEND}, \tau \to T)}{\Gamma \vdash_{\mathtt{PERMS}} \mathsf{register}(\mathtt{SEND}, \lambda x.t) : T'}$$

$$\textbf{(Typ send)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} n : \mathsf{Receiver}(\_, \tau \to T) \quad \Gamma \vdash_{\mathtt{PERMS}} v : \tau}{\Gamma \vdash_{\mathtt{PERMS}} \mathsf{send}(\mathtt{RECEIVE}, (n, v)) : T}$$

$$\textbf{(Typ let)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} t : \tau \qquad \Gamma, x : \tau \vdash_{\mathtt{PERMS}} t' : T}{\Gamma \vdash_{\mathtt{PERMS}} \mathsf{let}\ x = t\ \mathsf{in}\ t' : T}$$

$$\textbf{(Typ fork)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} t : \tau}{\Gamma \vdash_{\mathtt{PERMS}} \curlywedge t : \mathsf{Any}(\bot, \top)}$$

$$\textbf{(Typ choice)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} t : T \qquad \Gamma \vdash_{\mathtt{PERMS}} t' : T}{\Gamma \vdash_{\mathtt{PERMS}} t + t' : T}$$

$$\textbf{(Typ val-hyp)}\quad \frac{v : \tau \in \Gamma}{\Gamma \vdash_{\mathtt{PERMS}} v : \tau}$$

$$\textbf{(Typ val-void)}\quad \Gamma \vdash_{\mathtt{PERMS}} \mathsf{void} : \mathsf{Any}(\bot, \top)$$

$$\textbf{(Typ call)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} n : \mathsf{Activity}(\mathtt{CALL}, \tau \to \_ \Rightarrow \tau') \quad \Gamma \vdash_{\mathtt{PERMS}} v : \tau}{\Gamma \vdash [\mathtt{PERMS}]\ \mathsf{call}((n, v)) : \tau' \Rightarrow \_}$$

$$\textbf{(Typ return)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} v : \tau}{\Gamma \vdash [\mathtt{PERMS}]\ \mathsf{return}(v) : \_ \Rightarrow \tau}$$

Well-formed environment $\Gamma \vdash D$

$$\textbf{(Typ activity)}\quad \frac{D(n) = \mathsf{activity}(\mathtt{CALL}, \mathtt{PERMS}, \lambda x.t, \lambda x.t') \quad \Gamma, x : \tau \vdash_{\mathtt{PERMS}} t : \tau' \Rightarrow \tau'' \quad \Gamma, x : \tau' \vdash_{\mathtt{PERMS}} t' : \tau' \Rightarrow \tau''}{\Gamma, n : \mathsf{Activity}(\mathtt{CALL}, \tau \to \tau' \Rightarrow \tau'') \vdash D}$$

$$\textbf{(Typ receiver)}\quad \frac{D(n) = \mathsf{receiver}(\mathtt{SEND}, \mathtt{PERMS}, \lambda x.t) \quad \Gamma, x : \tau \vdash_{\mathtt{PERMS}} t : T}{\Gamma, n : \mathsf{Receiver}(\mathtt{SEND}, \tau \to T) \vdash D}$$

$$\textbf{(Typ provider)}\quad \frac{D(n) = \mathsf{provider}(\mathtt{READ}, \mathtt{WRITE}, v) \quad \Gamma \vdash_{\_} v : \mathsf{Any}(\mathtt{READ}, \mathtt{WRITE})}{\Gamma, n : \mathsf{Provider}(\mathtt{READ}, \mathtt{WRITE}) \vdash D}$$

Well-typed expression, stack $\Gamma \vdash e : T, \Gamma \vdash S : \tau \Rightarrow \tau'$

$$\textbf{(Typ encap)}\quad \frac{\Gamma \vdash_{\mathtt{PERMS}} t : T}{\Gamma \vdash [\mathtt{PERMS}]\ t : T}$$

$$\textbf{(Typ eval)}\quad \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : T}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : T}$$

$$\textbf{(Typ stack)}\quad \frac{\Gamma \vdash e : \tau \Rightarrow \tau' \qquad \Gamma, x : \tau \vdash e' : \tau \Rightarrow \tau' \quad \Gamma \vdash S : \tau' \Rightarrow \_}{\Gamma \vdash \langle e, \lambda x.e' \rangle :: S : \tau \Rightarrow \tau'}$$

Figure 2. Security type system

Figure 2 shows some of our typing rules. (The context PERMS is carried around to derive Stuck types, using rules that are not shown here. We also omit a subtyping rule for Stuck types, which allows us to infer any type for a stuck program.) We elaborate only on **(Typ call)** and **(Typ return)**; the remaining rules rely on standard ideas, and should not be difficult to follow. A program of the form $\mathsf{call}((n, \_))$ is given a stack type of the form $\tau' \Rightarrow \_$ whenever the type of programs that may be run in a window launched by n is of the form $\_ \Rightarrow \tau'$; indeed, values returned by such a window are passed back to the window running $\mathsf{call}((n, \_))$. Dually, the stack type of $\mathsf{return}(v)$ is of the form $\_ \Rightarrow \tau$ whenever v is of type τ.

By **(Typ stack)**, stack types are chained to type a stack of windows. (We assume that ϵ can be given any stack type.)

Finally, we define well-typed states.

