Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract)

Limin Jia[1], Jassim Aljuraidan[1], Elli Fragkaki[1], Lujo Bauer[1], Michael Stroucken[1], Kazuhide Fukushima[2], Shinsaku Kiyomoto[2], and Yutaka Miyake[2]

[1] Carnegie Mellon University, Pittsburgh, USA
{liminjia,aljuraidan,elli,lbauer,mxs}@cmu.edu
[2] KDDI R&D Laboratories, Inc., Tokyo, Japan
{ka-fukushima,kiyomoto,miyake}@kddilabs.jp

Abstract. Recent years have seen a dramatic increase in the number and importance of mobile devices. The security properties that these devices provide to their applications, however, are inadequate to protect against many undesired behaviors. A broad class of such behaviors is violations of simple information-flow properties. This paper proposes an enforcement system that permits Android applications to be concisely annotated with information-flow policies, which the system enforces at run time. Information-flow constraints are enforced both between applications and between components within applications, aiding developers in implementing least privilege. We model our enforcement system in detail using a process calculus, and use the model to prove noninterference. Our system and model have a number of useful and novel features, including support for Android's single- and multiple-instance components, floating labels, declassification and endorsement capabilities, and support for legacy applications. We have developed a prototype of our system on Android 4.0.4 and tested it on a Nexus S phone, verifying that it can enforce practically useful policies that can be implemented with minimal modification to off-the-shelf applications.

1 Introduction

Recent years have seen a dramatic increase in the number and importance of smartphones and other mobile devices. The security properties that mobile operating systems provide to their applications, however, are inadequate to protect against many undesired behaviors, contributing to the rapid rise in malware targeting mobile devices [27,20].

To mitigate application misbehavior, mobile OSes like Android rely largely on strong isolation between applications and permission systems that limit communication between applications and access to sensitive APIs. Researchers have investigated these mechanisms, finding them vulnerable to application collusion [32,21], information-flow leaks [32,12], and privilege-escalation attacks [9,13]. Attempts to address these issues have produced tools for detecting information leaks [11,7,17], improvements to permission systems (e.g., [26,24]), as well as other mechanisms for restricting applications' access to data and resources (e.g., [5]).

Many common misbehaviors that are beyond the reach of Android's permission system are violations of simple information-flow properties. This is because Android's

permission system supports only those policies that allow or deny communication or access to sensitive resources based on the (mostly static) permissions of the caller and callee. Once data has been sent from one application to another, the sender has relinquished all control over it.

Recent work on preventing undesired information flows on Android typically focuses on using a specific mechanism to enforce a pre-determined global policy [11,7]. Other works have developed more powerful mechanisms that track control flow and allow finer-grained control over communication and resource accesses [10,5]; these also typically lack a convenient policy language. Although a few formal analyses of Android's security architecture have provided some insight about its limitations [33], works that introduce more powerful mechanisms typically do not formally investigate the properties that those mechanisms exhibit.

This paper fills many of these gaps by proposing a DIFC-style enforcement system for Android that allows convenient, high-level specification of policy and has a well-understood theory, backed by a proof of noninterference. Building on techniques for controlling information flow in operating systems [19,36], our system permits policy to be specified via programmer- or system-defined labels applied to applications or application components. Enforcing information-flow policies at the level of application components is a practically interesting middle ground between process- (e.g., [19]) and instruction-level (e.g., [23]) enforcement, offering finer-grained control than process-level enforcement, but retaining most of its convenience. Labels specify a component's or application's secrecy level, integrity level, and declassification and endorsement capabilities. We also allow floating labels, which specify the minimal policy for a component, but permit multipurpose components (e.g., an editor) to be instantiated with labels derived from their callers (e.g., to prevent them from exfiltrating a caller's secrets).

We develop a detailed model of our enforcement system using a process calculus, using which we prove noninterference. The modeling—and the design of the system—is made particularly challenging by the desire to fully support key features of Android's programming model. Challenging features include single- and multiple-instance components and enforcement at two levels of abstraction—at the level of applications, which are strongly isolated from each other, and at the level of application components, which are not. Our formal analysis reveals that floating labels and the ability of single-instance components to make their labels stricter at run time—features that appear necessary to support practical scenarios—can, if not implemented carefully, easily compromise the noninterference property of the system.

Proving noninterference was also challenging because traditional ways in which information-flow systems are modeled in process calculi do not directly apply to Android: the security level of the channel through which an Android component communicates changes as the system executes. To model this, we enhance pi-calculus with a labeled process, $\ell[P]$, to associate each component with its run-time security level. The labeled process and the techniques for specifying noninterference can be applied to the modeling of other distributed systems, such as web browsers.

The contributions of this paper are the following:

1. We propose the first DIFC-style enforcement system for Android that allows convenient, high-level specification of policy and has a well-understood theory (§3).

2. We develop a faithful process-calculus model of Android's main programming abstractions and our system's enforcement mechanism (§4).
3. We define noninterference for our enforcement system and prove that it holds (§5), in the presence of dynamically changing security levels of components.
4. We implement our system on Android 4.0.4 and test it on a Nexus S phone; through a case study with minimally modified off-the-shelf applications, we show that our system can specify and enforce practically interesting policies (§6).

For space reasons, we omit many details, which appear in our technical report [1].

2 Background and Related Work

In this section we briefly introduce Android and review related work.

Android Overview Android is a Linux-based OS; applications are written in Java and each executes in a separate Dalvik Virtual Machine (DVM) instance. Applications are composed of *components*, which come in four types: *activities* define a specific user interface (e.g., a dialog window); *services* run in the background and have no user interface; *broadcast receivers* listen for system-wide broadcasts; and *content providers* provide an SQL-like interface for storing data and sharing them between applications.

Activities, services, and broadcast receivers communicate via asynchronous messages called *intents*. If a recipient of an intent is not instantiated, the OS will create a new instance. The recipient of an intent is specified by its class name or by the name of an "action" to which multiple targets can subscribe. Any component can attempt to send a message to any other component. The OS mediates both cross- and intra-application communications via intents. Between applications, intents are the only (non-covert) channel for establishing communication. Components within an application can also communicate in other ways, such as via public static fields. Such communication is not mediated, and can be unreliable because components are short lived—Android can garbage collect all but the currently active component. Hence, although Android's abstractions do not prevent unmediated communication between components, the programming model discourages it. We will often write that a component *calls* another component in lieu of explaining that the communication is via an intent.

