Information-Flow Control in Modern App Platforms: Modeling, Formal Verification, and Controlled Declassification

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Jassim M. Aljuraidan

B.S., Computer Engineering, Kuwait University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

December 2018

Abstract

Smartphones are today used by people around the world to manage and organize many aspects of their digital life from financial and health information to everyday social interactions. Modern mobile OSs allow applications from a variety of vendors to be installed. These applications might not all be trustworthy. While mobile OSs employ several security measures to protect the user data, researchers have shown that these measures are not enough. Android, the most widespread smartphone OS in the market, has been vulnerable to attacks such as information leakage and privilege escalation.

This thesis studies the application of information-flow control (IFC) schemes to Android. IFC can provide formally proven security guarantees that enable a high level of assurance.

We start by proposing a tag-based IFC policy specification scheme and run-time enforcement mechanism for Android. This mechanism can prevent attacks such as information leakage and privilege escalation. Policies can be specified for apps and individual components within an app. App-level policies are robustly enforced because Android provides strong isolation between apps. However, enforcement of component-level policies is best effort because Android allows components to communicate directly and avoid the monitoring of the enforcement mechanism. To bridge this gap, we develop a static analysis tool that detects direct information flows between components. This tool can be used by developers to ensure that their apps lack such unmonitored flows, which makes component-level enforcement robust. We used the tool to study more than 2,500 of the most popular Android apps. We found that about 17% of these apps do not use direct communications between components and thus have robust component-level enforcement. However, the majority of these apps had very well connected components, making enforcing component-level policies by, for example, preventing these flows through instrumentation, impractical.

In addition, we developed a faithful process-calculus based model of Android and our enforcement mechanism. We then formally prove that the model satisfies the basic noninterference property. However, basic noninterference only guarantees security in the absence of *declassification*. Declassification capabilities provide flexibility to the otherwise too strict IFC policies. Therefore, we propose a novel noninterference definition based on label versioning that, unlike other enhanced noninterference definitions, supports label reuse.

Even more expressive policies can be supported if conditions can be specified to control declassification. We propose an enhancement to *stateful declassification* that supports the specification of such conditions based on global, app-level, and instance-level state. We show that this approach can be used to enforce interesting security policies.

Acknowledgments

To begin, I would like to thank my advisor extraordinaire Lujo Bauer for all his time and effort spent on my education. It has been an enormous privilege to work with him. Throughout my years as a PhD student, he offered guidance and advice, and provided valuable feedback on many papers and presentations, all with incredible patience. I wish I can be to my future students a fraction of the advisor he has been to me.

I must also thank Limin Jia who has been my de facto co-advisor during my time in CMU. She provided valuable feedback for my dissertation and other papers. My thanks to Andrei Sabelfeld and Anupam Datta, the other members of my committee for their feedback, comments, and putting up with the inconvenience of scheduling times across continents.

I would also like to thank Kuwait University for the generous scholarship, without which I would not have been able to finish my PhD.

A heartful thank you goes to my whole family, especially my mother, who always gave support and unconditional love.

Last, but certainly not least, I would like to thank Ayat, my wife and my best friend. She endured uncountable hardships throughout these years far away from her home and family. I am forever grateful.

Thesis Committee:

Prof. Lujo Bauer (Carnegie Mellon University), Chair

Prof. Limin Jia (Carnegie Mellon University)

Prof. Anupam Datta (Carnegie Mellon University)

Prof. Andrei Sabelfeld (Chalmers University of Technology)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past decade, the rapid rise of smartphone usage has become a global phenomenon. In fact, using smartphones has become the norm in large parts of the world, adopted by adults, and in many cases children. Moreover, the usage of smartphones has become so integrated into our daily life that most of our digital footprint exists in these devices, from private pictures and videos to business emails, health records, and financial information. Applications installed on these smartphones are sourced from different vendors with varying degrees of trustworthiness. This combination of characteristics of modern smartphones makes it especially critical to protect the user's data. Unfortunately, the security mechanisms that the operating systems of these smartphones provide to protect users' data are inadequate. While mobile OSes employ various security mechanisms such as application isolation and the protection of sensitive APIs with permissions, research has shown many vulnerabilities still remain. Android, for example, the most popular of these mobile OSes, has been shown to be vulnerable to information leakage [1, 2] and privilege-escalation attacks [3, 4].

