Run-Time Enforcement of Information-Flow Properties on Android

Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Limin Jia, Kazuhide Fukushima,

Shinsaku Kiyomoto, and Yutaka Miyake

July 23, 2012

(Revised December 7, 2012)

CMU-CyLab-12-015

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Run-Time Enforcement of Information-Flow Properties on Android

Jassim Aljuraidan Elli Fragkaki
Lujo Bauer Limin Jia
Carnegie Mellon University
{aljuraidan,elli,lbauer,liminjia}@cmu.edu

Kazuhide Fukushima Shinsaku Kiyomoto
Yutaka Miyake
KDDI R&D Laboratories, Inc.
{ka-fukushima,kiyomoto,miyake}@kddilabs.jp

Abstract

Recent years have seen a dramatic increase in the number and importance in daily life 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 develop a detailed model of our enforcement system using a process calculus, and use the model to prove noninterference. Our system and model have a number of useful or novel features, including support for Android's single- and multiple-instance components, floating labels, declassification and endorsement capabilities, and support for legacy applications. Our design fits the Android programming model cleanly enough that we have developed a fully functional prototype on Android 4.0.4. We tested our prototype 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.

Keywords information flow; Android; run-time enforcement; language-based security

1. Introduction

Recent years have seen a dramatic increase in the number and importance in daily life of smartphones and similar mobile devices. The security properties that mobile devices and their operating systems provide to their applications, however, are inadequate to protect against many undesired behaviors, contributing to the rapid rise in the amount of malware targeting mobile devices [21, 27].

To mitigate application misbehavior, mobile OSes like Android rely largely on strong isolation between applications, as well as permission systems that limit communication between applications and access to resources and sensitive APIs. Researchers have investigated these mechanisms at length, finding them vulnerable to application collusion through the use of covert channels [22, 31], information-flow leaks [11, 31], and privilege-escalation attacks [7, 13]. Attempts to address these issues have produced tools for detecting information leaks [6, 10, 18], improvements to permission systems (e.g., [24, 26]), as well as more powerful mechanisms for restricting applications' access to data and resources (e.g., [4]).

Many commonly discussed 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 identity (and associated, 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. Similarly, when deciding whether to allow a caller to access an API, the system can base its decision only on the permissions that the caller holds; it cannot take into account the caller's prior behavior.

Recent work on understanding or preventing undesired information flows on Android typically focuses on using a specific mechanism to enforce a pre-determined global policy [6, 10]. Other works have developed more powerful mechanisms that track control flow to enable more informed enforcement and allow finer-grained control over communication and resource accesses [4, 8]; these also typically lack convenient policy languages. Although a few formal analyses of Android's security architecture have provided some insight about its limitations [32], works that introduce more powerful mechanisms typically do not formally investigate the properties that those mechanisms exhibit.

Our work is the first to propose 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 [20, 34], our system permits programmers to specify policy via programmer- or system-defined labels applied to applications or application components. Specifying and enforcing policy at the level of application components is a practically interesting middle ground between process-level (e.g., [20]) and instruction-level (e.g., [23]) enforcement of information-flow policies, 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 (such as 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 for the enforcement system. 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 application components, which are not. This 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.

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 (Section 3).
2. We develop a faithful process-calculus model of Android's main programming abstractions and our system's enforcement mechanism (Section 4).
3. We define noninterference for our enforcement system and prove that it holds (Section 5).
4. We implement our system on top of 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, and that this enforcement is compatible with the Android runtime (Section 6).

2. Background and Related Work

In this section we give a high-level overview of Android (Section 2.1) and review related work (Section 2.2).

2.1 Android Overview

Android is a Linux-based, open-source OS. Android 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* store data and provide an SQL-like interface for sharing data 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; hence, 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 neither Java's abstractions nor the Android abstractions built on top of them prevent unmediated communication between components, the Android programming model strongly 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.

2.2 Related Work

Information Flow Enforcing information-flow policies has been an active area of research. Some works use language-based techniques and develop novel information-flow type systems (cf. [30]) that provably enforce noninterference properties; others use runtime monitoring techniques (e.g., [2, 17]). Our approach is most similar to work on enforcing information-flow policies in operating systems [20, 33, 35]. There, each process is associated with a label specifying its information-flow policies. The components in our system can be viewed as processes in an operating system. However, most of these systems do not prove any formal properties of their enforcement mechanisms. Krohn et al. [19] presented one of the first proofs of noninterference for practical DIFC-based operating systems. Our design is inspired by Flume [20], but has many differences. For instance in Flume, floating labels are not allowed. 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 for our system: their definition of noninterference is based on a stable failure model, which is a simulation-based definition. Our definition is trace-based, and does not capture information leaks due to a high process stalling.

Recent work on run-time enforcement of information-flow policies on mobile code [1, 2, 17] tracks information flow at a much finer level of granularity than ours.

There has been a rich body of work on noninterference in process calculi [14, 29]. Recently, researchers have re-examined definitions of noninterference for reactive systems [3, 28]. In these systems, each component waits in a loop: once inputs are available, the component processes an input and produces 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 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.

Android Security Many works have recently focused on Android security, analyzing Android's permission system [6, 9], developing tools to detect misbehaving applications [12, 16], and proposing new protection mechanisms (e.g., [24, 26]). Android's permission system has been shown to be inadequate to protect against many common attacks, including privilege-escalation attacks [7, 13] and information leaks [5, 10, 11, 31].

Closest to the goal of our work are projects such as TaintDroid [10] and AppFence [18], which aim to automatically detect and prevent dangerous information leaks. They operate at a much finer granularity than our mechanism, tracking tainting at the level of variables, and enforce fixed policies. Several works have proposed more general mechanisms. Dietz et al. developed a system in which applications can attest (via digital signatures) to their calling context, and these attestations can be used later when reasoning about whether a call should be allowed [8]. In a similar vein, Bugiel et al. developed a system that monitors interactions between applications at run time and uses this information to make access-control decisions [4]. Both of these works develop powerful enforcement mechanisms, from which we borrow when implementing our own. Our focus, additionally, is on supporting flexible, application- and component-centric policies, and on formally verifying the properties of our enforcement mechanism.

Formal analyses of Android-related security issues and language-based approaches to solving them have received less attention. Shin et al. [32] developed a formal model to verify functional correctness properties of Android, which revealed a flaw in the permission naming scheme and a possible attack. Closest to our work is Sorbet, 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 Sorbet it is not possible to specify that information

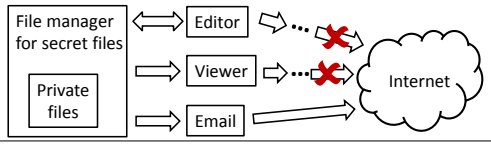

Figure 1. Motivating example: a simple scenario that cannot be implemented using Android permissions.

should not leak to a component unless that component is protected by some permission); and we formally model our enforcement system in much greater detail, thus providing much stronger guarantees about the correctness of our system.

3. Approach Overview

We next describe a scenario that exemplifies the inability of Android's permission system to specify and enforce many simple, practical policies (Section 3.1). We then discuss the key design choices of our system (Section 3.2), how it enforces policies (Section 3.3), and design limitations (Section 3.4).

3.1 Motivating Scenario

Suppose a system has the following applications: a *secret-file manager* for managing files such as a diary and lists of bank-account numbers; a 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 the files that are managed by the secret-file manager from being inadvertently or maliciously transmitted over the Internet; sending files over the Internet should be allowed only if the user explicitly requests it through the file manager. The files need to be viewed and edited. Building a special editor and viewer just for secret files would be impractical, and so these tasks have to be handled by a general-purpose editor and viewer. This scenario is shown in Figure 1.

The desired (information-flow) policy, and interactions between applications shown in this example, are representative of practical scenarios that Android currently cannot satisfactorily handle.

In Android, one might attempt to implement this policy as follows. The file manager could require that its callers have the ReadFile permission, which would prevent applications from reading the files directly from the content provider that stores them. When a user wished to edit or view a file, the file manager would dynamically grant a URI permission to permit access to that file. To prevent the editor and viewer from exfiltrating data, the user might choose to install only viewers and editors that do not have the Internet permission, which Android applications need in order to create TCP connections. The email application would need the Internet permission to send email; it, too, would be granted a URI permission by the file manager when the user wanted to send a file by email (the only legitimate reason, in our scenario, for sending a file over the Internet).

This attempt at implementing the desired policy using Android permissions fails. Once the editor and viewer have gained access to the secret data, they cannot send it over the Internet directly (as they lack the Internet permission), but they can exfiltrate it by sending it to the email application or to any other application that has the Internet permission. To compound the problem, both the viewer and the editor may have legitimate reasons to access the Internet when they are being used on *non*-secret data, and preventing them from doing this may be an unreasonable restriction.

A further concern is implementing least privilege within applications. If an application has many privileges (e.g., it is by policy allowed to access various sensitive resources), one would like to segment the code and the policy so that the privileges are granted only to the subset of the application's code that requires them. This concern is practically relevant in Android: applications often must be granted tens of permissions in order to function correctly; each permission is needed in only one or a small number of instances, but all components of the application receive all permissions.

We next describe some of our system's key features and how they help enforce our example policy. We revisit the example more concretely in Section 6.

3.2 Key Design Choices

We next discuss the design of our system, including the level of abstraction at which we enforce policies and specifying policies via information-flow labels.

3.2.1 Enforcement Granularity

Traditionally, languages and systems that supported information-flow properties did so either at instruction level (e.g., [17, 23]) or at process level (e.g., [20]). Android's division of applications into components invites the exploration of an interesting middle ground—each Android component could be regarded as a process, and policy could be specified and enforced at the level of components. Android applications are typically divided into a relatively small number of key components, e.g., an off-the-shelf file-manager application 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. Android strongly encourages the use of components as modules that communicate only through narrow, well-defined interfaces. In fact, the Android runtime may garbage collect any component that is not directly involved in interacting with the user; using the narrow interfaces 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 going through the Android-mediated interfaces for cross-component communication. Components are composed of normal Java classes, which can communicate independently of component boundaries by directly reading and writing public static fields. In other words, 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 low-level enforcement is possible, e.g., with instruction-level information-flow tracking [23], implementation and integration with existing systems is difficult and can cause substantial run-time overhead.

In this paper we pursue a hybrid approach. We allow policy specification at both component level and application level. Application-level policies are feasible to enforce strictly because Android provides strong isolation between applications; the channels between them are few enough for an enforcement system to be able to monitor them. Enforcement of component-level policies, however, is best-effort: When programmers adhere to Android's programming conventions, potential policy violations that are the result of application compromise or common programmer errors (i.e., errors that do not break the component abstraction) will be prevented by the enforcement system. On the other hand, if an application does not adhere to Android's programming conventions for implementing interactions between components, these components will be able to circumvent their own policies (but not their own application's or any other application's policies).

The purpose of component-level policies is to give programmers a tool to help them better police their own code and implement least privilege. In addition to helping regulate interactions between components within an application, component-level policy also acts in concert with application-level policy to regulate in more detail which cross-application interactions should be allowed. When two components belonging to different applications try to communicate, this will be allowed only if it is consistent with both application-level and component-level policy.

3.2.2 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 uses of 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 The choice to implement secrecy and integrity labels as sets of tags was motivated by the desire to help with backward compatibility with standard Android permissions. In Android, the set of permissions is not pre-defined; any application can declare new permissions at installation time. In our system, each application can similarly 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. To support legacy applications, the permissions that those applications possess or require of their callers can be mapped to tags, and the policy expressed through permissions can be mapped to a label. We discuss this further in Section 6.

Declassification and Endorsement The declassification capabilities, δ, that are part of a component's or application's label specify which tags the component or application may remove from s and which tags it may add to i. We specify the declassification capabilities as part of a component's (or application's) label because whether or not a component should be allowed to declassify or endorse is a part of the security policy; to make it easier to reason about policy, we want to express it clearly and in a declarative way. Furthermore, this way of specifying declassification policy also aids in 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 to make a call succeed.

Returning to the example from Section 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 file's 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 by removing FileSecret from its secrecy label. When it starts executing, the file manager's effective label will be the same as the static label we just described. If the file manager then 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 want to 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 through the use of declassification or endorsement.

Floating Labels Some components or applications, e.g., an editor or a library component, may have no secrets of their own that they need to protect 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.

In our example, the editor application's static policy is (F{}, F{}, {}). If instantiated by the file manager, the editor's effective secrecy label would become {FileSecret}, allowing the editor and the file manager to share data, but preventing the editor from calling any applications or APIs that have a weaker secrecy label than {FileSecret}. If the editor also had its own secrets to protect, we might give it the static label (F{EditorSecret}, F{}, {}). Then, the editor's effective label could be floated at instantiation to, e.g., ({EditorSecret, FinancialSecret}, {}, {}), but any instantiation of the editor would carry an effective secrecy label at least as restrictive as {EditorSecret}.

Returning to our example: 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

The crux of our enforcement system is a reference monitor that intercepts calls between components and permits or denies a call based on the caller's and callee's labels. Much of the reference monitor's responsibility is maintaining the mapping from applications and components (and their instances) to their effective labels. As we will discuss in Section 6, we build our reference monitor on top of Android's activity manager. In our formal model (Section 4) we abstract 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.

3.3.1 Application- and Component-level Enforcement

Whenever two components attempt to communicate via an intent, our reference monitor permits or denies the call by comparing the labels of the caller and the callee. In the simple case, when the caller and callee are part of the same application this comparison is straightforward: the call is allowed 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; otherwise, the call is denied.

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. This requires four comparisons

between pairs of labels. If any comparison indicates that the call should be denied, it is denied; otherwise the call is allowed.

If both the callee component and the callee's application have a floating label, and the callee's application is not running prior to a call, then the callee's and the callee's application's effective labels will be constructed by adding to their static labels the supersets of the tags of the caller and the caller's application. In other words, the callee's effective integrity label will be the union of its static integrity label, the caller's effective integrity label, and the caller's application's effective integrity label. The callee's effective secrecy label and the callee's application's effective labels will be constructed similarly.

Declassification and endorsement may change the effective labels of components and applications, and are permitted only when consistent with policy. For programmer convenience, we allow a caller component to declassify and endorse automatically when this is necessary for an outgoing call to be permitted. We discuss this further in Section 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 through calls to Android and Java APIs, which provide functions such as accessing the file system or opening sockets. These APIs are not themselves components or applications; however, we assign them labels similarly as we would to components.

3.3.2 Persistent State

Many components are multi-instance: each attempt to communicate with such a component causes a new instance to be created. Such components intuitively pose little difficulty for enforcing information-flow properties, since each attempt to communicate with one generates a fresh instance, 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, which seek to communicate via a colluding single-instance component C. Suppose that A's static secrecy label is $\{\mathsf{FileSecret}\}$ and B's is $\{\}$, making direct communication from A to B impossible; 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; this is permitted according to their labels. If the content of the secret file is "Attack at dawn," C exits; otherwise, C continues running. B calls C, then calls C again; and if B receives two calls from C, then it learns that A's secret file is "Attack at dawn." C can only send the second call to B after it exits, which only happens when A's secret file is "Attack at dawn."

The information leak in this scenario arises because C has changed (declassified!) its label by exiting. To prevent scenarios like this (and to allow us to prove noninterference, which also ensures that no similar scenarios remain undiscovered), raising a label must change not only a component's effective label, but also its static label.

3.4 Design Limitations

We explicitly avoid addressing several issues that impact the security our enforcement system provides in practice.

We do not attempt to address communication via covert channels, such as timing channels. Recent work has identified ways in which these may be mitigated via language-based techniques [36]; but such techniques are outside the scope of this paper.

A second major area that our work does not address is the robustness of Android's abstractions. In particular, stronger component-level abstractions might 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 further bolster the efficacy of our approach.

Many security architectures are vulnerable to user error. Our system does not address this problem; 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., agreeing to trust an untrustworthy application).

4. Process Calculus Model

We next show how to encode Android applications and our enforcement mechanism in a process calculus. Our 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.

4.1 Labels and Label Operations

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

$$\begin{array}{lllll} \textit{Simple labels} & \kappa & ::= & (\sigma, \iota) \\ \textit{Label quantifiers} & Q & ::= & C \mid F \\ \textit{Process labels} & K & ::= & (Q(\sigma), Q(\iota), \delta) \end{array}$$

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, as described in Section 3.2. A process label 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 $t_s \in \mathcal{S}$, 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; similarly, a component that has $+t_i$ can add the tag t_i to its integrity tags ι.

We define operations on labels to allow the reference monitor to compare labels, compute the effects of declassification, and instantiate floating 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. Erasure 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 Section 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. Dually, the

AM Erasure $C(\sigma)^- = \sigma \quad F(\sigma)^- = \top$
$(Q(\sigma), Q(\iota), \delta)^- = ((Q(\sigma))^-, \iota)$

PF Erasure $C(\iota)^* = \iota \quad F(\iota)^* = \top$
$(Q(\sigma), Q(\iota), \delta)^* = (\sigma, (Q(\iota))^*)$

Raise $(Q(\sigma), Q(\iota), \delta) \uplus_{rz} (\sigma', \iota') = (Q(\sigma \cup \sigma'), Q(\iota \cap \iota'), \delta)$

Declassify $(C(\sigma), C(\iota), \delta) \uplus_d \delta_1 =$
$(C(\sigma \backslash \{t | (-t) \in \delta_1\}), C(\iota \cup \{t | (+t) \in \delta_1\}), \delta)$

Merge $(C(\sigma), C(\iota)) \uplus_M (C(\sigma'), C(\iota')) = (C(\sigma \cup \sigma'), C(\iota \cap \iota'))$

Instantiate
$C(\sigma_1) \lhd_S C(\sigma_2) = C(\sigma_1) \quad F(\sigma_1) \lhd_S C(\sigma_2) = C(\sigma_1 \cup \sigma_2)$
$C(\iota_1) \lhd_I C(\iota_2) = C(\iota_1) \quad F(\iota_1) \lhd_I C(\iota_2) = C(\iota_1 \cup \iota_2)$
$(s_1, i_1, \delta) \lhd (C(\sigma_2), C(\iota_2)) = (s_1 \lhd_S C(\sigma_2), i_1 \lhd_I C(\iota_2), \delta)$

Figure 2. Summary of label operations.

raise operation $K \uplus_{rz} (\sigma, \iota)$ adds to K the secrecy tags in σ and removes from K the integrity tags in ι.

When a component that has a floating label is called, its label needs to be instantiated based on the caller's label. We write $K \lhd \kappa$ to denote the instantiation of a potentially floating label K based on the caller's simple label κ. The resulting label inherits all the tags from both of the labels.

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.

Process Calculus The syntax of our modeling calculus, defined below, is based on π-calculus. To avoid confusing the parallel composition in process calculus with the BNF definitions, we use $'|'$ instead of $|$ for parallel composition.

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

aid denotes an application identifier, and cid a component identifier, both drawn from a universe of identifiers. c denotes constant channel names. A composed constant name is a constant name c prefixed by an application ID ($aid{\cdot}c$) or by application and component IDs ($aid{\cdot}cid{\cdot}c$). These names model specific interfaces provided by an application or a component; we will show examples later in this section.

Expressions e include variables, names, and data constructors. We extend the standard definition of a process P with if statements, pattern-matching statements, and a pattern-matched input $\mathsf{in}\ x(\mathsf{patt})$. This input only accepts outputs that match with patt. A pattern can be a variable x, where the x will be bound in the process after the input; a wildcard $\_$ that matches everything; and a constance c. An equality check pattern is written $\_ = x$, where x is not considered bound in the process after the input; instead, x is bound earlier and by the time the pattern-matched is evaluated, x is substituted with a ground term. A pattern can also be a data constructor, or a tuple of patterns. These extensions can be encoded directly in π-calculus, but we add them as primitive constructors to simplify the representation of our model.

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.

We define standard structural congruence and labeled transition rules for the calculus in Appendix A.