Android uses *permissions* to protect components and sensitive APIs: a component or API protected by a permission can be called only by applications that hold this permission. Permissions are strings (e.g., android.permission.INTERNET) defined by the system or declared by applications. Applications acquire permissions only at install time, with the user's consent. Additionally, content providers use *URI permissions* to dynamically grant and revoke access to their records, tables, and databases.

Related Work We discuss two categories of most closely related work.

Information Flow Enforcing information-flow policies has been an active area of research. Some develop novel information-flow type systems (cf. [31]) that enforce noninterference properties statically; others use run-time monitoring, or hybrid techniques (e.g., [8,29,22,2,3,16]). These works track information flow at a much finer level of granularity than ours; in contrast, the goals of our design included minimally impacting legacy code and run-time performance on Android.

Our approach is most similar to work on enforcing information-flow policies in operating systems [37,35,19]. There, each process is associated with a label. The components in our system can be viewed as processes in an operating system. However, most of these works do not prove any formal properties of their enforcement mechanisms. Krohn et al. [18] presented one of the first proofs of noninterference for practical DIFC-based operating systems. Our design is inspired by Flume [19], but has many differences. For instance, Flume does not support floating labels. In Android, as we show through examples, floating labels are of practical importance. Because Flume has no floating labels, a stronger noninterference can be proved for it than can be proved for our system: Flume's definition of noninterference is based on a stable failure model, a simulation-based definition. Our definition is trace-based, and does not capture information leaks due to a high process stalling.

A rich body of work has focused on noninterference in process calculi [14,30]. Recently, researchers have re-examined definitions of noninterference for reactive systems [4,28]. In these systems, each component waits in a loop to process input and produce one or more outputs (inputs to other components). These works propose new definitions of noninterference based on the (possibly infinite) streams produced by the system. Our definition of noninterference is weaker, since we only consider finite prefixes of traces. These reactive models are similar to ours, but do not consider shared state between components, and assume the inputs and outputs are the only way to communicate, which is not the case for Android. Further, to model the component-based Android architecture more faithfully, we extend pi-calculus with a label context construct, which also enables formal analysis of our enforcement mechanism in the presence of Android components' ability to change their labels at run time. To our knowledge, such dynamic behavior has rarely been dealt with in the context of process calculus.

Android Security Android's permission system has been shown inadequate to protect against many attacks, including privilege-escalation attacks [9,13] and information leaks [11,32,6,12]. Closest to the goal of our work are projects such as TaintDroid [11] and AppFence [17], which automatically detect and prevent information leaks. They operate at a much finer granularity than our mechanism, tracking tainting at the level of variables, enforce fixed policies, and have not been formally analyzed.

Formal analyses of Android-related security issues and language-based approaches to solving them have received less attention. Shin et al. [33] developed a formal model to verify functional correctness properties of Android, which revealed a flaw in the permission naming scheme [34]. Our prior work proposed a set of enhancements to Android's permission system designed to enforce information-flow-like policies, for which some correctness properties were also formally proved [15]. The work described in this paper is different in several respects: we build on more well-understood theory of information flow; we support more flexible policies (e.g., in prior work it is not possible to specify that information should not leak to a component unless that component is protected by some permission); we make persistent state more explicit; and we formally model our enforcement system in much greater detail, thus providing much stronger correctness guarantees. The labeled context used in the modeling and the techniques developed for specifying noninterference in this paper can be applied to other systems,

and we view the formalism as a main contribution of this paper. In comparison, the proofs in our prior work are customized for that specific system.

3 Enforcing Information-Flow Properties

We next describe a scenario that exemplifies the Android permission system's inability to implement many simple, practical policies (§3.1). We then discuss key aspects of our system and show it can specify (§3.2) and enforce (§3.3) richer policies.

3.1 Motivating Scenario

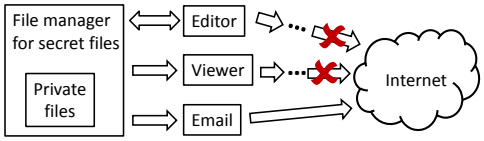

Fig. 1: A simple scenario that cannot be implemented using Android permissions.

Suppose a system has the following applications: a *secret-file manager* for managing files such as lists of bank-account numbers; a general-purpose text *editor* and a *viewer* that can modify and display this content; and an *email application*. Because of their sensitive content, we want to prevent files managed by the secret-file manager from being inadvertently or maliciously sent over the Internet; this should be allowed only if the user explicitly requests it through the file manager. This scenario is shown in Figure 1.

The desired (information-flow) policy is representative of practical scenarios that Android currently does not support. In Android, one might attempt to prevent the editor or viewer from exfiltrating data by installing only viewers and editors that lack the Internet permission; these then could not send out secret data directly, but they could still do so via another application that has the Internet permission (e.g., the email client).

3.2 Key Design Choices

In general, the attacker model we consider is one where an application may try to exfiltrate data or access sensitive resources that it is not permitted to access, including by taking advantage of cooperating or buggy applications.

We now discuss the design of our system and explain how it can specify and enforce our desired example policy. We revisit the example more concretely in §6.

Enforcement Granularity Traditionally, information-flow properties have been enforced either at instruction level (e.g., [23,16]) or at process level (e.g., [19]). Android's division of applications into components invites the exploration of an interesting middle ground between these two. Android applications are typically divided into a relatively few key components, e.g., an off-the-shelf file manager with which we experimented was comprised of five components. Hence, component-level specification would likely not be drastically more complex than application-level specification. This additional granularity, however, could enable policies to be more flexible and better protect applications (and components) from harm or misuse.

Unfortunately, enforcing purely component-level policies is difficult. The Android programming model strongly encourages the use of components as modules. In fact,

the Android runtime may garbage collect any component that is not directly involved in interacting with the user; using Android's narrow interface for communication between components is the only reliable method of cross-component communication. However, neither Android nor Java prevent components *that belong to the same application* from exchanging information without using the Android interfaces, e.g., by directly writing to public static fields. Hence, Android's component-level abstractions are not robust enough to be used as an enforcement boundary; fully mediating interactions between components would require a lower-level enforcement mechanism. Although such enforcement is possible, e.g., with instruction-level information-flow tracking [23], implementation and integration with existing platforms and codebases is difficult and can cause substantial run-time overhead.