DEFINITION 4.1. *A state $D; S; E$ is well-typed if there is Γ such that $\Gamma \vdash D$, $\Gamma \vdash S : \_$, and $\Gamma \vdash e : \_$ for each $e \in E$.*

4.2 Properties of typing

Our main theorem is the following data-flow security property for any sequence of reductions of a well-typed state.

THEOREM 4.2. *Let $D; S; E$ be a well-typed state such that* $\mathsf{provider}(\mathtt{READ}, \mathtt{WRITE}, m) \in \mathtt{rng}(D)$ *for some fresh name m. Suppose that $D; S; E \longrightarrow^\star D'; S'; E'$, such that* $\mathsf{provider}(\mathtt{READ}', \mathtt{WRITE}', m) \in \mathtt{rng}(D')$.

Then $\mathtt{READ}' \sqsupseteq \mathtt{READ}$ *and* $\mathtt{WRITE} \sqsupseteq \mathtt{WRITE}'$.

Informally, this theorem guarantees that a value can flow from provider n to provider n' only if readers of n' may already read n, and writers of n may already write n'. The theorem follows from a standard subject reduction lemma.

4.3 Untyped code

Theorem 4.2 holds only for well-typed states; in particular, it assumes that all components in the environment are well-typed. In practice, this requirement may be too strict—we may wish to consider the possibility of safely running an application even if it does not typecheck.

It turns out that such applications can indeed be run safely with permission $\bot$. Technically, we include a vacuous system of rules in the type system *that can only be applied in a $\bot$ context*; these rules consider $\mathsf{Any}(\bot, \bot)$ as a dynamic type, and allow previously untyped code with permission $\bot$ to typecheck. Conversely, we require all occurrences of PERMS in Figure 2 to be non-$\bot$, so that our core discipline cannot be bypassed.

However, this alone is not enough to recover Theorem 4.2. Typed code can still interact with previously untyped code, and we need to control such interactions. In particular, it should not be possible for previously untyped code to consume values of type $\mathsf{Any}(\mathtt{READ}, \_)$ or produce values of type $\mathsf{Any}(\_, \mathtt{WRITE})$ if READ or WRITE are non-$\bot$. Fortunately, a simple set of constraints in our core discipline (omitted here) suffice to eliminate such flows. Theorem 4.2 now holds for this augmented system.

Finally, we should point out that $\bot$ does not necessarily equate to "no permission" at run time. In fact, the lattice in the type system may be any order-preserving abstraction of the lattice in the operational semantics.

5. Discussion

In Section 1, we point out several possible security concerns of users (such as Alice) and developers (such as Bob) of Android applications. We now outline how our approach can quell those concerns. Indeed, Alice can safely run any well-typed application on her phone. By Theorem 4.2, any such application is guaranteed to preserve the secrecy and integrity of her data. If such an application does not conform to our core discipline, it necessarily runs with permission $\bot$, so Alice can still safely run such an application. Conversely, the only way Bob can convince Alice that his application is safe to run on her phone is by typechecking his application.

The only other study of Android security we are aware of is [9]. It reports a logic-based tool, Kirin, for determining whether the permissions declared by an application satisfy a certain global safety invariant. Typically, this invariant bans permissions that may result in data flows across applications. However, Kirin does not track data flows inside an application; thus, its analysis is necessarily less precise than ours. In particular, for an application that has several components, each of which require a disjoint set of permissions, Kirin conservatively considers the union of those permissions when deciding the safety of the application. In contrast, we track precise dependencies among the components, and thus may recognize the application to be safe even if Kirin cannot. This precision is important in the presence of signatures, which allow possibly unrelated applications to share the same set of permissions.

The approach in this paper is similar to our previous work on formalizing security on Windows Vista [8]. However, while in [8] we merely aim for a formal understanding of Windows Vista's security design, in this setting we can be much more ambitious, as discussed in Section 1. We plan to extend our analysis to application code in Java, and implement a certified installer for Android applications based on our analysis; we expect existing static analysis tools for Java, such as ESC/Java [10], Soot [12], and WALA [7] to provide a convenient foundation for such an implementation. Furthermore, our project bears several similarities with other existing projects for mobile code security, such as Mobius [5], Jif [4], and S2MS [6]. We expect that as we begin implementing our analysis, we will benefit from tools and techniques developed in the context of such projects.

In conclusion, we initiate a formal language-based study of security on Android in this paper. We believe that it is worthwhile to put in the necessary effort into bringing language-based security to the mainstream via this setting.

Acknowledgments

This research is supported in part by DARPA under grant ODOD.HR00110810073.

References

[1] Android developers. `http://developer.android.com/index.html`.

[2] Android market. `http://www.android.com/market/`.

[3] Android project. `http://source.android.com/`.

[4] The Jif project. `http://www.cs.cornell.edu/jif/`.

[5] The Mobius project. `http://mobius.inria.fr/twiki/bin/view/Mobius`.

[6] The S3MS project. `http://www.s3ms.org/index.jsp`.

[7] WALA. `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[8] A. Chaudhuri, P. Naldurg, and S. Rajamani. A type system for data-flow integrity on Windows Vista. In *PLAS'08: Programming Languages and Analysis for Security*, pages 89–100. ACM, 2008.

[9] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security & Privacy Magazine*, 7(1):10–17, 2009.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[11] G. C. Necula. Proof-carrying code. In *POPL'97: Principles of Programming Langauges*, pages 106–119. ACM, 1997.

[12] R. V. Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, 2000.