Common Encoding Structures We summarize the basic constructs that model commonly used programming idioms.

To establish communication between processes P and Q, P can generate a new channel r, send it only to Q, and then wait for messages on r. When a service process has many active instances, this ensures that a call to one instance will not be confused with calls to other instances.

We use the choice operator together with the pattern-matched input to encode a process that can handle several different requests. For example, the following process evaluates to P_1 if a request rd is received, and to P_2 if a request wt is received: $\mathsf{in}\ c(\mathsf{rd}, x_1).P_1 + \mathsf{in}\ c(\mathsf{wt}, x_2).P_2$.

We encode a recursive process using the ! operator. Often, a recursive process P is of the form $!(\mathsf{in}\ c(x).P')$, where P' contains $\mathsf{out}\ c(y)$. The ! operator is also used to encode the fragment of an application or a component from which run-time instances are generated. 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.

4.3 A Model of Android and Our Enforcement Architecture

We model as processes the three main kinds of 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 labels of the caller and the callee. 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 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. For a multi-instance component, a new instance is created on every call to that component.

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.

A single-instance component is initially *unlaunched*. When an intent is sent to an unlaunched component, the reference monitor creates a new instance of the component, which becomes *launched*. Once the instance exits, the component goes back to the unlaunched state. The reference monitor does not explicitly track whether multi-instance components have been launched.

Similarly to a single-instance component, an application is initially *unlaunched*. The first call to one of its components changes the application's state to *launched*, and creates a new application instance. After one of its components executes the exit application instruction, the application enters the *exiting* state. In this state, the reference monitor will not initiate new calls to the application. Once all of its components' instances exit, the application instance

Component body $A(cid, aid, I, c_{AI}, c_{sv}, c_{ls}, c_{nI}, c_{lock}, rt, V) ::=$

$\cdots$

$\mid \mathsf{out}\, t_m(\mathsf{raiseA}, aid, c_{AI}, \delta, \iota).A(\cdots)$

$\mid \mathsf{out}\, t_m(\mathsf{raiseC}, cid, c_{nI}, \delta, \iota).A(\cdots)$

$\mid \mathsf{out}\, t_m(\mathsf{dclassifyA}, c_{AI}, \delta).A(\cdots)$

$\mid \mathsf{out}\, t_m(\mathsf{dclassifyC}, c_{nI}, \delta).A(\cdots)$

$\mid \mathsf{out}\, a_m(\mathsf{call}_I, rt, aid, c_{AI}, cid_{ce}, I)\ '|'\ A(\cdots)$

$\mid \mathsf{out}\, a_m(\mathsf{call}_E, rt, c_{AI}, c_{nI}, aid_{ce}, cid_{ce}, I)\ '|'\ A(\cdots)$

$\mid \mathsf{out}\, a_m(\mathsf{exitA}, aid, cid, c_{AI}, c_{nI}, rt, e)$

$\mid \mathsf{out}\, a_m(\mathsf{exitC}, aid, cid, c_{AI}, c_{nI}, rt, e)$

$\mid \mathsf{out}\, c_{ls}(e).\mathsf{out}\, c_{lock}() \mid \mathsf{out}\, c_{sv}(\mathsf{wt}, e).A(\cdots)$

$\mid \nu r.\mathsf{out}\, c_{sv}(\mathsf{rd}, r).\mathsf{in}\, r(x).A(\cdots, V \cup \{x\})$

Figure 4. Partial encoding of component bodies.

exits, and the application returns to the unlaunched state. We assume a cooperative environment where an application exits only after each of its components exits on its own. We do not model the run-time monitor force quitting an application, as this makes it hard to correctly implement cleaning up dangling state, and does not critically affect noninterference.

Encoding Applications and Components The encoding of applications and components is shown in Figure 3. We summarize special-purpose channels in Figure 5. We delay explaining the label contexts $\ell[...]$ that surround processes until Section 5—they are annotations that facilitate proofs, and have no run-time meaning.

The encodings of single- and multi-instance components are the same except for a label that distinguishes between them; the differences in run-time behavior are due to how they are handled by the reference monitor.

The event loop body of a component $CB(arg, s)$ waits on its *new intent channel* c_{nI} before executing its program ($A(...)$, defined later). The output $\langle \mathsf{out}\, I(\mathsf{self}) \rangle$ is purely for the proof of noninterference; we will revisit this in Section 5. The parameters *arg*, together with s, appear free in A. They are generated by outer-layer processes, which we explain next. The event loop $CE(arg)$ is triggered by an input to the *local state channel* c_{ls}, which is sent by this component instance when the previous intent has been processed. This ensures that each iteration of a component instance shares the local state of the previous iterations.

A component $CP(aid, cid, c_{AI}, c_{sv})$ is launched from a designated creation channel $aid{\cdot}cid{\cdot}c_{cT}$. The message it receives on the creation channel 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. 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.

Once a component instance is created, it generates a new name for the local state channel c_{ls}. The process running in parallel with the component event loop launches the event loop by passing an initial local state σ_0 to the c_{ls} channel, and then sending the intent I to c_{nI}.

The body of the shared state *SVBody* processes read and write requests through the channel c_{sv}, and is launched through channel c_{svL}.

An application *App*(aid) with ID aid is composed of components $CP_i(aid, cid_i, c_{AI}, c_{sv})$ running in parallel, and a shared state *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 launches the shared state with an initial value. At this point, the application's components are ready to receive calls, and we call this application instance an *active launched* instance.

Component Body We define the body of a component in terms of the operations that a component can perform. It is parameterized over several variables, most of which were introduced previously. The only new variable is the last one, V, which is a set of variables that are free in the body; these are bound by outer-layer case statements or ν. In Figure 4, we omit the variables when they are clear from the context.

A component can use if statements and case statements. A component can change its label or its application's label only by sending explicit requests to the label manager.

A component can call another component in the same application by sending a request $\mathsf{out}\, a_m(\mathsf{call}_I, rt, \ldots)$ to the activity manager. Here a_m is the designated channel to send requests to the activity manager. The argument *rt* is NONE if no return is expected; otherwise the caller includes in *rt* its application's ID and application's instance ID, new intent channel, and the lock channel ($\mathsf{SOME}(aid, c_{AI}, c_{nI}, c_{lock})$).

To exit, an instance sends a request to the activity manager ($\mathsf{out}\, a_m(\mathsf{exitC}, \cdots)$), which contains information for cleaning up its state and any return information (*rt*). An instance can also terminate the application upon exiting ($\mathsf{out}\, a_m(\mathsf{exitA}, \cdots)$). After one of these two messages is sent, the instance is left as dead code guarded by the input to channel c_{ls}. No one can send new intents to this instance, since the channel c_{nI} is not available to receive input.

A component can also start another iteration of its body: ($\mathsf{out}\, c_{ls}(k, e).\mathsf{out}\, c_{lock}(k)$). The output to the channel c_{lock} will inform the activity manager that the component is ready to receive another intent. The activity manager always waits for an output on this channel before sending a new intent to the channel c_{nI}. Therefore, c_{lock} acts like a lock.

Finally, a component can read or write to the shared variable in its application.

Label Manager The label manager maintains the mappings from applications, components, and run-time instances to labels, and processes calls to update the mapping. The label manager's local state is a label map Ξ defined as follows:

$$\begin{array}{rcl} \Xi & ::= & \cdot \mid \Xi, aid \mapsto \mathsf{U}(K) \mid \Xi, aid \mapsto \mathsf{L}(c_{AI}, K) \\ & \mid & \Xi, aid \mapsto \mathsf{onEx}(c_{AI}, K) \mid \Xi, aid{\cdot}cid \mapsto \mathsf{S}(K) \\ & \mid & \Xi, aid{\cdot}cid \mapsto \mathsf{SL}(c_{nI}, K) \mid \Xi, aid{\cdot}cid \mapsto \mathsf{M}(K) \\ & \mid & \Xi, c_{AI} \mapsto \mathsf{DA}(K, \mathsf{LC}(c_{nI_1}, \ldots c_{nI_n})) \mid \Xi, c_{nI} \mapsto \mathsf{DC}(c_{lock}, K) \end{array}$$

If an application has not been launched, then its aid maps to $\mathsf{U}(K)$, where K is the application's static label. Once an application is launched, its aid maps to $\mathsf{L}(c_{AI}, K)$, where c_{AI} is its instance ID, and K the static label. After one of its components sends a call to terminate the application, but before the application exits, aid maps to $\mathsf{onEx}(c_{AI}, K)$.

Unlaunched single-instance components have their ID ($aid{\cdot}cid$) mapped to a $\mathsf{S}(K)$, where K is their static label. Ξ maps the ID of running single-instance components to $\mathsf{SL}(c_{nI}, K)$, where c_{nI} is the new-intent channel for this instance, and K the static label. For multi-instance components, Ξ maps the component ID to $\mathsf{M}(K)$.

The last kind of mapping in Ξ records the labels of each run-time instance (of applications and single- and multi-instance components). An application instance channel c_{AI} maps to a pair of its current label and a list of the active new-intent channels of its components. A component new intent channel c_{nI} maps to a pair (c_{lock}, K), where c_{lock} is the lock channel for the instance and K is its effective label.

We the detailed definition of the label manager can be found in Appendix C. When a component instance requests to declassify, the label manager checks that the calling instance has the capabilities to do so, and if appropriate updates the current label for that

$$
\begin{array}{llcl}
\textit{Parameters} & \textit{arg} & = & aid, cid, I, c_{AI}, c_{sv}, c_{ls}, c_{nI}, c_{lock}, rt \\
\textit{Comp. loop body} & CB(arg, s) & = & \mathsf{in}\, c_{nI}(I).\langle \mathsf{out}\, I(\mathsf{self})\rangle.A(arg, \{s\}) \\
\textit{Comp. event loop} & CE(arg) & = & !(\mathsf{in}\, c_{ls}(s).CB(arg, \{s\})) \\
\textit{Component} & CP(aid, cid, c_{AI}, c_{sv}) & = & !(\mathsf{in}\, aid{\cdot}cid{\cdot}c_{cT}(\_{=}c_{AI}, I, c_{nI}, c_{lock}, rt). \\
 & & & \quad (c_{AI}, c_{nI})[\nu c_{ls}.(\mathsf{out}\, c_{ls}(\sigma_0).\mathsf{out}\, c_{nI}(I)\ '|'\ CE(arg))] \\
\textit{Shared store body} & \textit{SVBody}(c_{svL}, c_{sv}) & = & \mathsf{in}\, c_{sv}(\mathsf{rd}, r).\mathsf{out}\, r(x).\mathsf{out}\, c_{svL}(x)\ +\ \mathsf{in}\, c_{sv}(\mathsf{wt}, v).\mathsf{out}\, c_{svL}(v) \\
\textit{Shared store} & \textit{SV}(c_{svL}, c_{sv}) & = & !(\mathsf{in}\, c_{svL}(x).\textit{SVBody}(c_{svL}, c_{sv})) \\
\textit{Application body} & \textit{AppBody}(aid, c_{AI}) & = & \nu c_{svL}.\nu c_{sv}.\mathsf{out}\, c_{svL}(s_0).(\textit{SV}(c_{svL}, c_{sv})\ '|'\ \\
 & & & \quad (c_{AI}, cid_1)[CP_1(aid, cid_1, c_{AI}, c_{sv})]\ '|'\ \cdots\ '|'\ (c_{AI}, cid_n)[CP_n(aid, cid_n, c_{AI}, c_{sv})]) \\
\textit{Application} & \textit{App}(aid) & = & aid[!(\mathsf{in}\, aid{\cdot}c_L(c_{AI}).c_{AI}[\textit{AppBody}(aid, c_{AI})])]
\end{array}
$$

Figure 3. Encoding of applications and components.

	ID	launch channel	instance ID	instance-specific channels
Application	aid	$aid{\cdot}c_L$	c_{AI}	c_{svL}: shared variable launch channel c_{sv}: shared variable read-write channel
Component	cid	$aid{\cdot}cid{\cdot}c_{cT}$	c_{nI}	c_{lock}: lock channel for processing new intents (returns) c_{ls}: launch channel of the instance's local state

Figure 5. Summary of identifiers for applications and components. c_{AI}, c_{nI}, and c_{lock} are generated by the activity manager; c_{svL}, c_{sv}, and c_{ls} are generated by the application or component.

instance. As discussed earlier, raising is allowed only if neither the application label nor the component label is floating, and raising the label of a single-instance component (or application) will cause the label of that component (or application) to stay raised even after the instance exits.

The label manager also answers queries from the activity manager for labels. Such queries cannot be made by components, since they would leak information. The label manager also processes requests from the activity manager to update its mapping when applications or components launch and exit. When a component exits, the label manager removes the current instance's mapping, and restores a single-instance component's mapping to its static label. When the last component of an application exits, the label manager cleans up the application's state.

Activity Manager Android's activity manager mediates all intent-based communication between components, preventing any communication that is prohibited by policy. We describe here the enhanced activity manager that we use for enforcement in our system.

The top-level process of the activity manager is of the form: $\mathsf{A_M} = !(AM_I\ +\ AM_E\ +\ AM_{EX}\ +\ AM_R)$. 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 focus on explaining the fragment of the encoding of the activity manager that processes calls between components within the same application (Figure 6).

When the activity manager receives a request to send intent I to a component with ID cid_{ce}, it first looks up the callee's label by sending a request to the label manager. The label manager's reply will indicate one of four cases: the callee (1) does not exist; (2) is a single-instance component without an active instance; (3) is a single-instance component with one instance; or (4) is a multi-instance component. If the callee does not exist, the activity manager exits. When the callee is a single-instance component with no active instance, the activity manager allows the call if the caller's label is lower than or equal to the callee's label. To do this, the activity manager (1) generates a new intent channel and a lock channel for the new instance; (2) updates the label for that component to indicate one instance launched, inserts the mapping for the new-intent channel, and records this new active instance in the label map and (3) sends a message containing the intent to the callee's

$$
\begin{array}{ll}
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}\ \mathsf{NE}\ \Rightarrow \mathbf{0} \\
3 & \mid \mathsf{S}(k_{ce}) \Rightarrow \\
 & \quad \mathsf{if}\ {kC_{cr}}^- \sqsubseteq {k_{ce}}^- \\
4 & \quad \mathsf{then}\ \nu c_{nI}.\nu c_{lock}. \\
5 & \qquad \mathsf{out}\, t_m(\mathsf{upd}, \{\ (aid{\cdot}cid_{ce}, \mathsf{SL}(c_{nI}, k_{ce})), \\
6 & \qquad\qquad (c_{nI}, (c_{lock}, kC_{cr} \lhd k_{ce}))\}, (c_{AI}, \{c_{nI}\})\}). \\
7 & \qquad (aid, (c_{AI}, c_{nI}))[\mathsf{out}\, aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, \mathsf{NONE})] \\
8 & \quad \mathsf{else}\ \mathbf{0} \\
9 & \mid \mathsf{SL}(c_{nI}, k_s) \Rightarrow \nu c.\ \mathsf{out}\, t_m(\mathsf{lookUp}, c_{nI}, c).\mathsf{in}\, c(\mathsf{DC}(c_{lock}, \_)). \\
10 & \quad \mathsf{in}\, c_{lock}().\ \nu c.\ \mathsf{out}\, t_m(\mathsf{lookUp}, c_{nI}, c).\mathsf{in}\, c(\mathsf{DC}(\_, k_d)). \\
11 & \quad \mathsf{if}\ {kC_{cr}}^- \sqsubseteq {k_d}^- \\
12 & \quad \mathsf{then\ if}\ {kC_{cr}}^- = {k_d}^- \vee concrete(k_s) \\
13 & \qquad \mathsf{then}\ (aid, (c_{AI}, c_{nI}))[\mathsf{out}\, c_{nI}(I)] \\
14 & \qquad \mathsf{else}\ (aid, (c_{AI}, c_{nI}))[\mathsf{out}\, c_{lock}()] \\
15 & \qquad\quad \mid (aid, (kA_{cr}, kC_{cr}))[\mathsf{out}\, a_m(\mathsf{call}_I, kA_{cr}, kC_{cr}, rt, \\
16 & \qquad\qquad\qquad aid, c_{AI}, cid_{ce}, I)] \\
17 & \quad \mathsf{else}\ (aid, (c_{AI}, c_{nI}))[\mathsf{out}\, c_{lock}()] \\
18 & \mid \mathsf{M}(k_{ce}) \Rightarrow \\
19 & \quad \mathsf{if}\ {k_{cr}}^- \sqsubseteq {k_{ce}}^- \\
20 & \quad \mathsf{then}\ \nu c_{nI}.\nu c_{lock}. \\
21 & \qquad \mathsf{out}\, t_m(\mathsf{upd}, \{\ (c_{nI}, (c_{lock}, kC_{cr} \lhd k_{ce}))\}, (c_{AI}, \{c_{nI}\})\}). \\
22 & \qquad (aid, (c_{AI}, c_{nI}))[\mathsf{out}\, aid{\cdot}aid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, rt)] \\
23 & \quad \mathsf{else}\ \mathbf{0}
\end{array}
$$

Figure 6. Partial encoding of the activity manager.

creation channel. Single-instance components never return, so the last argument of the message is always NONE.

If the callee is a single-instance component with an active instance, the activity manager waits to receive a message on the lock channel. At this point, it knows that no other process can have the lock to communicate with the callee. The activity manager then allows the call if the simple label of the caller is lower than or equal to that of the callee.

Because calls to single-instance components (e.g., an editor) often result in replies to the caller (e.g., to return edited text), legacy applications may function poorly if the callee's responses cannot be returned. Hence, if the activity manager detects that a callee has an effective label that would not allow a reply to the caller, but a static label that would, the activity manager delays the original call. After

the callee exits, the call has a chance to complete (although it may have to compete with other calls to the same callee). If the call is denied, the lock on the callee is released so that other processes can send new intents to the callee.

Calls to multi-instance callees are treated similarly to calls to unlaunched single-instance callees. If the call is permitted, a new callee instance is launched. Returns are treated similarly to calls to active instances of single-instance components.

The code fragment in Figure 6 illustrates some key design decisions of the activity manager: how to maintain the label mapping throughout the life cycle of an instance, and when to allow, deny, or delay a call. The omitted part of the encoding deals with cross-application calls; there, we need to enforce both the application-level policies and the component-level policies.

An application resembles a single-instance component: both allow only one active instance and both admit shared state within each instance; hence, the life cycle of applications is managed similarly to that of single-instance components, except that application exit is more complicated as it has to allow all its active component instances to exit.

Overall System We assume that an initial process init bootstraps the system. This process 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.

$$S = \mathsf{T_M} \mid \mathsf{A_M} \mid App_1(aid_1) \mid \cdots \mid App_n(aid_n) \mid \mathsf{init}$$

5. Noninterference

To show that our enforcement mechanism prevents information leakage, we prove a noninterference theorem. We formally define noninterference in terms of trace equivalence of processes, and sketch the strategy of the proof that our enforcement mechanism guarantees noninterference.

We use the simple label κ_L as the label of malicious components. For convenience, we call components whose labels are not lower than or equal to κ_L *high components*, and those with labels lower than or equal to κ_L *low components*. We want to show that, from the malicious components' point of view, a system S that contains both high and low components behaves the same as a system composed of only the low components in S. We rely on definitions of process equivalence to specify when two systems behave the same.

Choice of Process Equivalence Two of the most commonly used definitions of process equivalence are trace equivalence and barbed bisimulation. Processes P and Q are trace equivalent if for any trace t generated by P, Q can generate an equivalent trace t', and vice versa. Barbed bisimulation is a stronger notion of equivalence: two processes that are trace equivalent may not be bisimilar. Bisimulation 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 we 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. We next describe a timing channel involving floating components that prevents bisimulation-based noninterference from being established.