We pursue a hybrid approach. We allow policy specification at both component level and application level. Enforcement of component-level policies is best-effort: When programmers adhere to Android's programming conventions for implementing interactions between components, most potential policy violations that are the result of application compromise or common programmer errors will be prevented by the enforcement system. On the other hand, the components of a non-conformant application will be able to circumvent its component-level policy (but not its application-level policy, nor other applications' application- or component-level policy). Thus, component-level policies are a tool to help programmers to better police their own code and implement least privilege, and also act in concert with application-level policy to regulate cross-application interactions at a more fine-grained level. However, application-level policies are enforced strictly because Android provides strong isolation between applications.

Policy Specification via Labels We use labels to express information-flow policies and track information flows at run time. A *label* is a triple (s, i, δ), where s is a set of *secrecy tags*, i a set of *integrity tags*, and δ a set of *declassification and endorsement capabilities*. For convenience, we also refer to s as a secrecy label and i as an integrity label; and to δ as the set of declassification capabilities, even though δ also includes endorsement capabilities. Labels are initially assigned to applications and components by developers in each application's manifest; we call these *static* labels. At run time, each application and component also has an *effective* label, which is derived by modifying the static label to account for declassification and endorsement. Additionally, secrecy labels s and integrity labels i can be declared as *floating*; we explain this below.

Labels as Sets of Tags Implementing secrecy and integrity labels as sets of tags was motivated by the desire to help with backward compatibility with standard Android permissions. In Android, any application can declare new permissions at installation time. We similarly allow an application to declare new secrecy and integrity tags, which can then be used as part of its label. The lattice over labels, which is required for enforcement, does not need to be explicitly declared—this would be impractical if different applications declare their own tags. Rather, the lattice is defined by the subset relation between sets of tags. The permissions that legacy applications possess or require of their callers can be mapped to tags and labels. A more detailed discussion can be found in our technical report [1].

Declassification and Endorsement The declassification capabilities, δ, specify the tags a component or application may remove from s or add to i. We make the declassification

capabilities part of the label, because whether a component may declassify or endorse is a part of the security policy. Declaratively specifying declassification policy makes it easier to reason about and aids backward compatibility: declassification (or endorsement) that is permitted by policy can be applied to a legacy application or component automatically by the enforcement system when necessary for a call to succeed.

Returning to the example from §3.1: The secret-file manager application may be labeled with the policy ({FileSecret}, {FileWrite}, {-FileSecret}). Intuitively, the first element of this label conveys that the secret-file manager is tainted with the secret files' secrets (and no other secrets); the second element that the file manager has sufficient integrity to add or change the content of files; and the third element that the file manager is allowed to declassify. The file manager's effective label will initially be the same as this static label. If the file manager exercises its declassification capability -FileSecret, its effective label will become ({}, {FileWrite}, {-FileSecret}).

The complement to declassification and endorsement is *raising* a label. Any component may make its effective secrecy label more restrictive by adding tags to it, and its effective integrity label weaker by removing tags. After a component has finished executing code that required declassification or endorsement, it will typically raise its effective label to the state it was in prior to declassification or endorsement. Components without declassification capabilities can also raise their labels, but this is rarely likely to be useful, since raising a label can be undone only by declassifying or endorsing.

Floating Labels Some components or applications, e.g., an editor, may have no secrets of their own but may want to be compatible with a wide range of other applications. In such cases, we can mark the secrecy or integrity label as *floating*, e.g., (F{}, F{}, {}), to indicate that the secrecy or integrity element of a component's effective label is inherited from its caller. The inheriting takes place only when a component is instantiated, i.e., when its effective label is first computed. Floating labels serve a very similar purpose to polymorphic labels in Jif [23].

In our example, the editor's static policy is (F{}, F{}, {}). If instantiated by the file manager, the editor's effective secrecy label becomes {FileSecret}, allowing the editor and the file manager to share data, but preventing the editor from calling any applications or APIs that have a secrecy label weaker than {FileSecret}. If the editor also had secrets to protect, we might give it the static label (F{EditorSecret}, F{}, {}). Then, the editor's effective label could be floated to ({EditorSecret, FileSecret}, {}, {}), but any instantiation of the editor would carry an effective secrecy label at least as restrictive as {EditorSecret}. Similarly, when the editor is instantiated by the file manager, its static integrity label F{} would yield an effective integrity label {FileWrite}, permitting the editor to save files, and preventing components without a FileWrite integrity tag from sending data to the editor.

Unlike secrecy and integrity labels, declassification capabilities cannot be changed dynamically; they are sufficiently powerful (and dangerous) that allowing them to be delegated is too likely to yield a poorly understood policy.

3.3 Enforcement Approach and Limitations

We described in §3.2 how to specify rich, practically useful policies in our system; we next outline how they are enforced. The crux of our enforcement system is a reference

monitor that intercepts calls between components, which we build on top of Android's activity manager (§6). Much of its responsibility is maintaining the mapping from applications and components (and their instances) to their effective labels. Our formal model (§4) abstracts the bookkeeping responsibilities into a *label manager* and the purely enforcement duties into an *activity manager*. We next discuss how our reference monitor makes enforcement decisions and how our system handles persistent state.

Application- and Component-level Enforcement When two components try to communicate via an intent, our reference monitor permits or denies the call by comparing the caller's and the callee's labels. When the caller and callee are part of the same application, the call is allowed only if the caller's effective secrecy label is a subset of the callee's and the caller's effective integrity label is a superset of the callee's. The comparison is more interesting when the caller and callee are in different applications. Then, a call is allowed if it is consistent with both component-level labels and application-level labels of the caller's and callee's applications.

If the callee component (and application) has a floating (static) label, the callee's effective integrity label is constructed as the union of its static integrity label and effective integrity labels of the caller and the caller's application. The effective secrecy label (and the callee's application's effective labels) is constructed similarly.

Declassification and endorsement change the effective labels of components and applications, and are permitted only when consistent with policy. For programmer convenience, the reference monitor will automatically declassify or endorse a caller component when this is necessary for a call to be permitted. We discuss this further in §6.

From the standpoint of policy enforcement, returns (from a callee to a caller), including those that report errors, are treated just like calls. As a consequence, a return may be prohibited by policy (and prevented) even if a call is allowed.