Many solutions have been proposed to control undesired information leakage and privilege escalation, but no solution is comprehensive. One of the most promising approaches is information flow control (IFC) [5, 6]. Information flow control works by associating each part[1] of a computation environment with a label that usually represents a specific security level. Typically, a partial order relation is defined between these labels. The set of all labels in a system constitute its security policy. These labels are then used, by a policy enforcement mechanism, to restrict the flow of information such that information can only flow from one, otherwise isolated, part of the system to another if the source's label is less than or equal to the destination's according to the partial order. A problem with this approach, when applied naively, is that it's too restrictive. The standard solution is to allow information to leak, i.e., flow from parts with higher secrecy labels to ones with lower secrecy, but only through a privileged operation [7]. The privilege to use this operation, called declassification, should only be granted to trustworthy entities, and those entities should use it with care.

[1]A part maybe be a single variable, a whole application or anything in between.

Thesis Statement

Information-flow control policies and enforcement mechanisms, augmented with controlled declassification using traditional access-control policies, can provide expressive, practically useful, and provable security guarantees for Android.

To support this thesis, we present the following body of work: We define the IFC mechanism and policies (Ch. 2), and the controlled declassification (Ch. 5) for Android; We enhance the practicality and usefulness of our approach by designing our system to use tag-based policies with floating-labels (Sec. 2.2) and letting developers specify finer-grained policies (Chs. 2 and 3); We demonstrate through motivating scenarios with policies that cannot be specified in the regular Android OS (Sec. 2.1) or without our enhanced controlled declassification (Secs. 5.1 and 5.4) the expressiveness of our policies; and we formally prove the security guarantees of our approach (Ch. 4 and Sec. 5.3.3).

IFC for Android We first propose a new policy specification scheme and decentralized IFC (DIFC) run-time enforcement mechanism for Android that is inspired by DIFC designs in legacy operating systems [8]. Android applications are composed of smaller software modules called components. Policies are both specified for and enforced at the levels of whole applications and application components. Previous works either enforced a pre-determined policy [9, 10, 11] or enforced fine-grained policies, that are often cumbersome to specify and expensive to enforce. Most of these works also lack a formal investigation of the security properties they provide [9, 10]. Our proposed approach, we believe, strikes a better balance in terms of granularity of enforcement, and uses a high-level specification language. In addition, we present a formal proof of its security guarantees in the form of a noninterference property proof [7].

Formal proofs of properties, such as noninterference, are essential for high assurance security. However, these proofs depend on certain assumptions about the actual system. In our case, we assume that both applications, and their components, are isolated from each other. Unfortunately, this assumption does not entirely hold in Android at the component level.

Finer-grained policies and enforcement The policy in our work can be specified at both the application level and the component level. However, while the applications are robustly isolated in Android, the components within an application are not. This problem makes policy enforcement at the component level best-effort only, as it is the responsibility of developers to make sure that their components only communicate through channels that are monitored by the enforcement system, i.e., using Android APIs. As a step toward bridging this gap, we developed a static analysis tool that reports any potential inter-component information flow, which does not go through the monitored channels. We use this tool to analyze more than 2,500 apps from Google Play with the goal of studying the prevalence of such unmonitored flows. Using this tool, app developers or security engineers can analyze their apps and either adjust their policy or implementation if unmonitored flows are detected. Moreover, this tool can be used to insert additional run-time checks such that our enforcement is always sound at the component level —at the expense of potentially affecting the app functionality.

Even with robust isolation, the noninterference property guarantees are limited by the powerful declassification privileges. We aim to alleviate this limitation by introducing a new noninterference property.

Versioned-labels noninterference property One of the main advantages of using IFC is that, given an enforcement mechanism, we can formally prove the security of our enforcement using a proof of the noninterference property. This property guarantees that a malicious observer that has access to only the parts of the system with certain labels won't be able to access, or infer based on system behavior, any information from parts with higher secrecy labels.