We focus on secrecy labels, and use H as high and L as low. Component A is high, component B is single-instance, and has a floating low label, and components C and D are high. Component A starts first. If its secret is 1, it calls B; otherwise, it does not. D eventually calls B. Whenever B receives a call, it attempts to call C. Consider the following: If C receives a call from B then C learns that A's secret is 0 or that D called C before A did. If C does not receive a call from B for a long time, it guesses that A's secret is 1. When A calls B, B's label is instantiated to H, and its subsequent call to C is denied by the reference monitor. Therefore, C gains knowledge about A's secret based on whether it receives a call from B. This would not occur if B was a multi-instance component. D's call to B would start another instance of B, which would always be able to call C. Thus, whether or not A calls B would not change the number of calls that C receives from B.

The proof of bisimulation-based noninterference fails if we allow single-instance components or application labels to float. Trace equivalence-based noninterference does not flag this as an information leak, because the system with high components can always delay A's call and process D's call first, allowing B's call to C always to succeed. This is only feasible because trace equivalence does not require lock-step simulation and thus the trace generated by the system with high components can simulate the system without high components starting from the initial state.

One interesting observation is that bi-simulation-based noninterference can rule out certain timing attacks. The reason is that bisimulation implicitly assumes a fair scheduler that schedules every possible transition, since the definition requires every τ transition to be simulated.

Rather than disallowing floating labels, we compromise by using a weaker definition of non-interference. This definition is nevertheless strong enough that non-interference would not hold if our system allowed: (1) explicit communication between a high component and a low component; or (2) implicit leaks in the implementation of the reference monitor, such as branching on information from a high component and affecting low components differently depending on the branch. We believe, hence, that this definition of non-interference, though not as strong as a bisimulation-based definition, provides substantial assurance of our system's ability to prevent impermissible information flows.

High and Low Components We use the label contexts of processes to identify the high and low components in the system. The current label of a process can be deduced from these label contexts together with 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 the component level and application level. We would like to define a notion of noninterference that demonstrates 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 to decide whether a process is high or low for the our noninterference theorem.

Without loss of generality, we pick one application, whose components do not access the shared state, and decide whether each of its components is high or low solely based on its component-level label; while all other applications are treated as high or low at the granularity of an application, based on their application-level labels. For the rest of this section, 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 more formally. We 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.

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

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. Formal definitions of rules for deciding whether processes of this and other forms are high or low can be found in Appendix G.

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 results 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 by sending and receiving intents.

We assume that an intent I belongs to a (possibly infinite) set of constant channel names; i.e., intents are also modeled as channels. To facilitate our definition of noninterference, the encoding of components includes a special output action $\langle \mathsf{out}\, I(\mathsf{self})\rangle$ (Figure 3). This outputs to the intent channel the pair of the application ID and the current label of the instance. Here, self denotes the run-time labels of the the current application-instance and component-instance[1].

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 define Android-specific transitions rules consisting of (1) ordinary π-calculus transition rules for component instances, (2) the atomic transition rules for the reference monitor, and (3) two special transition rules shown below. We write a small-step transition as $S \xrightarrow{\alpha}_A S'$, and $S \xRightarrow{\tau}_A S'$ to denote zero or multiple τ transitions from S to S'. Relevant definitions can be found in Appendix E and F.

$$\frac{\Xi\langle c_{AI}\rangle = kA \qquad \Xi\langle c_{nI}\rangle = kC}{\nu\vec{x}.P \mid T_{MI}(\Xi) \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[\mathsf{out}\, I(\mathsf{self}).P]]]] \xrightarrow{\mathsf{out}\, I((aid,(kA,kC)))}_A \nu\vec{x}.P \mid T_{MI}(\Xi) \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]]}$$

$$\frac{\Xi\langle c_{AI}\rangle = kA \qquad \Xi\langle c_{nI}\rangle = kC}{\mathsf{A_M} \mid \mathsf{T_M} \mid \nu\vec{x}.T_{MI}(\Xi) \mid Q \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[\mathsf{out}\, I(\mathsf{self}).P]]]] \xrightarrow{\mathsf{out}\, I((aid,(kA,kC)))}_A \mathsf{A_M} \mid \mathsf{T_M} \mid \nu\vec{x}.T_{MI}(\Xi) \mid Q \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]]}$$

The first rule inserts the current label into the request so that we can annotate the activity manager with appropriate labels. The purpose of the second rule is to annotate the label of that output event.

Traces that we care about are composed of output actions to intent channels ($\mathsf{out}\, I((aid, (kA, kC)))$). We call such an action low, if $(aid, (kA^*, kC^*)) \sqsubseteq_{aid_c} \kappa_L$; and high otherwise.

Noninterference We define the projection of traces $t|_{\kappa_L}^{aid_c}$, 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 either lower than or at κ_L relative to aid_c. We call a sequence of transitions $\xRightarrow{t}_A$ *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 trace-equivalence-based noninterference theorem. The noninterference 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. Nonetheless, the theorem still captures the requirements on both cross-application and intra-application communications.

Theorem 5.1 (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 $S = \mathsf{A_M} \mid \mathsf{T_M} \mid$ App$(aid_1), \cdots,$ App(aid_n) be the initial system configuration, $S \Longrightarrow_A S'$,
$S' = \mathsf{A_M} \mid \mathsf{T_M} \mid \nu\vec{c}.(T_{MI}(\Xi) \mid AC(aid_c) \mid S'')$,
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 $\Xi' = \mathsf{projT}(\Xi; \kappa_L; aid_c)$,
$S_L = \mathsf{A_M} \mid \mathsf{T_M} \mid \nu\vec{c}'.(T_{MI}(\Xi') \mid \mathsf{proj}(AC(aid_c) \mid S'', \kappa_L, aid_c, \Xi'))$

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

Proof Strategy Proving condition (2) in Theorem 5.1 is trivial, since any trace generated by the system without high components can be generated by the system with high components. The harder case is condition (1). We would like to show that the system with high components simulates the system without high components. A simulation relation is defined as follows:

Definition 1 (Γ-simulation)**.** *Γ is a set of names. $\mathcal{R}$ is a simulation if and only if $P\ \mathcal{R}\ Q$ implies*

- *$P \xrightarrow{\tau} P'$, then exists Q' such that $Q \xRightarrow{\tau} Q'$, and $P'\ \mathcal{R}\ Q'$*
- *for every $b \in \Gamma$, $P \xrightarrow{\mathsf{out}\, b(\ell)} P'$ and $\ell \sqsubseteq_{aid_c} \kappa_L$ implies exists Q' such that $Q \xRightarrow{\mathsf{out}\, b(\ell)} Q'$, and $P'\ \mathcal{R}\ Q'$*

We define a relation $P\ \mathcal{R}_{(aid_c,\kappa_L)}\ Q$, which requires P and Q to agree on low components. We then prove that (1) $\mathcal{R}_{(aid_c,\kappa_L)}$ is a simulation relation, (2) S and the projection of S are related by $\mathcal{R}_{(aid_c,\kappa_L)}$. Then, by induction on the length of traces, we can prove condition (1) for Theorem 5.1.

6. Implementation and Case Study

We implemented our system on top of Android 4.0.4, leveraging techniques similar to those used by several other works (e.g., [4, 15]). Here we give a high-level overview and describe in detail our policy for the example scenario from Section 3.1.

[1] Note that the label could change between receiving the intent and output to the intent channel if another component raises the application-level label. This is not relevant to our noninterference proofs since the message processing is inherently asynchronous, and there is no guarantee that the label of the instance will remain the same between the receipt and the processing of a message.

Policy Specification We extend Android's manifest file syntax for declaring and using permissions to also specify information-flow labels. We treat each permission as a tag, and add a flag to indicate whether it is an integrity tag or a secrecy tag, or a declassification or endorsement capability. The static label of an application or component is thus derived by processing all the permission declaration statements. Unlike in stock Android, we allow both applications and components to be assigned labels; component labels are also defined in the application's manifest file.

Reference Monitor Our enforcement system is designed around a central reference monitor that mediates communication between components. Android's activity manager (ActivityManager) already largely accomplishes this, although it enforces a much less rich policy than we desire and often does not have sufficient information to enforce our information-flow policies.

We modify the activity manager so that mediation of all relevant calls is handled by our own system instead. Enforcing our information-flow policies requires substantial additional bookkeeping (the data structures of the label manager) for tracking running instances and their effective labels. We also modify the activity manager to automatically apply declassification and endorsement when these are necessary for a call to succeed; a developer can, if she wishes, explicitly override this, but the automation eases the burden on developers and helps with backward compatibility.

The biggest challenges in implementing our enforcement system were in providing more detailed information to the activity manager than it was previously basing its decisions on. More specifically, we had to enhance Android's IPC data structures to carry information about the calling component instance, rather than just about the calling component instance's application. Another challenge was to capture operations not mediated by the activity manager, such as opening a socket or a file. Android's permission-based policies that govern such operations are implemented not by the activity manager, but by access-control checks in the underlying OS. We used TOMOYO [25], middleware that replaces scattered, low-level access-control checks with centralized ones, and extended TOMOYO to redirect its access checks from its own reference monitor to ours. In addition to giving our reference monitor more direct control over system behaviors that Android's activity manager did not directly mediate, this approach has the advantage of extending our enforcement mechanism to application fragments written in native code.

Backwards Compatibility To aid in backward compatibility with legacy applications, we mapped system-declared permissions to secrecy and integrity tags: we mapped permissions that protect data resources from reading to secrecy tags; and permissions that protect data resources from being written, and security-relevant interfaces from being accessed, to integrity tags. In the absence of component-level policy (which is always absent for legacy applications), we assign to components their enclosing application's labels.

To capture the flow of information when a security-relevant Android or Java API method is called, we assigned signatures to API methods to express the secrecy and integrity labels that the caller should have to be able to call the method and to get a result. We assigned signatures to a fraction of the Java and Android APIs, but extending this to all of them involves no technical challenges.

This approach to obtain labels from permissions allowed us to compile existing applications without modifying their code. However, using these labels directly, some calls that would be permitted by the permission system will now be denied. Next we explain why and how to address this.

One reason why a call is denied is because it potentially leaks information. In Android's permission system, the caller normally does not specify policies. In our system, caller's permissions could be translated into secrecy tags. For example, the permission system allows the Messaging application to be started by the Contacts application to send a text message to a specific contact. The Contacts app has several secrecy tags that the Messaging app does not, such as Read_Social_Stream, which allows it to access, e.g., photos posted on social networks. The Contacts app could potentially leak this information through the Messaging app. In this case, to allow in our system the call (to start the Messaging app) that was allowed in stock Android, we give the Contacts app declassification capabilities for these secrecy tags that the Messaging app does not have. Since the Contacts app is built-in, this level of trust is appropriate, because these built-in Apps are considered trusted. Similar scenarios apply to integrity tags as well (please see our tech report for more detail).

We assigned declassification and endorsement capabilities manually to make sure that our assignments don't sacrifice security. During this process we identified applications that violate the principle of least privilege asking more permissions than they actually need. An example is that the Camera application requests a permission to read the SMS content provider without ever using it.

Motivating Scenario Revisited We now show how to specify the policy from our example from Section 3.1 and walk through its enforcement. Recall that our scenario involves a secret-file manager, a text editor, and an email application. The high-level policy is to prevent secret files from being leaked on the Internet, but to allow them to be manipulated by applications and emailed at the user's behest. The files may be edited, but the editor must not leak them via the network; files 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 the policy ({ReadContacts, ...}, {}, {+Internet, ...}). The ...in the policy are various additional secrecy tags and endorsement capabilities that enable the email application to read user accounts (ReadContacts), cause the phone to vibrate, access the network state (+Internet), 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 the policy (F{}, F{}, {-T, +WriteExternalStorage}). It has floating secrecy and integrity labels and can declassify all secrets it acquires from its callers (presumably after encrypting them). This gives it the ability to leak any secret, 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 in the following way: The editor's label floats to ({FileSecret}, {FileWrite}, {}) when it is called by the file manager. As the editor cannot declassify FileSecret, it cannot leak the file; because it has FileWrite, it can save the file back into secret storage. However, if the user wants to email the file, she can invoke the email application directly through the file manager. The file manager adds the file content to the intent that starts the email application, and removes the FileSecret tag by declassifying before sending the intent. The file can also be released, legitimately, via the encryption application. If invoked by the file manager, the encryption application floats to ({FileSecret}, {FileWrite}, {-T, +WriteExternalStorage}), but its capability to declassify any secret (-T) allows it to declassify and release it to any application.

Contacts application:
{{ReadContacts, GetAccounts, AccessFineLocation,
AccessCoarseLocation, ReadProfile, ReadSocialStream,
ReadPhoneState, ReadSyncSettings, GoogleAuthMail,
ReadWriteAllVoiceMail},
{CallPrivileged,WriteContacts, ManageAccounts, WriteProfile,
Internet, Nfc, ModifyAudioSettings, ModifyPhoneState,
WakeLock, WriteExternalStorage, WriteSettings, UseCredentials,
Vibrate, AddVoicemail, ReadWriteAllVoiceMail, Reboot,
AllowAnyCodecForPlayback, ReceiveBootCompleted},
{-GetAccounts, -ReadSocialStream, -ReadSyncSettings,
+-ReadWriteAllVoicemail}}

Mms application:
{{ReadContacts, ReadProfile, ReadSms, AccessNetworkState,
ReadPhoneState, AccessFineLocation, AccessCoarseLocation},
{ReceiveBootCompleted, CallPhone, WriteContacts,
ReceiveSms, ReceiveMms, SendSms, Vibrate, Internet
WriteSms, ChangeNetworkState, WakeLock,
WriteExternalStorage, InstallDrm},
{+ReceiveBootCompleted, +CallPhone, +WriteContacts,
+ReceiveSms, +ReceiveMms, +SendSms, +Vibrate,
+WriteSms, +ChangeNetworkState, +InstallDrm}}

Figure 7. The labels for the Contacts application and Mms application (the built-in messaging application) as used in our measurements.

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 and Performance Measurements Our case study is fully implemented and has been tested on a Nexus S phone. As part of booting the phone to the point where it can execute ordinary applications, over 50 Google-provided applications start running; this provided a good test of our backward compatibility and aided in debugging our system. Our case study used minimally modified off-the-shelf applications: Open Manager v2.1.8, Qute Text Editor v0.1, Android Privacy Guard v1.0.9, Email v2.3.4. We modified manifest files, added sending functionality to some, and added a content provider to Open Manager. Our system's implementation totaled approximately 1200 lines of code, not counting TOMOYO: approximately 650 in the reference monitor, 400 for bookkeeping, 100 for enhancing IPCs, and 50 for syntactic support for policies specified as labels.

Based on informal testing, the run-time performance of our system is sufficiently good for any additional latencies not to be observable to the user in practice.

We also ran microbenchmarks: we measured the overhead our system adds to the time it takes for an intent to be delivered, as well as the main component of that overhead that is added by our system, the time it takes to perform the checks incurred by each call. All measurements were taken on a Nexus S phone running Android compiled from unmodified source version 4.0.4_r1.1 (referred to as *stock Android*) or the same version of Android with our modifications added (referred to as *our system*), and were taken over 200 runs.

The measurements we report on are for intents sent between components of two Google-provided applications that are packaged with stock Android, the PeopleActivity component of the Contacts application (com.android.contacts) and the ComposeMessageActivity component of the Mms application (com.android.mms, Android's built-in messaging application). Each component's labels are the same as those for the enclosing application and are shown in Figure 6. The overall time to send an intent we measure from immediately before the call to startActivity(...) inside com.android.contacts.interactions.PhoneNumberInteraction.performAction(...) until the first line of com.android.mms.ui.ComposeMessageActivity.onCreate(...). For stock Android, the average measured time is 64.0 ms (std. dev. 11.3 ms; median 64.0 ms); for our system the average is 51.0 ms (std. dev. 29.7 ms; median 51.0 ms). That the average for our system is lower than for stock Android shows the inability of such benchmarks to accurately capture overheads in multithreaded environments with many processes running simultaneously, but the benchmark results also show that the overhead incurred by our system is small relative to the time it takes to deliver an intent.

When focusing just on the time our system takes to perform the four label checks required to decide whether a call between these two components should be allowed, we find that the set of four checks takes an average of 7.5 ms (std. dev. 6.4 ms; median 7.5 ms).

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.

References

[1] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE S&P*, 2012.

[2] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. POPL*, 2012.

[3] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proc. CCS*, 2009.

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

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

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

[7] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proc. ISC*, 2010.

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

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

[10] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. OSDI*, 2010.

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

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

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

[14] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. of Comput. Secur.*, 3:5–33, 1994.

[15] E. Fragkaki, L. Bauer, and L. Jia. Modeling and enhancing Android's permission system. In *Proc. ESORICS*, 2012.

[16] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. 2009.

[17] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE CSF*, 2012.

[18] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proc. CCS*, 2011.

[19] M. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *Proc. IEEE S&P*, 2009.

[20] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. SOSP*, 2007.

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

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

[23] A. C. Myers. Practical mostly-static information flow control. In *Proc. POPL*, 1999.

[24] M. Nauman, S. Khan, and X. Zhang. 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] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel. Semantically rich application-centric security in Android. In *Proc. ACSAC*, 2009.

[27] P. Passeri. 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] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Proc. PLAS*, 2011.

[29] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. *J. Comput. Secur.*, 9(1–2), 2001.

[30] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal Sel. Area. Comm.*, 21(1):5–19, 2003.

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

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

[33] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proc. SOSP*, 2009.

[34] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. OSDI*, 2006.

[35] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. NSDI*, 2008.

[36] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. PLDI*, 2012.

Appendix

A. Additional Definitions for the Process Calculus

Structural Congruence *Structural congruence* $\equiv$ is the smallest congruence on processes that satisfies the axioms below.

$$
\begin{array}{lcl}
id[\,P + Q\,] & \equiv & id[\,P\,] + id[\,Q\,] \\
id[\,P \mid Q\,] & \equiv & id[\,P\,] \mid id[\,Q\,] \\
id[\,\mathbf{0}\,] & \equiv & \mathbf{0} \\
id[\,\nu x.P\,] & \equiv & \nu x.id[\,P\,] \quad \text{if } x \notin \mathsf{fn}(id) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
P \mid Q & \equiv & Q \mid P \\
P \mid \mathbf{0} & \equiv & P \\
P + Q & \equiv & Q + P \\
P + (Q + R) & \equiv & (P + Q) + R \\
\nu x.\mathbf{0} & \equiv & \mathbf{0} \\
P + \mathbf{0} & \equiv & P \\
\nu x.\nu y.P & \equiv & \nu y.\nu x.P \\
\nu x.(P \mid Q) & \equiv & P \mid \nu x.Q \quad \text{if } x \notin \mathsf{fn}(P) \\
!P & \equiv & P \mid !P
\end{array}
$$

Labeled Transition Rules A complete list of the labeled transition rules is shown in Figure 8. The rule for pattern-matched input requires the existence of a substitution σ for bound variables in patt, such that the value z output on channel x is the same as the term resulting from applying the substitution to the pattern $\mathsf{patt}(\sigma)$.

Lemma A.1. *If* $P \xrightarrow{\alpha} P'$*, and* $P \equiv Q$*, then exists* Q' *such that* $Q \xrightarrow{\alpha} Q'$ *and* $Q' \equiv P'$

Proof. By induction on the structural congruence relation. □

B. Lemmas About Label Operations

We prove several key properties about label operations (Figure 2), which will be used in the noninterference proofs.

Lemma B.1. *If* ${K_2}^- \sqsubseteq {K_1}^-$ *then* $K_2^* \sqsubseteq (K_1 \lhd K_2)^*$ *and* $K_1^* \sqsubseteq (K_1 \lhd K_2)^*$.

Lemma B.2. $K_1^* \sqsubseteq K_1 \uplus_M K_2^*$ *and* $K_2^* \sqsubseteq K_1 \uplus_M K_2^*$

C. Encoding of the Label Manager

The encoding of the label manager is shown in Figure 9. The label manager is launched from channel t_t, and the argument sent to t_t is the current label map of the system. Requests to the label manager are sent to the channel t_m. At the end of processing each request, the label manager will launch itself again with the current label map. This also ensures that at any time there is only one active copy of the label manager running.