Much of the functionality of Android applications is accomplished by calling Android and Java APIs, e.g., for accessing files or opening sockets. We assign these APIs labels similarly as we would to components. For instance, sending data to sockets potentially allows information to be leaked to unknown third parties; therefore, we assign a label with an empty set of secrecy tags to the socket interface to prevent components with secrets from calling that API. We treat globally shared state, e.g., individual files, as components, and track their labels at run time.

Persistent State Multi-instance components intuitively pose little difficulty for enforcing information-flow policies: each call to such a component generates a fresh instance of the component bereft of any information-flow entanglements with other components.

More interesting are single-instance components, which can be targets for multiple calls from other components, and whose state persists between those calls. Interaction between single-instance components and the ability of components to raise their labels can at first seem to cause problems for information-flow enforcement.

Consider, for example, malicious components A and B that seek to communicate via a colluding single-instance component C. Suppose that A's static secrecy label is $\{\mathsf{FileSecret}\}$ and B's is $\{\}$, preventing direct communication from A to B; C's static secrecy label is $\{\}$. Component C, upon starting, sends B an intent, then raises its effective label to $\{\mathsf{FileSecret}\}$. A sends the content of a secret file to C; their labels permit this. If the content of the secret file is "Attack," C exits; otherwise, C continues running.

B calls C, then calls C again. If B receives two calls from C, then it learns that A's secret file is "Attack." C can only make the second call to B after exiting, which only happens when A's secret file is "Attack." The information leak arose because C changed its label by exiting. To prevent such scenarios (and to allow us to prove noninterference, which ensures that no similar scenarios remain undiscovered), raising a label must change not only a component's effective label, but also its static label.

Limitations We do not address communication via covert channels, e.g., timing channels. Recent work has identified ways in which these may be mitigated by language-based techniques [38]; but such techniques are outside the scope of this paper. We also do not address the robustness of Android's abstractions: stronger component-level abstractions would permit robust, instead of best-effort, enforcement of information-flow policies within applications. Improving these abstractions, or complementing them by, e.g., static analysis, could thus bolster the efficacy of our approach.

Many security architectures are vulnerable to user error. On Android, a user can at installation time consent to giving an application more privileges than is wise. Our system does not address this; we design an infrastructure that supports rich, practically useful policies. Because our approach allows developers to better protect their applications, they may have an incentive to use it. However, we do not tackle the problem of preventing the user from making poor choices (e.g., about which applications to trust).

4 Process Calculus Model

We next show a process calculus encoding of Android applications and our enforcement mechanism. The full encoding captures the key features necessary to realistically model Android, such as single- and multi-instance components, persistent state within component instances, and shared state within an application. Many details that we omit for brevity can be found in our technical report [1].

4.1 Labels and Label Operations

Labels express information-flow policies and are used to track flows at run time. A *label* is composed of sets of *tags*. We assume a universe of secrecy tags $\mathcal{S}$ and integrity tags $\mathcal{I}$. Each secrecy tag in $\mathcal{S}$ denotes a specific kind of secret, e.g., contact information. Each integrity tag in $\mathcal{I}$ denotes a capability to access a security-sensitive resource.

Simple labels $\kappa ::= (\sigma, \iota)$ *Process labels* $K ::= (Q(\sigma), Q(\iota), \delta)$ where $Q = C$ or $Q = F$

A simple label κ is a pair of a set of secrecy tags σ drawn from $\mathcal{S}$ and a set of integrity tags ι drawn from $\mathcal{I}$. Simple labels form a lattice $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set of simple labels and $\sqsubseteq$ is a partial order over simple labels. Intuitively, the more secrecy tags a component has, the more secrets it can gather, and the fewer components it can send intents to. The fewer integrity labels a component has, the less trusted it is, and the fewer other components it can send intents to. Consequently, the partial order over simple labels is defined as follows: $(\sigma_1, \iota_1) \sqsubseteq (\sigma_2, \iota_2)$ iff $\sigma_1 \subseteq \sigma_2$, and $\iota_2 \subseteq \iota_1$

Secrecy and integrity labels are annotated with C for concrete labels or F for floating labels. A process label K is composed of a secrecy label, an integrity label, and a set of declassification capabilities δ. An element in δ is of the form $-t_s$, where $t_s \in \mathcal{S}$,

AM Erasure of label	$C(\sigma)^- = \sigma$	$F(\sigma)^- = \top$	$(Q(\sigma), Q(\iota), \delta)^- = ((Q(\sigma))^-, \iota)$
PF Erasure of label	$C(\iota)^* = \iota$	$F(\iota)^* = \top$	$(Q(\sigma), Q(\iota), \delta)^* = (\sigma, (Q(\iota))^*)$
Label Declassify	$(C(\sigma), C(\iota), \delta) \uplus_d \delta_1 =$		
	$(C(\sigma \backslash \{t \mid (-t) \in \delta_1\}), C(\iota \cup \{t \mid (+t) \in \delta_1\}), \delta)$		

Fig. 2: Selected label operations.

or $+t_i$, where $t_i \in \mathcal{I}$. A component with capability $-t_s$ can remove the tag t_s from its secrecy tags σ; a component that has $+t_i$ can add the tag t_i to its integrity tags ι.

We define operations on labels (Figure 2). An *AM erasure* function K^- is used by the activity manager to reduce process labels to simple labels that can easily be compared. This function removes the declassification capabilities from K, and reduces a floating secrecy label to the top secrecy label. This captures the idea that declassification capabilities are not relevant to label comparison, and that a callee's floating secrecy label will never cause a call to be denied. The *PF* erasure function K^* is used in defining noninterference, and is explained in §5. The declassification operation $K \uplus_d \delta_1$ removes from K the secrecy tags in δ_1, and adds the integrity tags in δ_1.

4.2 Preliminaries

We chose a process calculus as our modeling language because it captures the distributed, message-passing nature of Android's architecture. The Android runtime is the parallel composition of component instances, application instances, and the reference monitor, each modeled as a process.

The syntax of our modeling calculus, defined below, is based on π-calculus. We use $'|'$ for parallel composition, and reserve $|$ for BNF definitions. aid denotes an application identifier, and cid a component identifier, both drawn from a universe of identifiers. c denotes constant channel names. Specific interfaces provided by an application or component are denoted as $aid{\cdot}c$ and $aid{\cdot}cid{\cdot}c$.

The only major addition is the labeled process $\ell[\,P\,]$. Label contexts ℓ include the unique identifiers for applications (aid) and components (cid), channel names (c) that serve as identifiers for instances, and a pair (ℓ_1, ℓ_2) that represents the label of a component and its application. Bundling a label with a process aids noninterference proofs by making it easier to identify the labels associated with a process.