A critical issue with the basic noninterference properties, and their corresponding security policies, including the one for our initial work described above, is that they only guarantee the correct behavior of the system as long as declassification is not used. This means that theoretically, and practically, once a declassification operation is used, there are no guarantees about the flow of information. Several approaches has been proposed that allow for noninterference properties and security policies to include declassification, including intransitive noninterference [12], relaxed noninterference [13], gradual release [14], and others [15, 16]; however, they are either specific to certain kinds of policies or don't provide guarantees when declassified labels are reintroduced by a label raise operation [12, 13, 15]. Therefore, we introduce a definition for a more precise noninterference property, which relies on the novel idea of *label versioning* to describe the effect of declassification, enable label reuse, and provide formal security guarantees. We then prove that our Android model does indeed satisfy this property.

The versioned-labels noninterference property provides a strong security guarantee for IFC policy enforcement. But, IFC policies may not be expressive enough for particular scenarios. Access-controlled declassification allows a richer set of security policies to be specified and enforced.

Access-controlled declassification A problem with declassification, is that it's usually unrestricted: as long as an app is granted the required capability, it can be used as many times as desired, on any amount or type of data, etc. The versioned-labels noninterference definition only restricts this usage to the specific apps with corresponding declassification capabilities. We investigate the following approach to address this issue. At each declassification point, access control checks are implemented to enable the enforcement of security policies that are more expressive than what is possible with current IFC systems. Previous works, which investigated similar approaches, were mostly for single programs not a platform with multiple apps like Android, and limited the declassification control to logical conditions, without access to both global and local state [11, 16]. *Stateful declassification* does enable conditions based on a state variable, but does not explicitly differentiate between global state and local state, nor address the challenge of managing access to the global state without violating the IFC policy as we do [17]. We start by modeling the Android runtime, the IFC enforcement mechanism, and the access control mechanisms. Then, we use this model to study what security properties can be proven and show how example policies can be specified and enforced.

The term "access control" is very broad and sometimes can encompass IFC. Thus, it is important to clarify that when we say "access control," we specifically mean access control schemes or models that are not IFC.

Contributions In summary, the following are the contributions of this thesis.

- Developing an IFC policy specification language (based on security labels) with run-time enforcement for Android, formally modeling the Android runtime and our enforcement mechanism using a process calculus, and proving trace-based noninterference for the model.[2]
- Developing a static analysis tool to investigate the previously unmonitored information flows between components of the same application, and studying 2,500+ of Android's most popular applications using this tool where we find that for the majority of these applications run-time monitoring at the component level is not practical.
- Developing a formal, simplified, labeled-transition-system model for the above enforcement mechanism and the Android runtime, and proving it using the Coq proof assistance for greater assurance.
- Proposing a novel approach for describing the effects of declassification on information flow for the purpose of defining and proving the versioned-labels noninterference property, which admits label reuse, for our model.
- Proposing an extension to the stateful declassification [17] approach, which enhances the expressiveness of regular IFC schemes by inserting controls at the declassification points

[2]This was part of a joint work with Limin Jia (first author), Lujo Bauer, Elli Fragkaki, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, Yutaka Miyake.

that implement access-control policies based on local, app-specific, and global state; and incorporating this approach in our model, and versioned-labels noninterference proof, to prove some interesting security properties, which are not usually proven for IFC systems.

Outline The remainder of this document is organized as follows. First, we complete this introduction with background information about the Android system. In Ch. 2, we discuss our proposed run-time enforcement mechanism for Android. After that, we discuss our static analysis study of component-level enforcement in Ch. 3. Next, in Ch. 4, we discuss the formal verification of security properties for our mechanisms. Then we detail our approach to enhance DIFC systems with access control checks in Ch. 5. Finally, Ch. 6 concludes this work with a summary and final remarks.

1.1 Android Architecture

Android is a Linux-based OS; its applications are written in Java and each executes in a separate Dalvik Virtual Machine (DVM) instance. Newer versions of Android replaced the DVM with Android Runtime (ART), though both runtimes execute the same application format. Applications are composed of *components*, which come in four types: 1) *activities* define a specific user interface (e.g., a dialog window); 2) *services* run in the background and have no user interface; 3) *broadcast receivers* listen for system-wide broadcasts; and 4) *content providers* provide an SQL-like interface for storing data and sharing them between applications.