Note that we do not allow a floating application and single-instance component to raise their labels. As we discussed in Section 3.2.2, the raise operation has to raise both the static and the effective label. If we allowed a floated instance to raise its label, we would have allowed a high component (this floated instance) to affect low components by changing its static label, which could be low.

D. Encoding of the Activity Manager

Recall that the top-level process of the activity manager is of the form: $\mathsf{A_M} = !(AM_I + AM_E + AM_{EX} + AM_R)$.

We have explained the encoding of the sub-process of the activity manager that handles intra-application calls (Figure 6) in Section 4.3. Here, we show the encoding of AM_E, which processes

Activity Manager Inter-app Call $AM_E =$

68 $\mathsf{in}\ a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I).$

69 $\quad \nu r.\mathsf{out}\ t_m(\mathsf{lookUp}, aid_{ce}).\mathsf{in}\ r(s1).\ \mathsf{out}\ t_m(\mathsf{lookUp}, aid_{ce}{\cdot}cid_{ce}, r).\mathsf{in}\ r(s2).$

70 $\quad\quad \mathsf{case}\ (s1, s2)\ \mathsf{of}\ (\mathsf{NE}, \_) \quad \Rightarrow \mathbf{0}$

71 $\quad\quad \mid (\_, \mathsf{NE}) \quad \Rightarrow \mathbf{0}$

72 $\quad\quad \mid (\mathsf{U}(kA_{ce}), \mathsf{S}(kC_{ce})) \quad \Rightarrow \mathsf{if}\ \ kA_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kA_{cr}{}^- \sqsubseteq kC_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kC_{ce}{}^-$

73 $\quad\quad\quad\quad\quad\quad \mathsf{then}\ \nu c_{AI}.\nu c_{nI}.\nu c_{lock}.$

74 $\quad\quad\quad\quad\quad\quad\quad \mathsf{out}\ t_m(\mathsf{upd}, \{\ (aid_{ce}, \mathsf{L}(kA_{ce}, c_{AI})),$

75 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (c_{AI}, \mathsf{DA}(kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr}), \mathsf{LC}(\{c_{nI}\}))),$

76 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (aid{\cdot}cid_{ce}, \mathsf{SL}(c_{nI}, kC_{ce})),$

77 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\}).$

78 $\quad\quad\quad\quad\quad\quad\quad (aid_{ce}, c_{AI})[\mathsf{out}\ aid_{ce}{\cdot}c_L(c_{AI})]$

79 $\quad\quad\quad\quad\quad\quad\quad \mid\ (aid_{ce}, (c_{AI}, c_{nI}))[\mathsf{out}\ aid_{ce}{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, \mathsf{NONE})]$

80 $\quad\quad\quad\quad\quad\quad \mathsf{else}\ \mathbf{0}$

81 $\quad\quad \mid (\mathsf{U}(kA_{ce}), \mathsf{M}(kC_{ce})) \quad \Rightarrow \mathsf{if}\ \ kA_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kA_{cr}{}^- \sqsubseteq kC_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kC_{ce}{}^-$

82 $\quad\quad\quad\quad\quad\quad \mathsf{then}\ \nu c_{AI}.\nu c_{nI}.\nu c_{lock}.$

83 $\quad\quad\quad\quad\quad\quad\quad \mathsf{out}\ t_m(\mathsf{upd}, \{\ (aid_{ce}, \mathsf{L}(kA_{ce}, c_{AI})),$

84 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (c_{AI}, \mathsf{DA}(kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr}), \mathsf{LC}(\{c_{nI}\}))),$

85 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\}).$

86 $\quad\quad\quad\quad\quad\quad\quad (aid_{ce}, c_{AI})[\mathsf{out}\ aid{\cdot}c_L(c_{AI})]$

87 $\quad\quad\quad\quad\quad\quad\quad \mid\ (aid_{ce}, (c_{AI}, c_{nI}))[\mathsf{out}\ aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, rt)]$

88 $\quad\quad\quad\quad\quad\quad \mathsf{else}\ \mathbf{0}$

89 $\quad\quad \mid (\mathsf{L}(c_{AI}, kA_{ce}), \mathsf{S}(kC_{ce})) \quad \Rightarrow \mathsf{out}\ t_m(\mathsf{lookUp}, c_{AI}, r).\mathsf{in}\ r(\mathsf{DA}(kA_d, \_)).$

90 $\quad\quad\quad\quad\quad\quad \mathsf{if}\ \ kA_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kA_{cr}{}^- \sqsubseteq kC_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kC_{ce}{}^-$

91 $\quad\quad\quad\quad\quad\quad \mathsf{then}\ \mathsf{if}\ \ (\mathit{conrete}(kA_{ce}) \wedge\ kA_{cr}{}^- \sqsubseteq kA_d{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_d{}^-\ )$

92 $\quad\quad\quad\quad\quad\quad\quad\quad \vee(\mathit{floating}(kA_{ce}) \wedge\ kA_{cr}{}^- = kA_d{}^- \wedge\ kC_{cr}{}^- = kA_d{}^-\ )$

93 $\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{then}\ \nu c_{nI}.\nu c_{lock}.\mathsf{out}\ t_m(\mathsf{upd}, \{\ (c_{AI}, \{c_{nI}\}),$

94 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (aid{\cdot}cid_{ce}, \mathsf{SL}(c_{nI}, kC_{ce})),$

95 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\}).$

96 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (aid_{ce}, (c_{AI}, c_{nI}))[\,\mathsf{out}\ aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, \mathsf{NONE})\,]$

97 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ \ (aid_{cr}, (kA_{cr}, kC_{cr}))[\,\mathsf{out}\ a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)\,]$

98 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ \mathbf{0}$

99 $\quad\quad \mid (\mathsf{L}(c_{AI}, kA_{ce}), \mathsf{SL}(c_{nI}, kC_{ce}))$

100 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathsf{out}\ t_m(\mathsf{lookUp}, c_{AI}, r).\mathsf{in}\ r(\mathsf{DA}(kA_d, \_)).$

101 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{out}\ t_m(\mathsf{lookUp}, c_{nI}, r).\mathsf{in}\ r(\mathsf{DC}(c_{lock}, \_)).\ \mathsf{in}\ c_{lock}().$

102 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{out}\ t_m(\mathsf{lookUp}, c_{nI}, r).\mathsf{in}\ r(\mathsf{DC}(\_, kC_d)).$

103 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{if}\ \ kA_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kA_{cr}{}^- \sqsubseteq kC_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kC_{ce}{}^-$

104 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{then}\ \mathsf{if}\ \ (\mathit{conrete}(kA_{ce}) \supset (kA_{cr}{}^- \sqsubseteq kA_d{}^- \wedge kC_{cr}{}^- \sqsubseteq kA_d{}^-))$

105 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge (\mathit{floating}(kA_{ce}) \supset (kA_{cr}{}^- = kA_d{}^- \wedge kC_{cr}{}^- = kA_d{}^-))$

106 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge (\mathit{conrete}(kC_{ce}) \supset (kA_{cr}{}^- \sqsubseteq kC_d{}^- \wedge kC_{cr}{}^- \sqsubseteq kC_d{}^-))$

107 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge (\mathit{floating}(kC_{ce}) \supset (kA_{cr}{}^- = kC_d{}^- \wedge kC_{cr}{}^- = kC_d{}^-))$

108 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{then}\ \ (aid_{ce}, (c_{AI}, c_{nI}))[\,\mathsf{out}\ c_{nI}(I)\,]$

109 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ \ (aid_{ce}, (c_{AI}, c_{nI}))[\,\mathsf{out}\ c_{lock}()\,]$

110 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid (aid_{cr}, (kA_{cr}, kC_{cr}))[\,\mathsf{out}\ a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)\,]$

111 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ \ (aid_{ce}, (c_{AI}, c_{nI}))[\,\mathsf{out}\ c_{lock}()\,]$

112 $\quad\quad \mid (\mathsf{L}(c_{AI}, kA_{ce}), \mathsf{M}(kC_{ce})) \quad \Rightarrow \mathsf{out}\ t_m(\mathsf{lookUp}, c_{AI}, r).\mathsf{in}\ r(\mathsf{DA}(kA_d)).$

113 $\quad\quad\quad\quad\quad\quad \mathsf{if}\ \ kA_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kA_{cr}{}^- \sqsubseteq kC_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kC_{ce}{}^-$

114 $\quad\quad\quad\quad\quad\quad \mathsf{then}\ \mathsf{if}\ \ (\mathit{conrete}(kA_{ce}) \wedge\ kA_{cr}{}^- \sqsubseteq kA_d{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_d{}^-\ )$

115 $\quad\quad\quad\quad\quad\quad\quad\quad \vee(\mathit{floating}(kA_{ce}) \wedge\ kA_{cr}{}^- = kA_d{}^- \wedge\ kC_{cr}{}^- = kA_d{}^-\ )$

116 $\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{then}\ \nu c_{nI}.\nu c_{lock}.\mathsf{out}\ t_m(\mathsf{upd}, \{\ (c_{AI}, \{c_{nI}\}),$

117 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\}).$

118 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (aid_{ce}, (c_{AI}, c_{nI}))[\,\mathsf{out}\ aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, rt)\,]$

119 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ \ (aid_{cr}, (kA_{cr}, kC_{cr}))[\,\mathsf{out}\ a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)\,]$

120 $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ \mathbf{0}$

121 $\quad\quad \mid (\mathsf{onEx}(), \_) \quad \Rightarrow \mathbf{0}$

Figure 11. Encoding of the subprocess of the activity manager that handles calls between applications.

$$\frac{}{\mathsf{out}\, x(y).P \xrightarrow{\mathsf{out}\, x(y)} P} \qquad \frac{}{\mathsf{in}\, x(y).P \xrightarrow{\mathsf{in}\, x(z)} P[z/y]}$$

$$\frac{\mathsf{patt}(\sigma) = z}{\mathsf{in}\, x(\mathsf{patt}).P \xrightarrow{\mathsf{in}\, x(z)} P(\sigma)} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \qquad \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\frac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\frac{P \xrightarrow{\mathsf{in}\, x(y)} P' \qquad Q \xrightarrow{\mathsf{out}\, x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{\mathsf{out}\, x(y)} P' \qquad Q \xrightarrow{\mathsf{in}\, x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P \xrightarrow{\mathsf{in}\, x(y)} P' \qquad Q \xrightarrow{\nu y.\mathsf{out}\, x(y)} Q' \qquad y \notin \mathsf{fn}(Q)}{P \mid Q \xrightarrow{\tau} \nu y.(P' \mid Q')}$$

$$\frac{P \xrightarrow{\nu y.\mathsf{out}\, x(y)} P' \qquad Q \xrightarrow{\mathsf{in}\, x(y)} Q' \qquad y \notin \mathsf{fn}(P)}{P \mid Q \xrightarrow{\tau} \nu y.(P' \mid Q')}$$

$$\frac{P \xrightarrow{\alpha} P' \qquad x \neq \mathsf{n}(\alpha)}{\nu x.P \xrightarrow{\alpha} \nu x.P'} \qquad \frac{P \xrightarrow{\mathsf{out}\, x(y)} P' \qquad x \neq y}{\nu y.P \xrightarrow{\nu y.\mathsf{out}\, x(y)} P'}$$

$$\frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(id) = \emptyset}{id[\,P\,] \xrightarrow{\alpha} id[\,P'\,]} \qquad \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

$$\frac{P \xrightarrow{\mathsf{in}\, x(y)} P' \qquad P \xrightarrow{\mathsf{out}\, x(y)} P''}{!P \xrightarrow{\tau} P' \mid P'' \mid !P}$$

$$\frac{P \xrightarrow{\mathsf{in}\, x(y)} P' \qquad P \xrightarrow{\nu y.\mathsf{out}\, x(y)} P'' \qquad y \notin \mathsf{fn}(P)}{!P \xrightarrow{\tau} \nu y.(P' \mid P'') \mid !P}$$

$$\frac{}{\mathsf{if\ true\ then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_1} \qquad \frac{}{\mathsf{if\ false\ then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_2}$$

$$\mathsf{case}\ ctr_i\ \vec{e}_i\ \mathsf{of}\{\mid ctr_1\vec{x}_1 \Rightarrow P_1 \cdots \mid ctr_n\vec{x}_n \Rightarrow P_n\} \xrightarrow{\tau} P_i[\vec{e}_i/\vec{x}_i]$$

Figure 8. Labeled Transition Rules for the Core Calculus

inter-application calls (Figure 11); AM_{EX}, which processes exits; and AM_R, which processes returns (Figure 10).

For the inter-application calls, we need to consider four kinds of labels: the application labels of the caller and the callee, and the component labels of the caller and the callee. Based on possible combinations of the kinds of the callee's application label and the component label, we have several cases to consider. We next describe several of the more interesting cases.

When the callee application is unlaunched, the callee component must be unlaunched as well. In this case, the activity manager allows the call only if the caller's current application label and component label are each lower than or equal to the static labels of the callee. If the call is allowed, the activity manager generates a new application instance channel c_{AI} and updates the application label from unlaunched to launched. The activity manager also launches a new instance for the callee component in the same way as in the intra-application case. It records the new instance of the component in the mapping of c_{AI}. Finally, it sends a message to the launch channel of the callee application, and sends the intent to the create channel of the callee component.

Note that the instantiation of a floating label takes both the caller's application label and the caller's component label into consideration. To help with this, we use the label merge operation $\kappa_1 \uplus_M \kappa_2$, which produces a simple label whose secrecy label is the union of the secrecy labels of κ_1 and κ_2, and whose integrity label is the intersection of the integrity labels of the two (Figure 2).

When the callee application is launched, and the callee component is either multi-instance or unlaunched single-instance, the activity manager's label checks also take into account the callee application's instance label. As mentioned earlier, an application is similar to a single-instance component. Therefore, when the static label of the callee application is a floating label, we delay the call unless

Label Mngr $\mathsf{T_M}$ =

```
0  !( in t_t(x).
1       in t_m(lookUp, id, r). out r(x(id)).out t_t(x)
2     +in t_m(cleanA, aid, cid, c_AI, c_nI).
3       case x(aid)of
4       L(c_AI, k) ⇒ out t_t(x[aid ↦ onEx(c_AI, k)]).
5                    out t_m(cleanC, aid, cid, c_AI, c_nI)
6       |_         ⇒ out t_m(cleanC, aid, cid, c_AI, c_nI)
7     +in t_m(cleanC, aid, cid, c_AI, c_nI).
8       case x(aid.cid)of SL(c_AI, c_nI, c_lock, kC_s) ⇒
9         if x(c_AI) = (k, {c_nI}) ∧ x(aid) = onEx(c_AI, kA_s)
10        then out t_t(x [aid.cid ↦ S(kC_s)]
11                       [aid ↦ U(kA_s)]\c_AI\c_nI)
12        else out t_t(x[aid.cid ↦ S(kC_s)]
13                      [c_AI ↦ x(c_AI)\c_nI]\c_nI)
14      | M(kC_s) ⇒if x(c_AI) = (kA_d, {c_nI})
15                    ∧ x(aid) = onEx(c_AI, kA_s)
16                  then out t_t(x[aid ↦ U(kA_s)]\c_AI\c_nI)
17                  else out t_t(x[c_AI ↦ x(c_AI)\c_nI]\c_nI)
18    +in t_m(dclassifyA, c_AI, δ).
19      if dL(x(c_AI)) ⊒ δ
20      then out t_t(x[c_AI ↦ x(c_AI) ⊎_d δ])
21      else out t_t(x)
22    +in t_m(dclassifyC, c_nI, δ).
23      if dL(x(c_nI)) ⊒ δ
24      then out t_t(x[c_nI ↦ x(c_nI) ⊎_d δ])
25      else out t_t(x)
26    +in t_m(raiseA, aid, c_AI, σ, ι).
27      if floating(x⟨aid⟩)
28      then out t_t(x)
29      else out t_t(x[aid ↦ x(aid) ⊎_rz (σ, ι)]
30                    [c_AI ↦ x(c_AI) ⊎_rz (σ, ι)])
31    +in t_m(raiseC, aid, cid, c_nI, d).
32      case x(aid·cid) of M(k) ⇒
33       out t_t(x[c_nI ↦ x(c_nI) ⊎_rz (σ, ι)])
34      | SL(_) ⇒
35       if floating(x⟨aid·cid⟩) ∨ floating(x⟨aid⟩)
36       then out t_t(x)
37       else out t_t(x[aid·cid ↦ x(aid·cid) ⊎_rz (σ, ι)]
38                     [c_nI ↦ x(c_nI) ⊎_rz (σ, ι)])
```

Figure 9. Encoding of the label manager.

Activity Manager Exit AM_{EX} =

39 in $a_m(\mathsf{exitA}, kA_{ce}, kC_{ce}, aid_{ce}, cid_{ce}, c_{AI_{ce}}, c_{nI_{ce}}, rt, I)$.
40 out $t_m(\mathsf{cleanA}, aid_{ce}, cid_{ce}c_{AI_{ce}}, c_{nI_{ce}})$.
41 case *rt* of NONE $\Rightarrow \mathbf{0}$
42 | SOME$(aid_{cr}, c_{AI}, c_{nI}, c_{lock}) \Rightarrow$
43 $(aid_{ce}, (kA_{ce}, kC_{ce}))[$out $a_m(\mathsf{return}, kA_{ce}, kC_{ce}, aid_{ce},$
44 $c_{AI_{ce}}, aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)]$
45 + in $a_m(\mathsf{exitC}, kA_{ce}, kC_{ce}, aid_{ce}, cid_{ce}, c_{AI_{ce}}, c_{nI_{ce}} rt, I)$.
46 out $t_m(\mathsf{cleanC}, aid_{ce}, cid_{ce}, c_{AI_{ce}}, c_{nI_{ce}})$.
47 case *rt* of NONE $\Rightarrow \mathbf{0}$
48 | SOME$(aid_{cr}, c_{AI}, c_{nI}, c_{lock}) \Rightarrow$
49 $(aid_{ce}, (kA_{ce}, kC_{ce}))[$out $a_m(\mathsf{return}, kA_{ce}, kC_{ce}, aid_{ce},$
50 $aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)]$

Activity Manager Return AM_R =

51 in $a_m(\mathsf{return}, kA_{ce}, kC_{ce}, aid_{ce}, c_{AI_{ce}}, aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)$.
52 in $c_{lock}()$.
53 if $aid_{ce} = aid_{cr}$
54 then νr.out $t_m(\mathsf{lookUp}, c_{nI}, r)$.in $r(s)$.
55 case s of $(\_, kC_d) \quad \Rightarrow$
56 if ${kC_{ce}}^- \sqsubseteq {kC_d}^-$
57 then $(aid_{cr}, (c_{AI}, c_{nI}))[$out $c_{nI}(I)]$
58 else $(aid_{cr}, (c_{AI}, c_{nI}))[$out $c_{lock}()]$
59 $| \Rightarrow \mathbf{0}$
60 else νr.out $t_m(\mathsf{lookUp}, c_{AI}, r)$.in $r(s1)$.
61 out $t_m(\mathsf{lookUp}, c_{nI}, r)$.in $r(s2)$.
62 case $(s1, s2)$ of $((\mathsf{DA}(kA_d), \_), (\_, kC_d)) \Rightarrow$
63 if ${kA_{ce}}^- \sqsubseteq {kA_d}^- \wedge {kA_{ce}}^- \sqsubseteq {kC_d}^-$
64 $\wedge\ {kC_{ce}}^- \sqsubseteq {kA_d}^- \wedge {kC_{ce}}^- \sqsubseteq {kC_d}^-$
65 then $(aid_{cr}, (c_{AI}, c_{nI}))[$out $c_{nI}(I)]$
66 else $(aid_{cr}, (c_{AI}, c_{nI}))[$out $c_{lock}()]$
67 $| \Rightarrow \mathbf{0}$

Figure 10. Encoding of the subprocesses of the activity manager that handle exit and return calls.

the label of the caller and the callee match exactly. This is for the same reason as the delay for a single-instance floating component: the delay preserves noninterference without affecting the functionality of the caller (as much as a denied call would). If the call is allowed, the activity manager takes additional actions as in the case described above, except that no application-instance channel needs to be generated. However, the label manager's application-instance label map is updated to include the newly launched component instance.