Names	$a ::= x \mid c \mid aid{\cdot}c \mid aid{\cdot}cid{\cdot}c$	*Proc*	$P ::= \mathbf{0} \mid \mathsf{in}\, a(x).P \mid \mathsf{in}\, a(\mathsf{patt}).P$
Label ctx	$\ell ::= aid \mid cid \mid c \mid (\ell_1, \ell_2)$		$\mid \mathsf{out}\, e_1(e_2).P \mid P_1 + P_2 \mid \nu x.P \mid !P$
Expr	$e ::= x \mid a \mid ctr\, e_1 \cdots e_k$		$\mid (P_1\, '|'\, P_2) \mid \ell[\,P\,] \mid \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$
	$\mid (e_1, \cdots, e_n)$		$\mid \mathsf{case}\ e\ \mathsf{of}\{ctr_1 \boldsymbol{x}_1 \Rightarrow P_1 ...$
			$\mid ctr_n \boldsymbol{x}_n \Rightarrow P_n\}$

We extend the standard definition of a process P with if statements, pattern-matching statements, and a pattern-matched input $\mathsf{in}\, x(\mathsf{patt})$ that accepts only outputs that match with patt. These extensions can be encoded directly in π-calculus, but we add them as primitive constructors to simplify the representation of our model.

$$
\begin{array}{llcl}
\textit{Application} & \textit{App}(aid) & = & aid[!(\mathsf{in}\ aid{\cdot}c_L(c_{AI}).c_{AI}[\textit{AppBody}(aid, c_{AI})])] \\
\textit{App body} & \textit{AppBody}(aid, c_{AI}) & = & \nu c_{svL}.\nu c_{sv}.\mathsf{out}\ c_{svL}(s_0).(SV(c_{svL}, c_{sv})\ '|'\ \\
 & & & (c_{AI}, cid_1)[CP_1(aid, cid_1, c_{AI}, c_{sv})]\ '|'\ \cdots \\
 & & & '|'\ (c_{AI}, cid_n)[CP_n(aid, cid_n, c_{AI}, c_{sv})]) \\
\textit{Component} & CP(aid, cid, c_{AI}, c_{sv}) & = & !(\mathsf{in}\ aid{\cdot}cid{\cdot}c_{cT}(_{=}c_{AI}, I, c_{nI}, c_{lock}, rt). \\
 & & & (c_{AI}, c_{nI})[...\mathsf{in}\ c_{nI}(I).\langle\mathsf{out}\ I(\mathsf{self})\rangle.A(...)]) \\
\textit{Comp body} & A(cid, aid, I, c_{AI}, rt, ...) & ::= & ...\ |\ \mathsf{out}\ a_m(\mathsf{call}_I, rt, aid, c_{AI}, cid_{ce}, I)\ '|'\ A(...)
\end{array}
$$

Fig. 3: Partial encoding of applications and components.

4.3 A Model of Android and Our Enforcement Architecture

We model as processes the three main constructs necessary to reason about our enforcement mechanism: application components, the activity manager, and the label manager. The activity manager is the part of the reference monitor that mediates calls and decides whether to allow a call based on the caller's and the callee's labels. The label manager is the part of the reference monitor that keeps track of the labels for each application, component, and application and component instance.

Life-cycles of Applications and Components and Their Label Map A large part of the modeling effort is spent on ensuring that the process model faithfully reflects the life-cycles of applications and components, which is crucial to capturing information flows through persistent states within or across the life-cycles. The reference monitor maintains a label map Ξ, which reflects these life-cycles.

Android supports single- and multi-instance components. Once created, a single-instance component can receive multiple calls; the instance body shares state across all these calls. A fresh instance of a single-instance component is created only when the previous instance has exited and the component is called again. A component does not share state across its instantiations. For a multi-instance component, a new instance is created on every call to that component. An application is similar to a single-instance component, and all component instances within one application instance share state.

All calls are asynchronous; returning a result is treated as a call from the callee to the caller. When a component instance is processing a call, any additional intents sent to that instance (e.g., new intents sent to a single-instance component, or results being returned to a multi-instance component) are blocked until the processing has finished.

Encoding Applications and Components A partial encoding of applications and components is shown in Figure 3. We delay explaining the label contexts $\ell[...]$ until §5—they are annotations that facilitate proofs, and have no run-time meaning.

We encode a recursive process using the ! operator. A process $!(\mathsf{in}\ c(x).P)$ will run a new process P each time a message is sent to c. This models the creation of a run-time instance of an application or a component. In both cases, we call channel c the *launch channel* of P, and say that P is *launched* from c.

An application $App(aid)$ with ID aid is the parallel composition of a shared state SV and components $CP_i(aid, cid_i, c_{AI}, c_{sv})$. Each application has a designated launch channel $aid{\cdot}c_L$. The channel c_{AI}, passed as an argument to the launch channel, serves as a unique identifier for an application instance. Once an application is launched, it

0 $AM_I = \mathsf{in}\, a_m(\mathsf{call}_I, kA_{cr}, kC_{cr}, rt, aid, c_{AI}, cid_{ce}, I).$
1 $\nu c.\ \mathsf{out}\, t_m(\mathsf{lookUp}, cid_{ce}, c).\ \mathsf{in}\, c(s).$
2 $\mathsf{case}\ s\ \mathsf{of}\ \ldots$
18 $\mid \mathsf{M}(k_{ce}) \Rightarrow \mathsf{if}\ k_{cr}{}^{-} \sqsubseteq k_{ce}{}^{-}$
19 $\mathsf{then}\ \nu c_{nI}.\nu c_{lock}.\ \mathsf{out}\, t_m(\mathsf{upd}, \{\ (c_{nI}, (c_{lock}, k_{ce} \lhd kC_{cr}{}^{-})), \ldots\}).$
20 $(aid, (c_{AI}, c_{nI}))[\mathsf{out}\ aid{\cdot}aid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, rt)]$
21 $\mathsf{else}\ \mathbf{0}$

Fig. 4: Partial encoding of the activity manager.

launches the shared state. At this point, the application's components are ready to receive calls, and we call this application instance an *active launched* instance.