Activities, services, and broadcast receivers communicate mainly via asynchronous messages called *intents*. An intent encapsulates possible arguments and the intended recipient. If a recipient of an intent is not instantiated, the OS will create a new instance. The recipient of an intent is specified by its class name or by the name of an "action" to which multiple targets can subscribe. Any component can attempt to send a message to any other component. In addition, components can use the *binder* API to communicate with components in other applications. While the binder API is more versatile to support, for example, streaming or synchronous communication, it requires significantly more setup and maintenance and thus is not suitable for regular interactions.[3] The OS mediates both cross- and intra-application communications via intents or binder APIs. Between applications, intents and the binder API are the only (non-covert) channel for establishing communication.[4]

Components within an application can also communicate in other ways, such as via public static fields. Such communication is not mediated, and can be unreliable because components are short lived—Android can garbage collect all but the currently active component. Hence, although Android's abstractions do not prevent unmediated communication between components, the programming model discourages it. We will often write that a component *calls*

[3]Refer to `https://developer.android.com/reference/android/os/Binder.html`. In fact, cross-application intents are delivered using the binder mechanism.

[4]The exception is native apps, which can utilize other OS level IPC mechanisms available in Linux, such as pipes or shared memory.

another component instead 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 at install time (or use time in recent versions of Android), with the user's consent. Additionally, content providers use *URI permissions* to dynamically grant and revoke access to their records, tables, and databases.

Chapter 2

Run-time Enforcement of IFC policies in Android

In this chapter we will present our approach to enforce an information flow policy in Android. This chapter aims to support our main thesis by showing how an IFC approach can provide more expressive but still practical policies than currently possible in Android. We start by describing a scenario that exemplifies the Android permission system's inability to implement many simple, practical policies (Sec. 2.1). We then discuss key aspects of our system and show it can specify (Sec. 2.2) and enforce (Sec. 2.3) richer policies that are still practically useful for app developers.

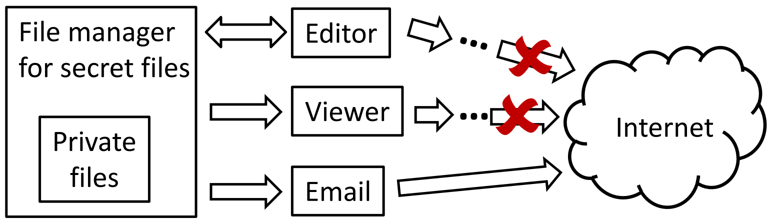

Figure 2.1: A simple scenario that cannot be implemented using Android permissions. The secret files should only be sent through the email app after the user approval.

2.1 Motivating Scenario

Suppose a system has the following applications: a *secret-file manager* for managing files such as lists of passwords and bank-account numbers; a general-purpose text *editor* and a *viewer* that can modify and display this content, respectively; and an *email application*. Because of their sensitive content, we want to prevent files managed by the secret-file manager from being inadvertently or maliciously sent over the Internet; this should be allowed only if the user explicitly requests it through the file manager. The files need to be viewed and edited. Building a special editor and viewer just for secret files is impractical, and so these tasks have to be handled by a general-purpose editor and viewer. This scenario is shown in Fig. 2.

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

2.2 Key Design Choices

In general, the attacker model we consider is one where an application may try, maliciously or inadvertently, to exfiltrate data or access sensitive resources in violation of the policy, including by taking advantage of cooperating or buggy applications.

We now discuss the design of our system, including the level of abstraction at which we enforce policies, and explain how it can specify and enforce our desired example policy.

2.2.1 Enforcement Granularity

Traditionally, information-flow properties have been enforced either at instruction level (e.g., [18, 19]) or at process level (e.g., [8]). Android's division of applications into components invites the exploration of an interesting middle ground between these two. Android applications are typically divided into a few key components, e.g., an off-the-shelf file manager with which we experimented was comprised of five components. Hence, component-level specification is most likely not drastically more complex than application-level specification. This additional granularity, however, could enable policies to be more flexible and better protect applications (and components) from harm or misuse.