Finally, when both the callee application and callee component are launched, the activity manager needs to check the instance labels. Similarly to the above case, calls may be delayed if either the application or the component is floating and the caller's labels do not match the dynamic labels of the callee.

When the activity manager receives a request to exit the component (application), it calls the label manager to update the mappings for the exiting component (application) instance, and restores labels for the component (application) if necessary. Please refer to the label manager encoding for details. If the return argument *rt* indicates that the caller is waiting for a result, the activity manager sends a request to itself to return to the caller based on the arguments in *rt*.

The return is handled similarly to a call to a launched single-instance component. The activity manager first waits for the lock of the component it wants to return to. When the activity manager obtaines the lock, it compares the application ID of the caller and the callee. If they are the same, then the return is treated as an intra-application return: only the component-level labels are compared to decide whether to allow the return or release the lock. If the application IDs of the caller and the callee are different, then the return is treated as an inter-application return, and both the application-level and the component-level labels are compared to decide whether to allow the return.

E. Stable Configurations

We define *stable configurations* as configurations where the activity manager and the label manager are at a *stable* state. We ignore the internal steps of the activity manager interacting with the label manager since these are invisible to applications. There is no need to model transitions of the Android system configuration at a finer granularity. We instead assume that activity and label managers take only atomic steps from a stable configuration to another stable configuration. We formally define these transition rules in Appendix F.

We first define relevant configurations of components and applications. We call the sub-processes of $CP(aid, cid, c_{AI}, c_{sv})$ after input to the creation channel a *starting configuration* of an instance of the component with ID $aid{\cdot}cid$. A process is an instance of the component with ID $aid{\cdot}cid$ if it is the starting configuration or any sub-process reduced from that starting configuration. Each instance is uniquely identified by its new intent channel c_{nI}. Similarly, we call the instantiated body of *App*(aid) a *starting configuration* of an instance of the application with ID aid. A process is an instance of the application with ID aid if it is the starting configuration or any sub-process reduced from the starting configuration. Each instance is uniquely identified by its instance channel c_{AI}.

We say that an application (component) instance is *active* when it is possible for that instance to reduce without input actions from other processes. *Terminal* instances are the ones that cannot reduce further without inputs from other processes. A terminal component instance (written *CompT*$(\vec{x})$) can be either an unlaunched component event loop, or the component loop body waiting for new intent calls. The first terminal configuration is reached after the instance executes either the application exit or the component exit instruction. In this case, the instance does not re-launch its local state (c_{ls}). Since no other process will send inputs to c_{ls}, this configuration cannot further reduce. The second kind of terminal instance has two subcases. The difference between the two is in the label context of the output to the lock channel. One is produced by the component instance, and therefore has a layered context; the other is produced by the activity manager, and therefore has a simple label context. The latter occurs when an activity manager instance obtained the lock, but the call was denied, and it has to re-release the lock so that other callers can proceed. Note that this simple label suffices for the proofs.

Terminal component instance CompT$(\vec{x})$ ::=

$aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[CE(\cdots)]]]]$
| $aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[CB(\cdots)]]]]$
$'|'\ aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[$out $c_{lock}()]]]]$
| $aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[CB(\cdots)]]]]$
$'|'\ (aid, (c_{AI}, c_{nI}))[$out $c_{lock}()]]]]$

Terminal application instance AppT$(\vec{x})$ =

$aid[c_{AI}[(c_{AI}, cid_1)[CP_1(aid, cid_1, c_{AI}, c_{sv})]]]$
$'|'\ \cdots$
$'|'\ aid[c_{AI}[(c_{AI}, cid_1)[CP_m(aid, cid_m, c_{AI}, c_{sv})]]]$
$'|'\ aid[c_{AI}[SV(c_{svL}, c_{sv})]]$
$'|'\ aid[c_{AI}[SVBody(c_{svL}, c_{sv})]]$
$'|'\ CT_1(\cdots) \cdots '|'\ CT_n(\cdots)$

A terminal application instance (written *AppT*$(\vec{x})$) contains all of its unlaunched component bodies, unlaunched shared variable, shared variable instance, and all terminal component instances.

Next, we define stable activity manager instances, denoted $A_{MI}(\vec{x})$. These instances include launching an application (component) and sending an intent to the launched instance; sending a new intent to a launched instance; and requests to return to the caller. These are the configurations where an atomic reduction step will stop. As we mentioned earlier, an activity instance can also generate an output to the instance's lock channel. We consider this output to be part of the configuration of the component instance, and, therefore, it is not included in the definitions of $A_{MI}(\vec{x})$. However, the execution of the activity manager does stop when such output is produced.

$$
\begin{array}{ll}
\textit{Stable activity manager instances} & A_{MI}(\vec{x}) ::= \\
& (aid, (c_{AI}, c_{nI}))[\,\mathsf{out}\; aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, O)\,] \\
\mid & (aid, c_{AI})[\mathsf{out}\; aid{\cdot}c_L(c_{AI})] \\
& {}'|'(aid, (c_{AI}, c_{nI}))[\mathsf{out}\; aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, O)] \\
\mid & (aid, (c_{AI}, c_{nI}))[\,\mathsf{out}\; c_{nI}(I)\,] \\
\mid & (aid, (kA, kC))[\\
& \mathsf{out}\; a_m(\mathsf{call}_E, kA, kC, O, aid, aid_{ce}, cid_{ce}, I)] \\
\mid & (aid(kA, kC))[\mathsf{out}\; a_m(\mathsf{return}, kA, kC, aid, \\
& \qquad\qquad aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)]
\end{array}
$$

We will write $T_{MI}(\Xi)$ to denote an instance of the label manager, which is the sub-process of $\mathsf{T_M}$ (Figure 9) after the input to the launch channel ($\mathsf{T_M} = !(\mathsf{in}\; t_t(\Xi).T_{MI}(\Xi))$). Here, Ξ is the label map of the label manager instance.

Definition 2. *A configuration S is a terminal configuration if all the application instances are terminal, and S does not contain any activity manager instances.*

Definition 3. *We say that a configuration of an application $AC(aid)$ is valid if $AC(aid) = App(aid) \mid Q \mid P_1 \mid \cdots \mid P_n$, where Q is a valid active instance of aid, and $\forall i \in [1, n]$, P_i is a terminal instance of aid.*

Definition 4. *We say that a configuration of an application P is a valid* projected *configuration if there exists Q such that Q can only contain the parallel composition of $App(aid)$ and $aid[c_{AI}[(c_{AI}, cid_i)[CP_i(aid, cid_i, c_{AI}, c_{sv})]]]$, where c_{AI} is the ID for the active instance in P, and $P \mid Q$ is a valid configuration for application aid.*

Each stable system configuration satisfies certain invariants. We define two relations, $\ll$ and $\ll_m$, between a static label and an effective label. The relation $\ll$ holds for every pair of static and effective labels of an application or single-instance component in a stable configuration. The relation $\ll_m$ holds for every pair of static and effective labels of a multi-instance component in a stable configuration. A multi-instance component can raise its instance label without affecting its static label. As we discussed earlier, an application or single-instance component has to raise its effective and static label at the same time. Therefore, the only time the effective label is not lower than or equal to the static label is when the static label is a floating label.

Definition 5. *$K \ll K'$ iff $K = (\_, \_, \delta)$ and $K' = (\_, \_, \delta)$ and one of the follow holds*

1. *$K^* \sqsubseteq K'^*$ and $K'^* \not\sqsubseteq K^*$ implies K is a floating label*
2. *$K^* \not\sqsubseteq K'^*$ and $K^* \uplus_d \delta \sqsubseteq K'^*$*

Definition 6. *$K \ll_m K'$ iff $K = (\_, \_, \delta)$ and $K' = (\_, \_, \delta)$ and one of the follow holds*

1. *$K^* \sqsubseteq K'^*$*
2. *$K^* \not\sqsubseteq K'^*$ and $K^* \uplus_d \delta \sqsubseteq K'^*$*

We define a stricter relation $K \ll_s K'$ ($K \ll_{ms} K'$), such that $K \ll_s K'$ ($K \ll_{ms} K'$) is true if and only if condition 1 in Definition 5 (Definition 6) holds. These two relations do not allow declassification.

We define a valid pair of a label map and an application configuration with respect to a set of free names, written *valid*$(\vec{x}, \Xi, AC(aid))$, as follows.

Definition 7. *valid$(\vec{x}, \Xi, AC(aid))$, if*

- $\mathsf{fn}(\Xi) \cup \mathsf{fn}(AC(aid)) \subseteq \vec{x}$
- *if $\Xi(aid) = \mathsf{U}(k)$, then $AC(aid)$ does not contain any active instances of aid*
- *if $\Xi(aid) = \mathsf{L}(c_{AI}, kA_s)$ or $\Xi(aid) = \mathsf{onEx}(c_{AI}, kA_s)$, then $\Xi(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(LC))$ and one of the following holds*
  1. *There is no active instance of aid yet, but there is an output to the launch channel of aid, and $kA_s \ll_s kA_d$; or*
  2. *All of the following hold*
     - *the active instance of aid is guarded by the label c_{AI}*
     - *$kA_s \ll kA_d$*
     - *if $\Xi(aid.cid) = \mathsf{S}(kC_s)$, then $AC(aid)$ does not contain any active instance of cid*
     - *if $\Xi(aid.cid) = \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_d)$, then $\Xi(c_{nI}) = \mathsf{DC}(c_{lock}, kC_d)$ and either (1) there is no active instance of cid yet, but there is an output to the create channel of cid, and $kC_s \ll_s kC_d$*
       or (2) $AC(aid)$ contains one active instance of cid guarded by c_{nI}, $c_{nI} \in LC$, and $kC_s \ll kC_d$
     - *if $\Xi(aid.cid) = \mathsf{M}(kC_s)$, then either*
       (1) $\Xi(c_{nI}) = \mathsf{DC}(c_{lock}, kC_d)$ and $AC(aid)$ may contain multiple active instances of cid, each guarded by a distinct channel c_{nI}, $c_{nI} \in LC$, and $kC_s \ll_m kC_d$; or
       (2) there are multiple outputs to the create channel of $aid.cid$, each has a label kC_d and a distinct channel name c_{nI} as the argument, and $c_{nI} \in LC$, and $kC_s \ll_{ms} kC_d$
     - *for all $c_{nI} \in LC$, c_{nI} is a new intent channel for either a single-instance or a multi-instance component instance (as described in the above two bullets)*

Now we define a stable system configuration *stb*(S) as follows:

Definition 8. *stb(S) iff*

$$
\begin{array}{l}
S = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi) \mid A_{MI1} \cdots \mid A_{MIk} \\
\qquad\qquad \mid AC(aid_1) \mid \cdots \mid AC(aid_n)
\end{array}
$$

where $AC(aid_i)$ is a (projected) valid configuration of an application, and $\forall i \in [1, n]$, valid$(\vec{x}, \Xi, AC(aid_i))$,

A stable configuration contains an activity manager, a label manager, a label manager instance, several stable activity manager instances, and valid application configurations. Note that $\vec{x}$ contains all the free names in the configuration.

F. Android-specific Transition Rules

We define all the Android-specific transitions rules in Figure 12.

Lemma F.1 (Progress of stable configurations)**.** *If stb(S), then either S is at a terminal state, or exists S' such that $S \xrightarrow{\alpha}_A S'$.*

Proof. By examining the definition of *stb*(S) □

Lemma F.2 (Preservation of stable configurations)**.** *If stb(S) and $S \xrightarrow{\alpha}_A S'$, then stb$(S')$.*

Proof. By examining the definition of *stb*(S) □

The above lemmas ensure the well-definedness of the transition relations of the Android-specific transitions.

$$\frac{S \xrightarrow{\alpha}_{\equiv} S' \qquad stb(S) \qquad stb(S')}{S \xrightarrow{\alpha}_{A} S'}$$

$S \xRightarrow{\tau}_{\equiv} S'$
$stb(S)$ $stb(S')$ the transitions in $\xRightarrow{\tau}$ are all internal transitions of the activity manager (instance) and label manager (instance) and S' is the first stable configuration
$$\frac{}{S \xrightarrow{\tau}_{A} S'}$$

$S \xRightarrow{\tau}_{\equiv} S'$
$stb(S)$ $stb(S')$ the transitions in $\xRightarrow{\tau}$ are all internal transitions of a read or a write to the shared variable of one instance and S' is the first stable configuration
$$\frac{}{S \xrightarrow{\tau}_{A} S'}$$

$$\frac{\Xi\langle c_{AI}\rangle = kA \qquad \Xi\langle c_{nI}\rangle = kC}{\begin{array}{l}\mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.(T_{MI}(\Xi) \mid P \\ \quad \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[\mathsf{out}\, a_m(\mathsf{req}, \vec{e})]]]]) \\ \xrightarrow{\tau}_{A} \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.(T_{MI}(\Xi) \mid P \\ \quad \mid aid[c_{AI}[(c_{AI}, cid)[(kA, kC)[\mathsf{out}\, a_m(\mathsf{req}, kA, kC, \vec{e})]]]])\end{array}}$$

$$\frac{\Xi\langle c_{AI}\rangle = kA \qquad \Xi\langle c_{nI}\rangle = kC}{\begin{array}{l}\mathsf{A_M} \mid \mathsf{T_M} \mid \nu\vec{x}.T_{MI}(\Xi) \mid Q \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[\mathsf{out}\, I(\mathsf{self}).P]]]] \\ \xrightarrow[A]{\mathsf{out}\, I((aid,(kA,kC)))} \\ \mathsf{A_M} \mid \mathsf{T_M} \mid \nu\vec{x}.T_{MI}(\Xi) \mid Q \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]]\end{array}}$$

Figure 12. Android-specific transition rules.

G. Process Projections

We define the projection functions for processes in Figures 13 and 14. The projection function $\mathsf{proj}(P, \Xi, aid_c, \kappa_L)$ removes from P any high processes (processes whose behavior should not affect the attacker) and dead code (processes guarded by channel names that do not exist in the label map). Figure 13 presents rules for processes that are guarded by label contexts that appear in application configurations. Figure 14 defines rules for processes with simplified label contexts. These label contexts come from the activity manager or the simplify function, which we introduce later. For nested label contexts, the only ones that are relevant for deciding the label of a process are the application ID on the outermost level, and the channel(s) on the innermost level. If a process's application ID is aid_c, then the component-level label is used; otherwise, the application-level label is used.

We say a process P contains fewer components than Q (written $P \leq Q$) if Q is the parallel composition of P and another process.

Definition 9. $P \leq Q \stackrel{def}{=} Q = P \mid Q_1$

H. Label Map Projection and Relations

We define projection functions on the label map in Figure 15. Function $\mathsf{projT}(\Xi; aid_c; \kappa_L)$ removes high mappings from the label map. Similarly to the process projection functions, we treat the mappings associated with an application ID that is not aid_c and those that are differently. We examine all mappings associated with one application together. If this application's ID is not aid_c and it is not launched yet, then the projection function keeps all the mappings when aid is mapped to a low label; otherwise the projection function removes all those mappings. If this application's ID is not aid_c and its running instance's channel ID is c_{AI}, then the projection keeps all the mappings when c_{AI} is mapped to a low label; when both aid and c_{AI} are mapped to high labels, the projection removes all of those mappings; when aid is mapped to a low label, but c_{AI} is mapped to a high label, the projection function only keeps the static labels for components, and restores aid_c to be unlaunched. The last case is the case for floating applications. A discrepancy in the mapping of aid and c_{nI} is only possible if aid's static label is floating and a high component launched this application. In a system without high components, such calls would not have existed; therefore, application aid remains unlaunched.

When the application's ID is aid_c, since our proof setup assumes that this particular application is launched, the label for aid_c and its corresponding instance channel c_{AI} remain the same after the projection. For this application, a separate projection function is

$\boxed{\mathsf{proj}(P, \Xi, aid_c, \kappa_L)}$

$$\frac{\Xi\langle c_{AI}\rangle = K \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, (c_{AI}, c_{nI}))[P], \Xi, aid_c, \kappa_L) = (aid, (c_{AI}, c_{nI}))[P]} \text{ SA-L1}$$

$$\frac{\Xi\langle c_{AI}\rangle = K \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, (c_{AI}, c_{nI}))[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SA-H1}$$

$$\frac{\Xi\langle c_{AI}\rangle = K \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, c_{AI})[P], \Xi, aid_c, \kappa_L) = (aid, c_{AI})[P]} \text{ SA-L2}$$

$$\frac{\Xi\langle c_{AI}\rangle = K \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, c_{AI})[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SA-H2}$$

$$\frac{P \neq l[P'] \qquad kA^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, (kA, kC))[P], \Xi, aid_c, \kappa_L) = (aid, (kA, kC))[P]} \text{ SA-L3}$$

$$\frac{P \neq l[P'] \qquad kA^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, (kA, kC))[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SA-H3}$$

$$\frac{\Xi\langle c_{nI}\rangle = K \qquad K^* \sqsubseteq \kappa_L}{\mathsf{proj}((aid, (c_{AI}, c_{nI}))[P]], \Xi, aid_c, \kappa_L) = (aid, (c_{AI}, c_{nI}))[P]} \text{ SC-L1}$$

$$\frac{\Xi\langle c_{nI}\rangle = K \qquad K^* \not\sqsubseteq \kappa_L}{\mathsf{proj}((aid, (c_{AI}, c_{nI}))[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SC-H1}$$

$$\frac{P \neq l[P'] \qquad kC^* \sqsubseteq \kappa_L}{\mathsf{proj}((aid, (kA, kC))[P], \Xi, aid_c, \kappa_L) = (aid, (kA, kC))[P]} \text{ SC-L2}$$

$$\frac{P \neq l[P'] \qquad kC^* \not\sqsubseteq \kappa_L}{\mathsf{proj}((aid, (kA, kC))[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SC-H2}$$

$$\frac{P \neq l[P'] \qquad kC^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, (kA, kC))[P], \Xi, aid_c, \kappa_L) = (aid, (kA, kC))[P]} \text{ SC-L3}$$

$$\frac{P \neq l[P'] \qquad kC^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}((aid, (kA, kC))[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SC-H3}$$

$$\frac{\begin{array}{c} c_{AI} \notin \mathsf{dom}(\Xi) \\ \text{or } aid \notin \mathsf{dom}(\Xi) \qquad \text{or } c_{nI} \notin \mathsf{dom}(\Xi)\end{array}}{\mathsf{proj}((aid, (c_{AI}, c_{nI}))[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ S-D1}$$

$$\frac{c_{AI} \notin \mathsf{dom}(\Xi) \qquad \text{or } aid \notin \mathsf{dom}(\Xi) \qquad aid \neq aid_c}{\mathsf{proj}((aid, c_{AI})[P], \Xi, aid_c, \kappa_L) = \mathbf{0}} \text{ SA-D2}$$

Figure 14. Projection of simplified processes.