A component $CP(aid, cid, c_{AI}, c_{sv})$ is launched from a designated creation channel $aid{\cdot}cid{\cdot}c_{cT}$ after a message is received on that channel. The message is a tuple $(_{=}c_{AI}, I, c_{nI}, c_{lock}, rt)$ whose first argument ($_$) must match the current application instance (c_{AI}). I is the intent conveyed by the call. c_{nI} is the new intent channel for the component to process multiple calls. c_{lock} is the channel used to signal the reference monitor that this instance has finished processing the current intent and is ready to receive a new one. Finally, *rt* contains information about whether and on what channel to return a result. A components receives messages on the new intent channel, then proceeds to execute its body (denoted A).

The body of a component is defined in terms of the operations that a component can perform. It is parameterized over several variables, which are free in the body and are bound by outer-layer constructs. A component can use if and case statements, and read or write to the shared state in its application. It can also request from the label manager to change its (and its application's) label, and can call another component by sending a request to the activity manager. All of these operations are encoded using the process calculus. E.g., the call operation is encoded as $\mathsf{out}\, a_m(\mathsf{call}_I, rt, \ldots)$, where a_m is the designated channel to send requests to the activity manager.

Label Manager & Activity Manager The label manager T_M maintains the label map Ξ and processes calls to update the mapping through a designated channel t_m.

Android's activity manager mediates all intent-based communication between components, preventing any communication that is prohibited by policy. The top-level process of the activity manager is of the form: $A_M = !(AM_I + AM_E + AM_{EX} + AM_R)$. The activity manager processes four kinds of calls: AM_I processes calls between components within the same application; AM_E processes inter-application calls; AM_{EX} processes exits, and AM_R processes returns.

We show an example of processing calls between components within the same application (Figure 4): When the activity manager receives a request to send intent I to a component cid_{ce}, it asks the label manager for the callee's label. A possible reply is one that indicates that the callee is a multi-instance component ($\mathsf{M}(k_{ce})$). The activity manager allows the call if the caller's label is lower than or equal to the callee's. If the call is permitted, a new callee instance is launched. To do this, the activity manager (1) generates a new-intent channel and a lock channel for the new instance; (2) updates the

label mapping to record the label of this new active instance; and (3) sends a message containing the intent to the callee's creation channel.

Overall System We assume that an initial process init bootstraps the system and launches the label manager with the static label map that reflects the labels of applications and components at install time, and then calls the first process with fixed labels: $S = T_M \mid A_M \mid App_1(aid_1) \mid ... \mid App_n(aid_n) \mid \mathsf{init}$.

5 Noninterference

To show that our system prevents information leakage, we prove a noninterference theorem. We use the simple label κ_L as the label of malicious components. We call components whose labels are not lower than or equal to κ_L *high components*, and others *low components*. Low components are considered potentially controlled by the attacker. We want to show that a system S that contains both high and low components behaves the same as a system composed of only the low components in S.

Choice between Trace & Bisimulation-based Equivalence Processes P and Q are trace equivalent if for any trace generated by P, Q can generate an equivalent trace, and vice versa. Another commonly-used equivalence, barbed bisimulation, is stronger: it additionally requires those two processes to simulate each other after every τ transition.

Our decision about which notion of process equivalence to use for our noninterference definition is driven by the functionality required of the system so that practically reasonable policies can be implemented. As discussed earlier, floating labels are essential to implement practical applications in Android. However, allowing an application (or single-instance component) to have a floating label weakens our noninterference guarantees: In this case, we cannot hope to have bisimulation-based noninterference (see our technical report [1] for an example).

Rather than disallowing floating labels, we use a weaker, trace-equivalence-based definition of noninterference. This still provides substantial assurance of our system's ability to prevent disallowed information flows: noninterference would not hold if our system allowed: (1) explicit communication between high and low components; or (2) implicit leaks in the reference monitor's implementation, such as branching on data from a high component affecting low components differently depending on the branch.

High and Low Components Most commonly seen techniques that classify high and low events based on a fixed security level assigned to each channel cannot be directly applied to the Android setting, as the components may declassify, raise, or instantiate their labels at run time. Whether an input (output) is a high or low event depends on the run-time label of the component that performs the input (output). Similarly, whether a component is considered high or low, also depends on its run-time label. This makes the definitions and proofs of noninterference more challenging. To capture such dynamic behavior, we introduce the label contexts of processes, and use the run-time mapping of these labels in the label manager to identify the high and low components in the system. The current label of a process can be computed from its label context and the label map Ξ. For a process with nested label contexts $\ell_1[...\ell_n[P]...]$, the innermost label ℓ_n reflects the current label of process P.

Our mechanism enforces information-flow policies at both component and application level; we consequently define noninterference to demonstrate the effectiveness of the enforcement at both levels. Next, we explain how to use the application ID, the component-level label, and the application-level label to decide whether a process is high or low for our noninterference theorem.

Without loss of generality, we pick one application whose components do not access the shared state of that application, and decide whether each of its components is high or low solely based on each component's label; all other applications, whose components may access the shared applicate state, are treated as high or low at the granularity of an application, based on their application-level labels. We write aid_c to denote the specific application whose components we treat as individual entities and disallow their accesses to the shared state.

Now we can define the procedure of deciding whether a process is high or low. We first define a binary relation $\sqsubseteq_{aid_c}$ between a label context $(aid, (\kappa_1, \kappa_2))$ and a simple label κ. We say that $(aid, (\kappa_1, \kappa_2))$ is lower than or equal to κ relative to aid_c. This relation compares the application-level label (κ_1) to κ_L if the application is not aid_c, and compares the component-level label (κ_2) to κ_L if the application ID is aid_c.

$(aid, (\kappa_1, \kappa_2)) \sqsubseteq_{aid_c} \kappa_L$ iff $\kappa_1 \sqsubseteq \kappa_L$ when $aid \neq aid_c$ and $\kappa_2 \sqsubseteq \kappa_L$ when $aid = aid_c$

Now, given the label map Ξ, let $\Xi\langle c\rangle$ denote the label associated with a channel name c in Ξ. We say that a process of the form $aid[...(c_{AI}, c_{nI})[P]...]$ is a low process with regard to κ_L if $(aid, ((\Xi\langle c_{AI}\rangle)^*, (\Xi\langle c_{nI}\rangle)^*) \sqsubseteq_{aid_c} \kappa_L$; otherwise, it is a high process. Please see our tech report for a formal definition and additional details [1].