$\boxed{\mathsf{proj}(P, \Xi, aid_c, \kappa_L)}$

$$\frac{P \neq l[P'] \qquad aid \mapsto \mathsf{U}(K) \in \Xi \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[P], \Xi, aid_c, \kappa_L) = aid[P]} \text{ A-L1}$$

$$\frac{P \neq l[P'] \qquad aid \mapsto \mathsf{U}(K) \in \Xi \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[P], \Xi, aid_c, \kappa_L) = 0} \text{ A-H1}$$

$$\frac{P \neq l[P'] \qquad aid \mapsto Q(c, K) \in \Xi \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[P], \Xi, aid_c, \kappa_L) = aid[P]} \text{ A-L2}$$

$$\frac{P \neq l[P'] \qquad aid \mapsto Q(c, K) \in \Xi \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[P], \Xi, aid_c, \kappa_L) = 0} \text{ A-H2}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \mapsto \mathsf{DA}(K, L) \in \Xi \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[c_{AI}[P]], \Xi, aid_c, \kappa_L) = aid[c_{AI}[P]]} \text{ A-L3}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \mapsto \mathsf{DA}(K, L) \in \Xi \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[c_{AI}[P]], \Xi, aid_c, \kappa_L) = 0} \text{ A-H3} \qquad \frac{P \neq l[P'] \qquad c_{AI} \notin \mathsf{dom}(\Xi)}{\mathsf{proj}(aid[c_{AI}[P]], \Xi, aid_c, \kappa_L) = 0} \text{ A-D1}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \mapsto \mathsf{DA}(K, L) \in \Xi \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[c_{AI}[(c_{AI}, cid)[P]]], \Xi, aid_c, \kappa_L) = aid[c_{AI}[(c_{AI}, cid)[P]]]} \text{ A-L4}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \mapsto \mathsf{DA}(K, L) \in \Xi \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[c_{AI}[(c_{AI}, cid)[P]]], \Xi, aid_c, \kappa_L) = 0} \text{ A-H4}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \notin \mathsf{dom}(\Xi) \qquad \text{or } cid \notin \mathsf{dom}(\Xi)}{\mathsf{proj}(aid[c_{AI}[(c_{AI}, cid)[P]]], \Xi, aid_c, \kappa_L) = 0} \text{ AC-D2}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \mapsto \mathsf{DA}(K, L) \in \Xi \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]], \Xi, aid_c, \kappa_L) = aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]]} \text{ A-L5}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \mapsto \mathsf{DA}(K, L) \in \Xi \qquad K^* \not\sqsubseteq \kappa_L \qquad aid \neq aid_c}{\mathsf{proj}(aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]], \Xi, aid_c, \kappa_L) = 0} \text{ A-H5}$$

$$\frac{P \neq l[P'] \qquad c_{AI} \notin \mathsf{dom}(\Xi) \qquad \text{or } cid \notin \mathsf{dom}(\Xi) \qquad \text{or } c_{nI} \notin \mathsf{dom}(\Xi)}{\mathsf{proj}(aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]], \Xi, aid_c, \kappa_L) = 0} \text{ AC-D3}$$

$$\frac{}{\mathsf{proj}(aid_c[c_{AI}[P]], \Xi, aid_c, \kappa_L) = aid_c[c_{AI}[P]]} \text{ C} \qquad \frac{P \neq l[P'] \qquad \Xi(aid_c.cid) = K \qquad K^* \sqsubseteq \kappa_L}{\mathsf{proj}(aid_c[c_{AI}[(c_{AI}, cid)[P]]], \Xi, aid_c, \kappa_L) = aid_c[c_{AI}[(c_{AI}, cid)[P]]]} \text{ C-L1}$$

$$\frac{P \neq l[P'] \qquad \Xi(aid_c.cid) = K \qquad K^* \not\sqsubseteq \kappa_L}{\mathsf{proj}(aid_c[c_{AI}[(c_{AI}, cid)[P]]], \Xi, aid_c, \kappa_L) = 0} \text{ C-H1}$$

$$\frac{P \neq l[P'] \qquad c_{nI} \mapsto \mathsf{DC}(c, K) \in \Xi \qquad K^* \sqsubseteq \kappa_L}{\mathsf{proj}(aid_c[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]], \Xi, aid_c, \kappa_L) = aid_c[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]]} \text{ C-L2}$$

$$\frac{P \neq l[P'] \qquad c_{nI} \mapsto \mathsf{DC}(c, K) \in \Xi \qquad K^* \not\sqsubseteq \kappa_L}{\mathsf{proj}(aid_c[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[P]]]], \Xi, aid_c, \kappa_L) = 0} \text{ C-H2}$$

$$\frac{}{\mathsf{proj}(P \mid Q, \Xi, aid_c, \kappa_L) = \mathsf{proj}(P, \Xi, aid_c, \kappa_L) \mid \mathsf{proj}(Q, \Xi, aid_c, \kappa_L)} \text{ PARR}$$

Figure 13. Projection of processes.

defined for the mappings for its components (instances). The function $\mathsf{projTC}(\Xi; aid_c; \kappa_L)$ removes high mappings related to components from the application with ID aid_c. In most cases, the projection preserves the mapping if the label is low, and removes it if it is high. The special case is when a single-instance component's static label is low, but the effective label for its instance channel c_{nI} is high. This is analogous to the case of floating application. Here, the single-instance component has a floating label, and therefore the projection removes the instance label and maps the component's static label to be single, unlaunched.

We also define two relations between two label maps: $\Xi \precsim^{\kappa_L}_{aid_c} \Xi'$ and $\Xi \precsim^{\kappa_L}_{\mathsf{comp}} \Xi'$. These two relations closely match the projection functions. Intuitively, Ξ is the label map for the system with high components, and Ξ' is the label map for the system without high components. These two relations force Ξ and Ξ' to agree on low mappings. Again, we need two relations because of the different treatment of labels based on whether they are related to application aid_c. The relation $\precsim^{\kappa_L}_{\mathsf{comp}}$ relates two label maps only containing mappings related to aid_c. Most cases are straightforward. The interesting cases are AL-Float and C-SL-Float, where the mapping of a floated high instance relates to an unlaunched low static map.

We prove several basic properties of the label map functions and projection functions.

Lemmas H.1 and H.2 state that a label map relates to the projected label map.

Lemma H.1.
If $\Xi = \Xi_0, \Xi_1$ *and* $\Xi_0 = aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL))$ *and* $\mathsf{projTC}(\Xi; aid_c; \kappa_L) = \Xi'$*, then* $\Xi' = \Xi'_0, \Xi'_1$ *where* $\Xi'_0 = aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL'))$*, and* $\Xi_1 \precsim^{\kappa_L}_{comp} \Xi'_1$

Proof. By induction on the derivation of $\mathsf{projTC}(\Xi; aid_c; \kappa_L)$. □

Lemma H.2. $\Xi \precsim^{\kappa_L}_{aid_c} \mathsf{projT}(\Xi; aid_c; \kappa_L)$

Proof. By induction on the derivation of $\mathsf{projT}(\Xi; aid_c; \kappa_L)$. □

Lemma H.3. $\precsim^{\kappa_L}_{aid_c}$ *is transitive.*

Proof. By induction on the first $\precsim^{\kappa_L}_{aid_c}$ relation. □

Lemma H.4. *If* $\Xi_1 \prec \Xi_2$*, where* $\prec$ *is* $\precsim^{\kappa_L}_{aid_c}$ *or* $\precsim^{\kappa_L}_{\mathsf{comp}}$ *then for all* x *such that* $x \in \mathsf{dom}(\Xi_1)$*, and* $(\Xi_2\langle x\rangle)^* \sqsubseteq \kappa_L$*,* $\Xi_2\langle x\rangle = \Xi_1\langle x\rangle$*.*

Proof. By induction on the $\prec$ relation. □

Lemma H.5. *If* $\Xi_P \precsim^{\kappa_L}_{aid_c} \Xi_Q$*, then* $\mathsf{proj}(P, \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(P, \kappa_L, aid_c, \Xi_Q)$

Proof. By induction on the structure of P. Use Lemma H.4 in the base case. That the projection on the left exists implies the projection on the right also exists. □

We prove that the we can obtain a stable configuration by projecting the label map Ξ_P, and then projecting the process with the new label map.

Lemma H.6. *If* S *is a stable configuration,*
$S = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi_P) \mid P$*,* $\Xi'_P = \mathsf{projT}(\Xi_P; aid_c; \kappa_L)$
$S' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}'.T_{MI}(\Xi'_P).\mathsf{proj}(P, \kappa_L, aid_c, \Xi'_P)$ *is a stable configuration where* $\vec{x}' \subseteq \vec{x}$*.*

Proof. By examining the definition of *stb*(S). □

I. Label Updates

Operations such as calls, raising labels, declassification and exiting will cause the label map to be updated. We classify these label updates either as a *high* update or a *low* update. The judgment $aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{LO}$ states that an update in Ξ is a low update with regard to aid_c, κ_L and a label map Ξ_1. Here, Ξ_1 is the current label map of the configuration. Similarly, the judgment $aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{HI}$ states that Ξ is a high update.

Figures 17 and 18 present rules for valid low label updates. Rules in Figure 17 apply to updates related to an application whose ID is not aid_c; while rules in Figure 18 apply to updates to an application of ID aid_c. Deciding whether an update is low or high is based on the effective label of the instance of the updated application (component), which is the last premise of each rule. When the $\sqsubseteq$ in that premise is replaced by $\not\sqsubseteq$, we obtain rules for valid *high* updates, except for updates resulting from declassification, where the update is a high update if the declassified label is high. We show these two rules in Figure 19.

The main observation is that when the update is a high update, the changes to the system configuration only affect high components and are not observable by the attacker.

The following lemmas concern the effects of low (high) label updates on the label map relation and the process projection functions. When an update is a low update, the update preserves the label map relation, when applied to both maps (Lemma I.1). When the update is a high update, updating the label map on the left of that relation preserves that relation (Lemma I.2).

Lemma I.1.

1. *If* $\Xi_1 \precsim^{\kappa_L}_{\mathsf{comp}} \Xi_2$ *and* $aid_c; \kappa_L; (\Xi_0, \Xi_1) \vdash \Xi : \mathsf{LO}$ *where* $\Xi_0 = aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL))$*, then* $\Xi_1 \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_2 \uplus \Xi$
2. *If* $\Xi_1 \precsim^{\kappa_L}_{aid_c} \Xi_2$*,* $aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{LO}$ *then* $\Xi_1 \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_2 \uplus \Xi$

Proof. By examining all valid updates. □

Lemma I.2.

1. *If* $\Xi_1 \precsim^{\kappa_L}_{\mathsf{comp}} \Xi_2$ *and* $aid_c; \kappa_L; (\Xi_0, \Xi_1) \vdash \Xi : \mathsf{HI}$ *where* $\Xi_0 = aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL))$*, then* $\Xi_1 \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_2$
2. *If* $\Xi_1 \precsim^{\kappa_L}_{aid_c} \Xi_2$*,* $aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{HI}$ *then* $\Xi_1 \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_2$

Proof. By examining all valid updates. □

Lemma I.3 states that projecting a process using the label map of Ξ updated with high updates, results in a process containing fewer components than projecting the same process with Ξ. This is because high updates to a label map result in more high mappings in the map, and, therefore, more processes will be removed by the projection using the updated map than the original one.

Lemma I.3. *If* $aid_c; \kappa_L; (\Xi_0, \Xi_1) \vdash \Xi : \mathsf{HI}$ *and* P *does not contain processes guarded by new names (*c_{AI} *or* c_{nI}*) in* Ξ*, then* $\mathsf{proj}(P, \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(P, \kappa_L, aid_c, \Xi_P)$

Proof. By induction on the structure of P. □

The next lemma states that low updates to a label map preserve the less-than relation of two projected processes.

Lemma I.4. *If* $\mathsf{proj}(P, \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(Q, \kappa_L, aid_c, \Xi_Q)$*,* $\Xi_1 \precsim^{\kappa_L}_{aid_c} \Xi_2$*,* P *does not contain processes guarded by new names (*c_{AI} *or* c_{nI}*) in* Ξ*, and* $aid_c; \kappa_L; (\Xi_0, \Xi_1) \vdash \Xi : \mathsf{LO}$ *then* $\mathsf{proj}(P, \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(Q, \kappa_L, aid_c, \Xi_Q \uplus \Xi)$

$\boxed{\mathsf{projT}(\Xi;\, aid_c;\, \kappa_L)}$

$$\frac{}{\mathsf{projT}(\cdot;\, aid_c;\, \kappa_L) = \cdot}\ \text{A-EMPTY}$$

$$\frac{\mathsf{projT}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad K^* \sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad aid \neq aid_c}{\mathsf{projT}(\Xi,\, aid \mapsto \mathsf{U}(K), \Xi_1;\, aid_c;\, \kappa_L) = \Xi',\, aid \mapsto \mathsf{U}(K), \Xi_1}\ \text{A-UL}$$

$$\frac{\mathsf{projT}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad K^* \not\sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad aid \neq aid_c}{\mathsf{projT}(\Xi,\, aid \mapsto \mathsf{U}(K), \Xi_1;\, aid_c;\, \kappa_L) = \Xi'}\ \text{A-UH}$$

$$\frac{\mathsf{projT}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad Q = \mathsf{L} \text{ or } \mathsf{onEx} \qquad kA_d^* \sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad aid \neq aid_c}{\begin{array}{c}\mathsf{projT}(\Xi,\, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1;\, aid_1;\, \kappa_L) \\ = \Xi',\, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1\end{array}}\ \text{A-LL}$$

$$\frac{\mathsf{projT}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad kA_s^* \not\sqsubseteq \kappa_L \qquad kA_d^* \not\sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad aid \neq aid_c}{\mathsf{projT}(\Xi,\, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1;\, aid_1;\, \kappa_L) = \Xi'}\ \text{A-LH}$$

$$\frac{\begin{array}{c}\mathsf{projT}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \\ kA_s^* \sqsubseteq \kappa_L \qquad kA_d^* \not\sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad \mathsf{dom}(\Xi_2) = CL \qquad aid \neq aid_c\end{array}}{\begin{array}{l}\mathsf{projT}(\Xi,\, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1, \Xi_2;\, aid_c\kappa_L) \\ \qquad = \Xi',\, aid \mapsto U(kA_s), \Xi_1\end{array}}\ \text{A-FH}$$

$$\frac{\mathsf{projTC}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad \mathsf{projT}(\Xi_1;\, aid_c;\, \kappa_L) = \Xi_1'}{\mathsf{projT}(\Xi, \Xi_1;\, aid_c;\, \kappa_L) = \Xi', \Xi_1'}\ \text{A-C}$$

$\boxed{\mathsf{projTC}(\Xi;\, aid_c;\, \kappa_L)}$

$$\frac{}{\mathsf{projTC}(aid_c \mapsto \mathsf{L}(c_{AI}, k), c_{AI} \mapsto x;\, aid_c;\, \kappa_L) = aid_c \mapsto \mathsf{L}(c_{AI}, k), c_{AI} \mapsto x}\ \text{C-BASE}$$

$$\frac{\mathsf{projTC}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad K^* \sqsubseteq \kappa_L}{\mathsf{projTC}(\Xi,\, aid_c{\cdot}cid \mapsto \mathsf{S}(K);\, aid_c;\, \kappa_L) = \Xi',\, aid_c{\cdot}cid \mapsto \mathsf{S}(K)}\ \text{C-SL} \qquad \frac{\mathsf{projTC}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad K^* \not\sqsubseteq \kappa_L}{\mathsf{projTC}(\Xi,\, aid_c{\cdot}cid \mapsto \mathsf{S}(K);\, aid_c;\, \kappa_L) = \Xi'}\ \text{C-SH}$$

$$\frac{\mathsf{projTC}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad K^* \sqsubseteq \kappa_L}{\mathsf{projTC}(\Xi,\, aid_c{\cdot}cid \mapsto \mathsf{M}(K);\, aid_c;\, \kappa_L) = \Xi',\, aid_c{\cdot}cid \mapsto \mathsf{M}(K)}\ \text{C-ML} \qquad \frac{\mathsf{projTC}(\Xi;\, aid_c;\, \kappa_L) = \Xi' \qquad K^* \not\sqsubseteq \kappa_L}{\mathsf{projTC}(\Xi,\, aid_c{\cdot}cid \mapsto \mathsf{M}(K);\, aid_c;\, \kappa_L) = \Xi'}\ \text{C-MH}$$

$$\frac{kC_s^* \not\sqsubseteq \kappa_L \qquad kC_d^* \not\sqsubseteq \kappa_L \qquad \mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI}));\, aid_c;\, \kappa_L) = \Xi'}{\mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)),\, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d);\, aid_c;\, \kappa_L) = \Xi'}\ \text{C-SLH}$$

$$\frac{KC_s^* \sqsubseteq \kappa_L \qquad KC_d^* \not\sqsubseteq \kappa_L \qquad \mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI}));\, aid_c;\, \kappa_L) = \Xi'}{\begin{array}{l}\mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)),\, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d);\, aid_c;\, \kappa_L) \\ \qquad = \Xi',\, aid_c{\cdot}cid \mapsto \mathsf{S}(kC_s)\end{array}}\ \text{C-FH}$$

$$\frac{kC_d^* \sqsubseteq \kappa_L \qquad \mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL));\, aid_c;\, \kappa_L) = \Xi'}{\begin{array}{l}\mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)),\, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d);\, aid_c;\, \kappa_L) \\ \qquad = \Xi',\, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d)\end{array}}\ \text{C-SLL}$$

$$\frac{K_d^* \sqsubseteq \kappa_L \qquad \mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL));\, aid_c;\, \kappa_L) = \Xi'}{\mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)), c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d);\, aid_c;\, \kappa_L) = \Xi', c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d)}\ \text{C-CL}$$

$$\frac{K_d^* \not\sqsubseteq \kappa_L \qquad \mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI}));\, aid_c;\, \kappa_L) = \Xi'}{\mathsf{projTC}(\Xi,\, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)), c_{nI} \mapsto \mathsf{DC}(c_{lock}, kC_d);\, aid_c;\, \kappa_L) = \Xi'}\ \text{C-CH}$$

Figure 15. Projection of label map for a specific application.