The function K^* (Figure 2) removes the declassification capabilities in K, and reduces floating integrity labels to the lowest integrity label (on the lattice). This is because a call to a component with a floating integrity label may result in a new instance with a low integrity label, a low event observable by the attacker; hence, a floating component should always be considered a low component.

Traces The actions relevant to our noninterference definitions are intent calls received by an instance, since the only explicit communication between the malicious components (applications) and other parts of the system is via intents. We model intents I as channels. The encoding of components includes a special output action $\langle \mathsf{out}\, I(\mathsf{self})\rangle$ right after the component receives a new intent (Figure 3). This outputs to the intent channel the current labels of the component, denoted by self. Traces consist of these outputs ($\mathsf{out}\, I(aid, (kA, kC))$), which contain information about both what the recipient has learned and the security label of the recipient. We call such an action low, if $(aid, (kA, kC)) \sqsubseteq_{aid_c} \kappa_L$, and high otherwise.

We restrict the transition system to force the activity manager's processing of a request—from receiving it to denying, allowing, or delaying the call—to be atomic. Some requests require that a lock be acquired; we assume the activity manager will only process a request if it can grab the lock. This matches reality, since the run-time monitor will process one call at a time, and the run-time monitor's internal transitions are not visible to the outside world. We write a small-step Android-specific transition as $S \xrightarrow{\alpha}_A S'$, and $S \xRightarrow{\tau}_A S'$ to denote zero or multiple τ transitions from S to S'.

Noninterference We define the projection of traces $t|^{aid_c}_{\kappa_L}$, which removes all high actions from t. The function $\mathsf{projT}(\Xi; \kappa_L; aid_c)$ removes from Ξ mappings from IDs

or channel names to high labels. Similarly, $\mathsf{proj}(P, \kappa_L, aid_c, \Xi)$ removes high components, applications, and instances from P. The resulting configuration is the low system that does not contain secrets or sensitive interfaces.

We say that a declassification step is *effective* with regard to κ_L and aid_c if the label of the declassified instance before the step is not lower than or equal to κ_L relative to aid_c, and the label after is. We call a sequence of transitions $\overset{t}{\Longrightarrow}_A$ *valid* if each step preserves the application-level label of aid_c (application aid_c cannot exit the application or raise its application-level label), and if it is not an effective declassification step.

We prove a noninterference theorem, which captures the requirements on both cross-application and intra-application communications. The theorem only concerns traces generated by valid transitions. Declassification can cause the low actions that follow it to differ between the two systems. However, we do allow arbitrary declassification prior to the projection of the high components. A component that declassified will be treated as a low component, and will afterward be denied any secrets unless further declassification occurs elsewhere. Changing aid_c's application-level label interferes with our attempt to view components in aid_c as independent entities.

Theorem 1 (Noninterference)
For all κ_L, for all applications App$(aid_1), \cdots$, App(aid_n),
given a aid_c ($aid_c = aid_i$), $i = 1 \ldots n$, whose components do not access the shared variable,
let $S = A_M \mid T_M \mid$ App$(aid_1), \cdots$, App(aid_n) be the initial system configuration, $S \Longrightarrow_A S'$,
$S' = A_M \mid T_M \mid \nu \boldsymbol{c}.(T_{MI}(\Xi) \mid AC(aid_c) \mid S'')$, where $T_{MI}(\Xi)$ is an instance of the tag manager, Ξ is the current label map, and $AC(aid_c)$ is an active launched instance of aid_c,
let $\Xi' = \mathsf{projT}(\Xi; \kappa_L; aid_c)$,
$S_L = A_M \mid T_M \mid \nu \boldsymbol{c}'.(T_{MI}(\Xi') \mid \mathsf{proj}(AC(aid_c) \mid S'', \kappa_L, aid_c, \Xi'))$

1. *$\forall t$ s.t. $S' \overset{t}{\Longrightarrow}_A S_1$, and $\overset{t}{\Longrightarrow}_A$ is a sequence of valid transitions, $\exists t'$ s.t. $S_L \overset{t'}{\Longrightarrow}_A S_{L1}$, and $t|^{\sqsubseteq_{aid_c}}_{\kappa_L} = t'|^{\sqsubseteq_{aid_c}}_{\kappa_L}$*
2. *$\forall t$ s.t. $S_L \overset{t}{\Longrightarrow}_A S_{L1}$, and $\overset{t}{\Longrightarrow}_A$ is a sequence of valid transitions, $\exists t'$ s.t. $S' \overset{t'}{\Longrightarrow}_A S_1$, and $t|^{\sqsubseteq_{aid_c}}_{\kappa_L} = t'|^{\sqsubseteq_{aid_c}}_{\kappa_L}$*

6 Case Study and Implementation

We implemented our system on Android 4.0.4, using techniques similar to those used by other works [5,15]. Here we describe in detail our policy for the example scenario from §3.1, and briefly discuss our implementation.

Motivating Scenario Revisited The policy of our example from §3.1 prohibits secret files from being leaked on the Internet, but allows them to be manipulated by applications and emailed at the user's behest. Files may be edited, but can be emailed only if the file manager itself calls the email application. We extend the example to also allow files to be emailed if they are first encrypted.

We first show how to implement this policy by assigning application-level labels. The file manager is labeled with ({FileSecret}, {FileWrite}, {-FileSecret}). The editor is labeled with (F{}, F{}, {}), to indicate that its effective secrecy and integrity labels are inherited from its caller, but it has no ability to declassify or endorse. The email application is labeled with ({ReadContacts, ...}, {}, {+Internet, ...}). The "..." signify

additional secrecy tags and endorsement capabilities that enable the email application to read user accounts, cause the phone to vibrate, etc. To permit callers with low integrity, tags that permit access to resources (e.g., to vibration functionality) appear as endorsement capabilities rather than integrity tags. The encryption application is labeled with (F{}, F{}, {-T, +WriteExternalStorage}). It has floating secrecy and integrity labels and can declassify all secrets it acquires, and so it must be trusted to correctly encrypt files and not reveal files without encrypting them. The encryption application also needs the WriteExternalStorage tag to be able to store encrypted data on the SD card.

This choice of labels achieves our desired functionality as follows: When called by the file manager, the editor's label floats to ({FileSecret}, {FileWrite}, {}). The editor cannot declassify FileSecret and so cannot leak the file; because it has FileWrite, it can save the file to secret storage. To email the file, a user invokes the email application via the file manager, which adds the file content to the intent that starts the email application, and removes FileSecret by declassifying before sending the intent. The file can also be released via the encryption application. If invoked by the file manager, the encryption application floats to ({FileSecret}, {FileWrite}, {-T,+WriteExternalStorage}); its capability to declassify any secret (-T) allows it to release data to any application.

We used component-level policy to restrict the file manager's declassification capability to only the component whose task is to send files to other applications. The duties of the components can be inferred from their names. We label the Main activity and the File provider with ({FileSecret}, {FileWrite}, {}) since they need to handle files; the Help and DirectoryInfo activities with ({FileSecret}, {}, {}); the Settings activity with ({FileSecret}, {FileWrite}) because it needs to return a result to the Main activity; and the Send activity with ({FileSecret}, {FileWrite}, {-FileSecret}).

Implementation Our case study is fully implemented and has been tested on a Nexus S phone. We extended Android's manifest file syntax to support our labels. Run-time enforcement is via extensions to Android's activity manager, which already mediates communication between components. The biggest challenges were in providing more detailed information about callers to the activity manager and capturing low-level actions that it did not mediate; we do this via kernel-level middleware [25]. For backward compatibility, we mapped system-declared permissions to secrecy and integrity tags, and assigned label signatures to Android and Java APIs. Please see our technical report [1] for more detail about the implementation.

As part of booting the phone to the point where it can execute ordinary applications, over 50 built-in applications start running. Our case study used minimally modified off-the-shelf applications: Open Manager 2.1.8, Qute Text Editor 0.1, Android Privacy Guard 1.0.9, Email 2.3.4. Our system's implementation totaled ∼1200 lines of code: ∼650 in the reference monitor, 400 for bookkeeping, 100 for enhancing IPCs, and 50 for syntactic support for labels. We measured overheads on the order of 7.5 ms for the label checks incurred by each call[3]. Performance was sufficiently good for this overhead not to be observable to the user.

[3] Even averaging over hundreds of runs, variance between sets of runs was too great to report more precise measurements.

7 Conclusion

We propose the first DIFC-style enforcement system for Android that allows convenient, high-level specification of policy and has a well-understood theory. To support Android's programming model the system had to incorporate several features that are new to information-flow systems, including multi-level policy specification and enforcement, floating labels, and support for persistent state and single-instance components. Our system strikes a balance between providing strong formal properties (noninterference) and applicability, achieving most of each. A prototype and case study validate the design of our system, and confirm that it can enforce practical policies on a Nexus S phone.

Acknowledgments This research was supported in part by Department of the Navy grant N000141310156 issued by the Office of Naval Research; by NSF grants 0917047 and 1018211; by a gift from KDDI R&D Laboratories Inc.; and by Kuwait University.

References

1. Aljuraidan, J., Fragkaki, E., Bauer, L., Jia, L., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on Android. Technical Report CMU-CyLab-12-015, Carnegie Mellon University (2012)
2. Arden, O., George, M.D., Liu, J., Vikram, K., Askarov, A., Myers, A.C.: Sharing mobile code securely with information flow control. In: Proc. IEEE S&P. (2012)
3. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proc. POPL. (2012)
4. Bohannon, A., Pierce, B.C., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive noninterference. In: Proc. CCS. (2009)
5. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: Proc. NDSS. (2012)
6. Chaudhuri, A.: Language-based security on Android. In: Proc. PLAS. (2009)
7. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proc. MobiSys. (2011)
8. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: Proc. IEEE CSF. (2010)
9. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: Proc. ISC. (2010)
10. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight provenance for smart phone operating systems. In: Proc. USENIX Sec. (2011)
11. Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. OSDI. (2010)
12. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: Proc. USENIX Sec. (2011)
13. Felt, A.P., Wang, H., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: Proc. USENIX Sec. (2011)
14. Focardi, R., Gorrieri, R.: A classification of security properties for process algebras. J. of Comput. Secur. **3** (1994) 5–33

15. Fragkaki, E., Bauer, L., Jia, L.: Modeling and enhancing Android's permission system. In: Proc. ESORICS. (2012)
16. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proc. IEEE CSF. (2012)
17. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In: Proc. CCS. (2011)
18. Krohn, M., Tromer, E.: Noninterference for a practical DIFC-based operating system. In: Proc. IEEE S&P. (2009)
19. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Proc. SOSP. (2007)
20. Loftus, J.: DefCon dings reveal Google product security risks. `http://gizmodo.com/5828478/` (2011) [accessed 10-Jul-2012].
21. Marforio, C., Francillon, A., Čapkun, S.: Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich (April 2011)
22. Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: Proc. IEEE CSF. (2011)
23. Myers, A.C.: Practical mostly-static information flow control. In: Proc. POPL. (1999)
24. Nauman, M., Khan, S., Zhang, X.: Apex: extending Android permission model and enforcement with user-defined runtime constraints. In: Proc. ASIACCS. (2010)
25. NTT Data Corporation: TOMOYO Linux. `http://tomoyo.sourceforge.jp/` (2012) [accessed 10-Apr-2012].
26. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.D.: Semantically rich application-centric security in Android. In: Proc. ACSAC. (2009)
27. Passeri, P.: One year of Android malware (full list). `http://hackmageddon.com/2011/08/11/one-year-of-android-malware-full-list/` (2011) [accessed 10-Jul-2012].
28. Rafnsson, W., Sabelfeld, A.: Limiting information leakage in event-based communication. In: Proc. PLAS. (2011)
29. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proc. IEEE CSF. (2010)
30. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. J. Comput. Secur. **9**(1–2) (2001)
31. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal Sel. Area. Comm. **21**(1) (2003) 5–19
32. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: Proc. NDSS. (2011)
33. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the Android framework. In: Proc. SocialCom/PASSAT. (2010)
34. Shin, W., Kwak, S., Kiyomoto, S., Fukushima, K., Tanaka, T.: A small but non-negligible flaw in the Android permission scheme. In: Proc. POLICY. (2010)
35. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F.: Improving application security with data flow assertions. In: Proc. SOSP. (2009)
36. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: Proc. OSDI. (2006)
37. Zeldovich, N., Boyd-Wickizer, S., Mazières, D.: Securing distributed systems with information flow control. In: Proc. NSDI. (2008)
38. Zhang, D., Askarov, A., Myers, A.C.: Language-based control and mitigation of timing channels. In: Proc. PLDI. (2012)