$\boxed{\Xi \preceq^{\kappa_L}_{aid_c} \Xi'}$

$$\frac{}{\cdot \preceq^{\kappa_L}_{aid_c} \Xi}\ \text{A-EMPTY} \qquad \frac{\mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad \Xi \preceq^{\kappa_L}_{aid_c} \Xi' \qquad K^* \sqsubseteq \kappa_L \qquad aid \neq aid_c}{\Xi, aid \mapsto \mathsf{U}(K), \Xi_1 \preceq^{\kappa_L}_{aid_c} \Xi', aid \mapsto \mathsf{U}(K), \Xi_1}\ \text{AU-LOW}$$

$$\frac{\Xi \preceq^{\kappa_L}_{aid_c} \Xi' \qquad K^* \not\sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \qquad aid \neq aid_c}{\Xi, aid \mapsto \mathsf{U}(K), \Xi_1 \preceq^{\kappa_L}_{aid_c} \Xi'}\ \text{AU-HIGH}$$

$$\frac{\Xi \preceq^{\kappa_L}_{aid_c} \Xi' \qquad Q = \mathsf{L} \text{ or } \mathsf{onEx} \qquad kA^*_d \sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \cup CL \qquad aid \neq aid_c}{\Xi, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1 \preceq^{\kappa_L}_{aid_c} \Xi', aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)), \Xi_1}\ \text{AL-LOW}$$

$$\frac{\Xi \preceq^{\kappa_L}_{aid_c} \Xi' \qquad kA^*_s \not\sqsubseteq \kappa_L \qquad kA^*_d \not\sqsubseteq \kappa_L \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \cup CL \qquad aid \neq aid_c}{\Xi, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1 \preceq^{\kappa_L}_{aid_c} \Xi'}\ \text{AL-HIGH}$$

$$\frac{\begin{array}{l}\Xi \preceq^{\kappa_L}_{aid_c} \Xi' \qquad \mathsf{dom}(\Xi_1) = \{aid{\cdot}cid_1, \cdots, aid{\cdot}cid_k\} \\ \mathsf{dom}(\Xi_2) = CL \qquad kA^*_s \sqsubseteq \kappa_L \qquad kA^*_d \not\sqsubseteq \kappa_L \qquad Q = \mathsf{L} \text{ or } \mathsf{onEx} \qquad aid \neq aid_c\end{array}}{(\Xi, aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(KA_d, \mathsf{LC}(CL)), \Xi_1, \Xi_2 \preceq^{\kappa_L}_{aid_c} \Xi', aid \mapsto \mathsf{U}(kA_s), \Xi_1}\ \text{AL-FLOAT}$$

$$\frac{\Xi \preceq^{\kappa_L}_{aid_c} \Xi' \qquad \Xi_1 \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'_1 \qquad aid \neq aid_c}{\Xi, aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL)), \Xi_1 \preceq^{\kappa_L}_{aid_c} \Xi' aid_c \mapsto \mathsf{L}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL')), \Xi'_1}\ \text{A-COMP}$$

$\boxed{\Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}$

$$\frac{}{\cdot \preceq^{\kappa_L}_{\mathsf{comp}} \Xi}\ \text{C-EMPTY} \qquad \frac{\Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi' \qquad K^* \sqsubseteq \kappa_L}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{S}(K) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi', aid_c{\cdot}cid \mapsto \mathsf{S}(K)}\ \text{C-SU-LOW}$$

$$\frac{\Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi' \qquad K^* \sqsubseteq \kappa_L}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{M}(K) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi', aid_c{\cdot}cid \mapsto \mathsf{M}(K)}\ \text{C-M-LOW}$$

$$\frac{K^*_d \sqsubseteq \kappa_L \qquad \Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, K_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi', aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, K_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d)}\ \text{C-SL-LOW}$$

$$\frac{K^*_d \sqsubseteq \kappa_L \qquad \Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}{\Xi, c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi', c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d)}\ \text{C-NI-LOW} \qquad \frac{\Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi' \qquad K^* \not\sqsubseteq \kappa_L}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{S}(K) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}\ \text{C-SU-HIGH}$$

$$\frac{K^*_s \sqsubseteq \kappa_L \qquad K^*_d \not\sqsubseteq \kappa_L \qquad \Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, K_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d) \preceq^{\kappa_L}_{\mathsf{comp}} \qquad \Xi', aid_c{\cdot}cid \mapsto \mathsf{S}(K_s)}\ \text{C-SL-FLOAT}$$

$$\frac{\Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi' \qquad K^* \not\sqsubseteq \kappa_L}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{M}(K) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}\ \text{C-M-HIGH} \qquad \frac{K^*_s \not\sqsubseteq \kappa_L \qquad K^*_d \not\sqsubseteq \kappa_L \qquad \Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}{\Xi, aid_c{\cdot}cid \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, K_s), c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}\ \text{C-SL-HIGH}$$

$$\frac{K^*_d \not\sqsubseteq \kappa_L \qquad \Xi \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}{\Xi, c_{nI} \mapsto \mathsf{DC}(c_{lock}, K_d) \preceq^{\kappa_L}_{\mathsf{comp}} \Xi'}\ \text{C-NI-HIGH}$$

Figure 16. Relation between label maps.

$\boxed{aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{LO}}$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{onEx}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI})), \backslash c_{nI} : \mathsf{LO}} \text{ A-onExitA-M}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(\{c_{nI}\})) \qquad kA_d^* \sqsubseteq \kappa_L}{aid, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{U}(kA_s), \backslash c_{nI}, \backslash c_{AI} : \mathsf{LO}} \text{ A-ExitA-M1}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{onEx}(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(\{c_{nI}\})) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{U}(kA_s), \backslash c_{nI}, \backslash c_{AI} : \mathsf{LO}} \text{ A-ExitA-M2}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \Xi_1(aid{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{onEx}(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI})), aid{\cdot}cid \mapsto \mathsf{S}(kC_s), \backslash c_{nI} : \mathsf{LO}} \text{ A-onExitA-S}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(aid{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(\{c_{nI}\})) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{U}(kA_s), aid{\cdot}cid \mapsto \mathsf{S}(kC_s), \backslash c_{nI} : \mathsf{LO}} \text{ A-ExitA-S1}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{onEx}(c_{AI}, kA_s) \qquad \Xi_1(aid{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(\{c_{nI}\})) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{U}(kA_s), aid{\cdot}cid \mapsto \mathsf{S}(kC_s), \backslash c_{nI} : \mathsf{LO}} \text{ A-ExitA-S2}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash, c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI})), \backslash c_{nI} : \mathsf{LO}} \text{ A-ExitC-M}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \Xi_1(aid{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI})), aid{\cdot}cid \mapsto \mathsf{S}(kC_s), \backslash c_{nI} : \mathsf{LO}} \text{ A-ExitC-S}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = Q(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \delta \subseteq dL(kA_d) \qquad (kA_d)^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d \uplus_d \delta, \mathsf{LC}(CL)) : \mathsf{LO}} \text{ A-DeclassifyA}$$

$$\frac{\begin{array}{c} aid \neq aid_c \\ \Xi_1(aid) = Q(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad \delta \subseteq dL(kC_d) \qquad (kA_d)^* \sqsubseteq \kappa_L \end{array}}{aid_c, \kappa_L, \Xi_1 \vdash c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_d \delta) : \mathsf{LO}} \text{ A-DeclassifyC}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = Q(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto Q(c_{AI}, kA_s \uplus_{rz} (\sigma, \iota)), c_{AI} \mapsto \mathsf{DA}(kA_d \uplus_{rz} (\sigma, \iota), \mathsf{LC}(CL)) : \mathsf{LO}} \text{ A-RaiseA}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid{\cdot}cid \mapsto \mathsf{SL}(kC_s \uplus_{rz} (\sigma, \iota), c_{nI}), c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_{rz} (\sigma, \iota)) : \mathsf{LO}} \text{ A-RaiseC-S}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid{\cdot}cid) = \mathsf{M}(kC_s) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid{\cdot}cid \mapsto \mathsf{M}(kC_s), c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_{rz} (\sigma, \iota)) : \mathsf{LO}} \text{ A-RaiseC-M}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid{\cdot}cid) = \mathsf{M}(kC_s) \qquad \Xi_1(aid) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash c_{nI} \mapsto \mathsf{DC}(c, kC_s \lhd K), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \cup \{c_{nI}\})) : \mathsf{LO}} \text{ A-Call-C-M}$$

$$\frac{aid \neq aid_c \qquad \Xi_1(aid) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(aid{\cdot}cid) = \mathsf{S}(kC_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad kA_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid{\cdot}cid \mapsto \mathsf{SL}(kC_s, c_{nI}), c_{nI} \mapsto \mathsf{DC}(c, kC_s \lhd K), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \cup \{c_{nI}\})) : \mathsf{LO}} \text{ A-Call-C-S}$$

$$\frac{\begin{array}{c} aid \neq aid_c \\ \Xi_1(aid{\cdot}cid) = \mathsf{M}(kC_s) \qquad \Xi_1(aid) = \mathsf{U}(kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad K^- \sqsubseteq {kA_s}^- \qquad kA_s \lhd K^* \sqsubseteq \kappa_L \end{array}}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{L}(c_{AI}, kA_s), c_{nI} \mapsto \mathsf{DC}(c, kC_s \lhd K), c_{AI} \mapsto \mathsf{DA}(kA_s \lhd K, \mathsf{LC}(\{c_{nI}\})) : \mathsf{LO}} \text{ A-Call-AC-M}$$

$$\frac{\begin{array}{c} aid \neq aid_c \\ \Xi_1(aid) = \mathsf{U}(kA_s) \qquad \Xi_1(aid{\cdot}cid) = \mathsf{S}(kC_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad K^- \sqsubseteq {kA_s}^- \qquad kA_s \lhd K^* \sqsubseteq \kappa_L \end{array}}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto \mathsf{L}(c_{AI}, kA_s), aid{\cdot}cid \mapsto \mathsf{SL}(kC_s, c_{nI}), c_{nI} \mapsto \mathsf{DC}(c, kC_s \lhd K), c_{AI} \mapsto \mathsf{DA}(kA_s \lhd K, \mathsf{LC}(\{c_{nI}\})) : \mathsf{LO}} \text{ A-Call-AC-S}$$

Figure 17. Valid low lable updates (part 1 of 2).

$\boxed{aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{LO}}$

$$\frac{\Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad kC_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash, c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI})), \backslash c_{nI} : \mathsf{LO}} \text{ C-ExitC-M}$$

$$\frac{\Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \Xi_1(aid_c{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad kC_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \backslash c_{nI})), aid_c{\cdot}cid \mapsto \mathsf{S}(kC_s), \backslash c_{nI} : \mathsf{LO}} \text{ C-ExitC-S}$$

$$\frac{\Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad \delta \subseteq dL(kC_d) \qquad (kC_d)^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_d \delta) : \mathsf{LO}} \text{ C-DeclassifyC}$$

$$\frac{\Xi_1(aid_c{\cdot}cid) = \mathsf{SL}(kC_s, c_{nI}) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad kC_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid_c{\cdot}cid \mapsto \mathsf{SL}(kC_s \uplus_{rz} (\sigma, \iota), c_{nI}), c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_{rz} (\sigma, \iota)) : \mathsf{LO}} \text{ C-RaiseC-S}$$

$$\frac{\Xi_1(aid_c{\cdot}cid) = \mathsf{M}(kC_s) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad kC_d^* \sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash aid_c{\cdot}cid \mapsto \mathsf{M}(kC_s), c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_{rz} (\sigma, \iota)) : \mathsf{LO}} \text{ C-RaiseC-M}$$

$$\frac{\begin{array}{c}\Xi_1(aid_c{\cdot}cid) = \mathsf{M}(kC_s) \qquad \Xi_1(aid_c) = \mathsf{L}(c_{AI}, kA_s) \\ \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad K^- \sqsubseteq {kC_s}^- \qquad kC_s \lhd K^* \sqsubseteq \kappa_L\end{array}}{aid_c, \kappa_L, \Xi_1 \vdash c_{nI} \mapsto \mathsf{DC}(c, kC_s \lhd K), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \cup \{c_{nI}\})) : \mathsf{LO}} \text{ C-Call-C-M}$$

$$\frac{\begin{array}{c}\Xi_1(aid_c) = \mathsf{L}(c_{AI}, kA_s) \qquad \Xi_1(aid_c{\cdot}cid) = \mathsf{S}(kC_s) \\ \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad K^- \sqsubseteq {kC_s}^- \qquad kC_s \lhd K^* \sqsubseteq \kappa_L\end{array}}{aid_c, \kappa_L, \Xi_1 \vdash aid_c{\cdot}cid \mapsto \mathsf{SL}(kC_s, c_{nI}), c_{nI} \mapsto \mathsf{DC}(c, kC_s \lhd K), c_{AI} \mapsto \mathsf{DA}(kA_d, \mathsf{LC}(CL \cup \{c_{nI}\})) : \mathsf{LO}} \text{ C-Call-C-S}$$

Figure 18. Valid low lable updates (part 2 of 2).

$\boxed{aid_c, \kappa_L, \Xi_1 \vdash \Xi : \mathsf{HI}}$

$$\frac{\begin{array}{c}aid \neq aid_c \\ \Xi_1(aid) = Q(c_{AI}, kA_s) \qquad \Xi_1(c_{AI}) = \mathsf{DA}(kA_d, \mathsf{LC}(CL)) \qquad \delta \subseteq dL(kA_d) \qquad (kA_d \uplus_d \delta)^* \not\sqsubseteq \kappa_L\end{array}}{aid_c, \kappa_L, \Xi_1 \vdash aid \mapsto Q(c_{AI}, kA_s), c_{AI} \mapsto \mathsf{DA}(kA_d \uplus_d \delta, \mathsf{LC}(CL)) : \mathsf{HI}} \text{ A-DeclassifyA-HI}$$

$$\frac{\Xi_1(c_{nI}) = \mathsf{DC}(c, kC_d) \qquad \delta \subseteq dL(kC_d) \qquad (kC_d \uplus_d \delta)^* \not\sqsubseteq \kappa_L}{aid_c, \kappa_L, \Xi_1 \vdash c_{nI} \mapsto \mathsf{DC}(c, kC_d \uplus_d \delta) : \mathsf{HI}} \text{ C-DeclassifyC-HI}$$

Figure 19. Selected rules for valid high label updates.

Proof. By induction on the structure of P. □

J. Proof of Noninterference

Recall that the core of the noninterference proof is to find a simulation relation between the system with high components and the system without high components (Section 5). We define a relation $\mathcal{R}_{(aid_c, \kappa_L)}$ below, and show that it is a simulation relation (Lemma J.5).

We first define a function $\mathsf{sp}(P)$ to simplify a labeled process wrapped in adjacent nested labels to a labeled process wrapped by a pair of its application ID and the innermost label:
$\mathsf{sp}(aid[l_1[l_2 \cdots l_k[P]]]) = (aid, l_k)[P]$ or $aid[P]$,
where $P \neq l[P']$.

Definition 10. *$P \mathcal{R}_{(aid_c, \kappa_L)} Q$ iff*

- *P and Q are stable configurations,*
- $P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi_P) \mid P_0$,
- $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}'.T_{MI}(\Xi_Q) \mid Q_0$,
- $\Xi_P \precsim^{\kappa_L}_{aid_c} \Xi_Q$
- $\mathsf{proj}(\mathsf{sp}(P_0), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_P)$

We first prove several key lemmas that are used in the proof of Lemma J.5. These lemmas identify conditions on the label map updates and the system configuration P to ensure the simulation relation between the system with high components (P) and the related system without high components (Q) holds.

The following lemma states that if the label map does not change and P does not generate any new low components when P takes a step to P', then P' relates to Q.

Lemma J.1. *If*

1. $P \mathcal{R}_{(aid_c, \kappa_L)} Q$, $P \xrightarrow{\tau}_A P'$,
2. $P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi_P) \mid P_0$,
3. $P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}'.T_{MI}(\Xi_P) \mid P_0'$,
4. $\mathsf{proj}(\mathsf{sp}(P_0'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(P_0), \kappa_L, aid_c, \Xi_P)$

then $P' \mathcal{R}_{(aid_c, \kappa_L)} Q$

Proof. By $P \mathcal{R}_{(aid_c, \kappa_L)} Q$, (1)-(3) hold
(1) $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.T_{MI}(\Xi_Q) \mid Q_0$,
(2) $\Xi_P \precsim^{\kappa_L}_{aid_c} \Xi_Q$
(3) $\mathsf{proj}(\mathsf{sp}(P_0), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_Q)$
By (3), assumption 4, and $\leq$ is transitive

(4) $\mathsf{proj}(\mathsf{sp}(P_0'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_Q)$
By (1) (2) and (4), $P' \mathcal{R}_{(aid_c,\kappa_L)} Q$ □

Lemma J.2 states that if the label map does not change and the components that reduce in P are low components when P steps to P', then Q can take the same step as P to Q', and P' and Q' relate.

Lemma J.2. *If*

- $P \mathcal{R}_{(aid_c,\kappa_L)} Q$,
- $P \xrightarrow{\tau}_A P'$,
- $P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi_P) \mid P_1 \mid P_2$,
- $P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}'.T_{MI}(\Xi_P) \mid P_1 \mid P_2'$,
- $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.T_{MI}(\Xi_Q) \mid Q_1 \mid Q_2$,
- $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(Q_2) = \mathsf{sp}(P_2)$
- *Q takes the same step as P to Q'*

then $P' \mathcal{R}_{(aid_c,\kappa_L)} Q'$

Proof. By $P \mathcal{R}_{(aid_c,\kappa_L)} Q$, (1) and (2) hold
(1) $\Xi_P \precsim^{\kappa_L}_{aid_c} \Xi_Q$
(2) $\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_1), \kappa_L, aid_c, \Xi_Q)$
Because Q take the same step as P, so
(3) $Q' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}'.T_{MI}(\Xi_Q) \mid Q_1 \mid Q_2'$
By $\mathsf{sp}(P_2) = \mathsf{sp}(Q_2)$ and (3),
(4) $\mathsf{sp}(P_2') = \mathsf{sp}(Q_2')$
By Lemma H.5 on (1) and (4)
(5) $\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_2'), \kappa_L, aid_c, \Xi_Q)$
By (2) and (4),
(6) $\mathsf{sp}(\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P))$
$\leq \mathsf{proj}(\mathsf{sp}(Q_1 \mid Q_2'), \kappa_L, aid_c, \Xi_Q)$
By (1), (3) and (6)
$P' \mathcal{R}_{(aid_c,\kappa_L)} Q'$ □

If the label map changes when P steps to P', but the update preserves the label map relation between P' and Q, and P' does not have more low components than P according to the new label map, then P' and Q relate.

Lemma J.3. *If*

1. $P \mathcal{R}_{(aid_c,\kappa_L)} Q$,
2. $P \xrightarrow{\tau}_A P'$,
3. $P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi_P) \mid P_0$,
4. $P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}'.T_{MI}(\Xi_P') \mid P_0'$,
5. $\Xi_P' \precsim^{\kappa_L}_{aid_c} \Xi_Q$
6. $\mathsf{proj}(\mathsf{sp}(P_0'), \kappa_L, aid_c, \Xi_P') \leq \mathsf{proj}(\mathsf{sp}(P_0), \kappa_L, aid_c, \Xi_P)$

then $P' \mathcal{R}_{(aid_c,\kappa_L)} Q$

Proof. By $P \mathcal{R}_{(aid_c,\kappa_L)} Q$, (1)-(2) hold
(1) $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.T_{MI}(\Xi_Q) \mid Q_0$,
(2) $\mathsf{proj}(\mathsf{sp}(P_0), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_Q)$
By (2), assumption 6, and $\leq$ is transitive
(3) $\mathsf{proj}(\mathsf{sp}(P_0'), \kappa_L, aid_c, \Xi_P') \leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_Q)$
By (3) and assumption 5, $P' \mathcal{R}_{(aid_c,\kappa_L)} Q$ □

Finally, when the update to the label map is a low update, and the components that reduce in P exist in Q as well, then Q can take the same step as P and P' and Q' relate.

Lemma J.4. *If*

1. $P \mathcal{R}_{(aid_c,\kappa_L)} Q$,
2. $P \xrightarrow{\tau}_A P'$,
3. $P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}.T_{MI}(\Xi_P) \mid P_1 \mid P_2$,
4. $P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{x}'.T_{MI}(\Xi_P \uplus \Xi) \mid P_1 \mid P_2'$,
5. $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.T_{MI}(\Xi_Q) \mid Q_1 \mid Q_2$,
6. $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(Q_2) = \mathsf{sp}(P_2)$
7. *Q takes the same step as P to Q'*
8. $aid_c, \kappa_L, \Xi_P \vdash \Xi : \mathsf{LO}$

then $P' \mathcal{R}_{(aid_c,\kappa_L)} Q'$

Proof. By $P \mathcal{R}_{(aid_c,\kappa_L)} Q$, (1) and (2) hold
(1) $\Xi_P \precsim^{\kappa_L}_{aid_c} \Xi_Q$
(2) $\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_1), \kappa_L, aid_c, \Xi_Q)$
Because Q take the same step as P,
(3) $Q' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}'.T_{MI}(\Xi_Q \uplus \Xi) \mid Q_1 \mid Q_2'$
By $\mathsf{sp}(P_2) \leq \mathsf{sp}(Q_2)$,
(4) $\mathsf{sp}(P_2') = \mathsf{sp}(Q_2')$
by Lemma I.1 on (1) and assumption 8,
(5) $\Xi_P \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_Q \uplus \Xi$
By Lemma H.5 on (4) and (5)
(6) $\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(Q_2'), \kappa_L, aid_c, \Xi_Q \uplus \Xi)$
By Lemma I.4 on (1), (2) and assumption 8
(7) $\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(Q_1), \kappa_L, aid_c, \Xi_Q \uplus \Xi)$
By (6) and (7),
(8) $\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(Q_1 \mid Q_2'), \kappa_L, aid_c, \Xi_Q \uplus \Xi)$
By (3), (5) and (8), $P' \mathcal{R}_{(aid_c,\kappa_L)} Q'$ □

Now we prove the main lemma, which states that $\mathcal{R}_{(aid_c,\kappa_L)}$ is a simulation relation.

Lemma J.5. *$\mathcal{R}_{(aid_c,\kappa_L)}$ is a Γ-simulation relation.*

Proof. We need to prove two conditions: (1) τ transition preserves $\mathcal{R}_{(aid_c,\kappa_L)}$, and (2) if $P \mathcal{R}_{(aid_c,\kappa_L)} Q$, $P \xrightarrow{\mathsf{out}\, b(aid,(K_A,K_C))} P'$ and $(aid, (K_A^*, K_C^*)) \sqsubseteq_{aid_c} \kappa_L$, then $Q \xRightarrow{\mathsf{out}\, b(aid,(K_A,K_C))} Q'$, and $P' \mathcal{R}_{(aid_c,\kappa_L)} Q'$.

By the definition of $P \mathcal{R}_{(aid_c,\kappa_L)} Q$, we know that *stb*$(P)$. We examine all possible proof cases for conditions (1) and (2), given that P is a stable configuration. The proof cases for (1) are categorized below. There is only one proof case for condition (2), and it is similar to case 1 for condition (1), and we omit the details here.

1. Transitions that do not change the labels of any components (Figure 20).
   (a) Read/write to shared variable.
   (b) Launch component loop body for the first time through channel c_{ls}.
   (c) Re-launch component loop body through c_{ls}.
   (d) First output to c_{nI} right after the instance is created.
   (e) Output to c_{nI} from the activity manager.
2. Transitions that cause the component's label context to change (Figure 20, 21).
   (a) The special transitions that record the current labels for requests to the activity manager.
   (b) Launch an application.
   (c) Launch the shared variable *SVBody*.
   (d) Launch a component instance through its create channel c_{cT}.
3. Operations on labels.
   (a) Raise.
   (b) Declassify.
4. Component exit and application exit (Figure 23).
5. Calls and returns.
   (a) Calls (returns) are denied or delayed.
   In these cases, the label map does not change, and no low components are generated by the transition. The proofs for these cases are straightforward: we use Lemma J.1 and J.2.

(b) Calls are allowed.
In all proof cases, we distinguish between whether the update to the configuration is low or high. In the former case, because of the properties of the label operations, both the caller and the callee's label have to be low and thus Q takes the same step. In the latter case, P' relates to Q directly. We show two proof cases, one for intra-application calls (Figure 24) and the other for inter-application calls (Figure 25). The rest of the cases follow the same strategy.

(c) Returns are allowed.
The proofs for this case are similar to the case where a call to a launched application or a single-instance component is allowed.

□

Transitions that do not change the labels of any components. We show the case of reading the shared variable; the proofs for all other cases follow the same strategy.
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P) \mid P_1 \mid P_2)$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P) \mid P_1 \mid P_2')$
$P_2 = aid[...[(c_{AI}, c_{nI})[\nu r.\mathsf{out}\ aid{\cdot}c_{sv}(\mathsf{rd}, r).\mathsf{in}\ r(x).A(...)]].]])$
$\quad \mid aid[...[(c_{AI}, c_{nI})[\mathit{SVBody}(...)]]]]$
$P_2' = aid[...[(c_{AI}, c_{nI})[A(...)]]]]) \mid aid[...[(c_{AI}, c_{nI})[\mathit{SVBody}(...)]]]]$

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$, and Q contains P_2 as a sub-process,
Let $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_Q) \mid Q_1 \mid P_2)$
Q takes the same step as P to Q'
By Lemma J.2, $P'\ \mathcal{R}_{(aid_c, \kappa_L)}\ Q'$.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P) = \mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$.
Therefore,
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$.
By Lemma J.1, $P'\ \mathcal{R}_{(aid_c, \kappa_L)}\ Q$.

Transitions that cause the component's label context to change.

(a) Special transitions that record the current labels for requests to the activity manager. We list the case for call_E; other cases are similar.
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2)$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2')$
$P_2 = aid[\cdot\cdot[(c_{AI}, c_{nI})[\mathsf{out}\ a_m(\mathsf{call}_E, \mathit{rt}, aid, aid_{ce}, cid_{ce}, I)]]],$
$P_2' = (aid, (kA, kC))[\mathsf{out}\ a_m(\mathsf{call}_E, kA, kC, \mathit{rt}, aid, aid_{ce}, cid_{ce}, I)])$
The label map remains the same.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$, and Q contains P_2 as a sub-process.
Q takes the same step as P to Q'.
By Lemma J.2, $P'\ \mathcal{R}_{(aid_c, \kappa_L)}\ Q'$.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$.
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$.
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P)$
$\quad \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$.
Let $Q' = Q$,
By Lemma J.1, $P'\ \mathcal{R}_{(aid_c, \kappa_L)}\ Q'$.

(b) Launch an application.
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2)$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2')$
$P_2 = (aid, c_{AI})[\mathsf{out}\ aid{\cdot}c_L(c_{AI})] \mid aid[\mathit{App}(aid)]$
$P_2' = aid[c_{AI}[\mathit{AppBody}(aid, c_{AI})]] \mid aid[\mathit{App}(aid)]$
Let $\Xi(aid) = Q(kA_s, \_), \Xi(c_{AI}) = \mathsf{DA}(kA_d, \_)$.
By P is a stable configuration, we know that $kA_s \ll_s kA_d$.
By our assumption that the application with aid_c is already launched, we know that $aid \neq aid_c$.
The label map remains the same.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$, and Q contains P_2 as a sub-process.
Q takes the same step as P to Q'.
By Lemma J.2, $P'\ \mathcal{R}_{(aid_c, \kappa_L)}\ Q'$.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$,
or $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = aid[\mathit{App}(aid)]$.
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P) = \mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P)$.
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$.
Let $Q' = Q$,
By Lemma J.1, $P'\ \mathcal{R}_{(aid_c, \kappa_L)}\ Q'$.

Figure 20. Selected proof cases for Lemma J.5 from categories 1 and 2 (part 1 of 2).

Transitions that cause the component's label context to change.

(c) Launch SV
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2)$
$P_2 = aid[c_{AI}[\nu c_{svL}.\nu c_{sv}.\mathsf{out}\, c_{svL}(s_0).(P_3 \mid SV(c_{svL}, c_{sv}))]]$
$P_3 = (c_{AI}, cid_1)[CP_1(aid, cid_1, c_{AI}, c_{sv})] \mid \cdots \mid (c_{AI}, cid_n)[CP_n(aid, cid_n, c_{AI}, c_{sv})]$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.\nu c_{svL}.\nu c_{sv}.(P_1 \mid P_2' \mid P_3')$
$P_2' = aid[c_{AI}[(c_{AI}, cid_1)[CP_1(aid, cid_1, c_{AI}, c_{sv})]]] \mid \cdots \mid aid[c_{AI}[(c_{AI}, cid_n)[CP_n(aid, cid_n, c_{AI}, c_{sv})]]]$
$P_3' = aid[c_{AI}[SVBody(c_{svL}, c_{sv})]] \mid aid[c_{AI}[(SV(c_{svL}, c_{sv}))]]$
By our assumption that the application with aid_c is already launched, we know that $aid \neq aid_c$.
let $P_3' = P_2' \mid aid[c_{AI}[SVBody(c_{svL}, c_{sv})]] \mid aid[c_{AI}[(SV(c_{svL}, c_{sv}))]]$
The label map remains the same.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$, and Q contains P_2 as a sub-process
Q takes the same step as P to Q'. By Lemma J.2, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$
By all components in P_2 and $(P_2' \mid P_3')$ are guarded by the same channel c_{AI},
$\mathsf{proj}(\mathsf{sp}(P_2' \mid P_3'), \kappa_L, aid_c, \Xi_P) = \mathsf{proj}(\mathsf{sp}(P_3), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2' \mid P_3'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$.
Let $Q' = Q$,
By Lemma J.1, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

(d) Launch a component instance through its create channel c_{cT}.
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2)$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(P_1 \mid P_2')$
$P_2 = (aid, (c_{AI}, c_{nI}))[\mathsf{out}\, aid{\cdot}cid{\cdot}c_{cT}(I, c_{nI}, c_{lock}, rt)] \mid aid[c_{AI}[(c_{AI}, cid)[CP(aid, cid, c_{AI}, c_{sv})]]]$
$P_2' = aid[c_{AI}[(c_{AI}, cid)[CP(aid, cid, c_{AI}, c_{sv})]]] \mid aid[c_{AI}[(c_{AI}, cid)[(c_{AI}, c_{nI})[\nu c_{ls}.(\mathsf{out}\, c_{ls}(\sigma_0).\mathsf{out}\, c_{nI}(I) \mid CE(\cdots))]]]]$

Case: $aid \neq aid_c$. Even though P' contain components with a different label than P, they are still guarded by c_{AI}, so the proofs are similar to the previous case where no label is changed in the transition (1).

Case: $aid = aid_c$.
$\Xi(cid) = \mathsf{M}(kC_s)$, or $\Xi(cid) = \mathsf{SL}(\cdots, kC_s))$ and $\Xi(c_{nI}) = \mathsf{DC}(kC_d, \_)$
By P is a stable configuration, we know that $kC_s \ll_s kC_d$ or $kC_s \ll_{ms} kC_d$

subcase: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$, and Q contains P_2 as a sub-process.
Q takes the same step as P to Q'.
By Lemma J.2, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

subcase: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$, or
$\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = (aid, (c_{AI}, cid))[CP(aid, cid, c_{AI}, c_{sv})]$
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P) = \mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P)$.
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$.
Let $Q' = Q$,
ByLemma J.1, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

Figure 21. Selected proof cases for Lemma J.5 from category 2 (part 1 of 2).

Operations on labels.

1. An application raises its label
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P) \mid P_1 \mid P_2)$
$P_2 = aid[\cdots[(c_{AI}, c_{nI})[\mathsf{out}\, t_m(\mathsf{raiseA}, aid, c_{AI}, \sigma, \iota).A(\cdots)]]]$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P \uplus \Xi) \mid P_1 \mid P_2')$
$\Xi = aid \mapsto \Xi_P(aid) \uplus_{rz} (\sigma, \iota), c_{AI} \mapsto \Xi_P(c_{AI}) \uplus_{rz} (\sigma, \iota)$
$P_2' = aid[\cdots[(c_{AI}, c_{nI})[A(\cdots)]]]$
Since we disallow components in aid_c to raise the application label, it is the case that $aid \neq aid_c$.
Let $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_Q) \mid Q_0)$

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$, and Q contains P_2 as a sub-process.
Q takes the same step as P to Q'.
$aid_c, \kappa_L, \Xi_P \vdash \Xi : \mathsf{LO}$
By Lemma J.4, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

Case: $\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$,
$aid_c, \kappa_L, \Xi_P \vdash \Xi : \mathsf{HI}$
By Lemma I.2, $\Xi_P \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_Q$
The mapping of c_{AI} became higher after the raise operation.
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi) = \mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$,
By Lemma I.3,
$\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P)$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P) \leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_Q)$
Let $Q = Q'$, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

2. A component raises its label.
When $aid \neq aid_c$, the application-level label does not changes, so the proof is similar to proof case (1).
When $aid = aid_c$, it is similar to the raising application-level label case.

Figure 22. Selected proof cases for Lemma J.5 from category 3.

Component exit and application exit. We show one case where the exiting component is single-instance, application does not exit, and the rest of the cases are similar, since we only need to classify the updates to the label map as high or low.
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P)P_1 \mid P_2)$
$P_2 = aid[(kA, kC)[\mathsf{out}\, a_m(\mathsf{exitC}, kA, kC, aid, cid, c_{AI}, c_{nI}, rt, e)]]$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P \uplus \Xi) \mid P_1 \mid P_2'$
$P_2' = (aid_{ce}, (kA_{ce}, kC_{ce}))[\mathsf{out}\, a_m(\mathsf{return}, kA_{ce}, kC_{ce}, \cdots)]$
(where $rt = \mathsf{SOME}(\cdots)$, need to return to caller)
or $P_2' = \mathbf{0}$ (if $rt = \mathsf{NONE}$, no need to return to caller)
$\Xi = (aid{\cdot}cid \mapsto S(kC_s)), (c_{AI} \mapsto \Xi_P(c_{AI}) \backslash c_{nI}), \backslash c_{nI}$
Let $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_Q) \mid Q_0)$
By the rules for $\xrightarrow{\tau}_A$, and that exit is always the last instruction in a component,
$\Xi_P(aid{\cdot}cid) = \mathsf{SL}(\cdots, kC), \Xi_P(c_{AI}) = Q(kA, \_)$

Case: $(aid, (kA, kC))^* \sqsubseteq \kappa_L$,
$\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$ and Q contains P_2 as a sub-process.
Q takes the same step as P to Q'.
$aid_c, \kappa_L, \Xi_P \vdash \Xi : \mathsf{LO}$.
By Lemma J.4, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

Case: $(aid, (kA, kC))^* \not\sqsubseteq \kappa_L$
$\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$,
Because P_2 and P_2' are guarded by the same label,
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$= \mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$,
$aid_c, \kappa_L, \Xi_P \vdash \Xi : \mathsf{HI}$
By Lemma I.2, $\Xi_P \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_Q$
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$= \mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathbf{0}$
By Lemma I.3,
$\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P)$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$
$\leq \mathsf{proj}(\mathsf{sp}(Q_0), \kappa_L, aid_c, \Xi_Q)$
Let $Q' = Q$, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$.

Figure 23. Selected proof cases for Lemma J.5 from category 4.

Case: Activity manager processes an intra-application request to call an unlaunched single-instance component, and the call is allowed:
$aid = aid_c, \Xi_P(aid{\cdot}cid_{ce}) = \mathsf{S}(kC_{ce})$ and ${kC_{cr}}^- \sqsubseteq {kC_{ce}}^-$
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P) \mid P_1 \mid P_2)$
$P_2 = (aid, (kA_{cr}, kC_{cr}))[\,\mathsf{out}\, a_m(\mathsf{call}_I, kA_{cr}, kC_{cr}, rt, aid, c_{AI}, cid_{ce}, I)\,]$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.\nu c_{nI_{ce}}.\nu c_{lock}(T_{MI}(\Xi_P \uplus \Xi) \mid P_1 \mid P_2'$
$\Xi = aid{\cdot}cid_{ce} \mapsto \mathsf{SL}(c_{nI_{ce}}, c_{lock}, kC_{ce}),$
$c_{nI_{ce}} \mapsto \mathsf{DC}(c_{lock}, kC_{ce} \lhd kC_{cr})$
$P_2' = (aid, (c_{AI}, c_{nI_{ce}}))[\,\mathsf{out}\, aid{\cdot}cid_{ce}{\cdot}c_{cT}(c_{AI}, I, c_{nI_{ce}}, c_{lock}, \mathsf{NONE})\,]$

subcase: $kC_{ce} \lhd kC_{cr}^* \sqsubseteq \kappa_L$,
$aid, \kappa_L, \Xi_P \vdash \Xi : \mathsf{LO}$
By Lemma B.1, $kC_{cr}^* \sqsubseteq \kappa_L$ and $kC_{ce}^* \sqsubseteq \kappa_L$
Q contains P_2 as a subprocess
$\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$
Let $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_Q) \mid Q_1 \mid P_2)$
Q takes the same step as P to Q'
By Lemma J.4, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$

subcase: $kC_{ce} \lhd kC_{cr}^* \not\sqsubseteq \kappa_L$
$aid, \kappa_L, \Xi_P \vdash \Xi : \mathsf{HI}$
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi) = \mathbf{0}$
By Lemma I.2, $\Xi_P \uplus \Xi \precsim^{\kappa_L}_{aid_c} \Xi_Q$
By Lemma I.3,
$\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P)$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(P_1, \kappa_L, aid_c, \Xi_P)$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi)$
$\leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$
By Lemma J.3, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q$

Figure 24. A proof case for an intra-application call (for Lemma J.5, category 5).

Activity manager processes an inter-application request to call an launched application and an unlaunched single-instance component, and the call is allowed:
$aid_{ce} \neq aid_c, \Xi_P(aid_{ce}) = \mathsf{U}(kA_{ce})$,
$\Xi_P(aid_{ce}{\cdot}cid_{ce}) = \mathsf{S}(kC_{ce})$
${kA_{cr}}^- \sqsubseteq {kA_{ce}}^-, {kA_{cr}}^- \sqsubseteq {kC_{ce}}^-$,
${kC_{cr}}^- \sqsubseteq {kA_{ce}}^-, {kC_{cr}}^- \sqsubseteq {kC_{ce}}^-$
$P = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_P) \mid P_1 \mid P_2)$
$P_2 = (aid, (kA_{cr}, kC_{cr}))[\,\mathsf{out}\, a_m(\mathsf{call}_I, kA_{cr}, kC_{cr}, rt, aid, c_{AI}, cid_{ce}, I)\,]$
$P' = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.\nu c_{AI}.\nu c_{nI}.\nu c_{lock}(T_{MI}(\Xi_P \uplus \Xi) \mid P_1 \mid P_2')$
$P_2' = (aid_{ce}, c_{AI})[\mathsf{out}\, aid_{ce}{\cdot}c_L(c_{AI})]$
$\quad \mid (aid_{ce}, (c_{AI}, c_{nI}))[\mathsf{out}\, aid_{ce}{\cdot}cid{\cdot}c_{cT}(I, c_{nI}, c_{lock}, \mathsf{NONE})]$
$\Xi = aid_{ce} \mapsto \mathsf{L}(kA_{ce}, c_{AI}))$,
$\quad c_{AI} \mapsto (\mathsf{DA}(kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})\mathsf{LC}(\{c_{nI}\}))$,
$\quad aid{\cdot}cid_{ce} \mapsto \mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_{ce})$,
$\quad c_{nI} \mapsto (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr}))$

subcase: $kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})^* \sqsubseteq \kappa_L$,
$aid, \kappa_L, \Xi_P \vdash \Xi : \mathsf{LO}$
By Lemma B.1, Lemma B.2 $kC_{cr}^* \sqsubseteq \kappa_L$, $kA_{cr}^* \sqsubseteq \kappa_L$, and $kA_{ce}^* \sqsubseteq \kappa_L$
Q contains P_2 as a subprocess
$\mathsf{proj}(\mathsf{sp}(P_2), \kappa_L, aid_c, \Xi_P) = \mathsf{sp}(P_2)$
Let $Q = \mathsf{T_M} \mid \mathsf{A_M} \mid \nu\vec{y}.(T_{MI}(\Xi_Q) \mid Q_1 \mid P_2)$
By $\Xi_P \overset{\kappa_L}{\underset{aid_c}{\sim}} \Xi_Q$,
$\Xi_P(aid_{ce}{\cdot}cid) = \Xi_Q(aid_{ce}{\cdot}cid)$
Q can take the same step as P to Q'
By Lemma J.4
$P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q'$

subcase: $(kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr}))^* \not\sqsubseteq \kappa_L$,
$aid, \kappa_L, \Xi_P \vdash \Xi : \mathsf{HI}$
$\mathsf{proj}(\mathsf{sp}(P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi) = \mathbf{0}$
By Lemma I.2,
$\Xi_P \uplus \Xi \overset{\kappa_L}{\underset{aid_c}{\sim}} \Xi_Q$
By Lemma I.3,
$\mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(\mathsf{sp}(P_1), \kappa_L, aid_c, \Xi_P)$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(P_1, \kappa_L, aid_c, \Xi_P)$
$\mathsf{proj}(\mathsf{sp}(P_1 \mid P_2'), \kappa_L, aid_c, \Xi_P \uplus \Xi) \leq \mathsf{proj}(\mathsf{sp}(P_1 \mid P_2), \kappa_L, aid_c, \Xi_P)$
By Lemma J.3, $P' \; \mathcal{R}_{(aid_c, \kappa_L)} \; Q$

Figure 25. A proof case for external calls (for Lemma J.5, category 5).