Unfortunately, enforcing purely component-level policies is difficult. The Android programming model strongly encourages the use of components as modules. In fact, the Android runtime may garbage collect any component that is not directly involved in interacting with the user; using Android's narrow interface for communication between components is the only reliable method of cross-component communication. However, neither Android nor Java prevent components *that belong to the same application* from exchanging information without using the Android interfaces, e.g., by directly writing to public static fields. Hence, Android's component-level abstractions are not robust enough to be used as an enforcement boundary; fully mediating interactions between components will require a lower-level enforcement mechanism. Although such low-level enforcement is possible, e.g., with instruction-level information-flow tracking [18], implementation and integration with existing platforms and codebases is difficult and can cause substantial run-time overhead.

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

2.2.2 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 is a set of *integrity tags*, and δ is 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*, or *dynamic*, 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 Implementing secrecy and integrity labels as sets of tags was motivated by the desire to help with backward compatibility for standard Android permissions. In Android, any application can declare new permissions at installation time. We similarly allow an application to declare new secrecy and integrity tags, which can then be used as part of its label. The lattice over labels does not need to be explicitly declared—this would be impractical if different applications declare their own tags. Rather, the lattice is defined by the subset relation between sets of tags. The permissions that legacy applications possess or require of their callers can be mapped to tags and labels. For the purpose of enforcement only a partial-ordering of the labels is really required. We use the term *lattice* because it is often used in related papers.

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

Returning to the example from Sec. 2.1: The secret-file manager application may be labeled with the policy ({FileSecret}, {FileWrite}, {-FileSecret}). Intuitively, the first element of this label conveys that the secret-file manager is tainted with the secret files' secrets (and no other secrets); the second element conveys that the file manager has sufficient integrity to add or change the content of files; and the third element states that the file manager is allowed to declassify by removing FileSecret from its secrecy label. The file manager's effective label will initially be the same as this static label. If the file manager exercises its declassification capability -FileSecret, its effective label will become ({}, {FileWrite}, {-FileSecret}).

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

Floating labels Some components or applications, e.g., an editor, may have no secrets of their own but may want to be compatible with a wide range of other applications. In such cases, we can mark the secrecy or integrity label as *floating*, e.g., (F{}, F{}, {}), to indicate that the secrecy or integrity element of a component's effective label is inherited from its caller. The inheriting takes place only when a component is instantiated, i.e., when its effective label is first computed. Floating labels serve a very similar purpose to polymorphic labels in Jif [18].

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

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

2.3 Enforcement Approach and Limitations

In Sec. 2.2, we described how to specify rich, practically useful policies in our system; we next outline how they are enforced. The crux of our enforcement system is a reference monitor that intercepts calls between components and permits or denies a call based on the caller's and callee's labels. Much of the reference monitor responsibility is maintaining the mapping from applications and components (and their instances) to their effective labels. We next discuss how our reference monitor makes enforcement decisions and how our system handles persistent state.

2.3.1 Application- and Component-Level Enforcement

When two components try to communicate via an intent, our reference monitor permits or denies the call by comparing the caller's and the callee's labels. When the caller and callee are part of the same application, the call is allowed only if two criteria are met: 1) the caller's effective secrecy label is a subset of the callee's and 2) the caller's effective integrity label is a superset of the callee's. If both criteria are not met, 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.

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

Declassification and endorsement change the effective labels of components and applications, and are permitted only when consistent with policy.

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

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

2.3.2 Persistent State

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

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

Consider, for example, malicious components A and B that seek to communicate via a colluding single-instance component C. Suppose that A's static secrecy label is $\{\mathsf{FileSecret}\}$ and B's is $\{\}$, preventing direct communication from A to B; C's static secrecy label is $\{\}$. Component C, upon starting, sends B an intent, then raises its effective label to $\{\mathsf{FileSecret}\}$. A sends the content of a secret file to C; their labels permit this. If the content of the secret file is “Attack at dawn,” C exits; otherwise, C continues running. B calls C, then calls C again. If B receives two calls from C, then it learns that A’s secret file is “Attack at dawn.” C can only make the second call to B after exiting, which only happens when A’s secret file is “Attack at dawn.” The information leak in this scenario arose because C changed its label by exiting. To prevent such scenarios (and to allow us to prove noninterference, which ensures that no similar scenarios remain undiscovered), raising a label must change not only a component’s effective label, but also its static label.

2.3.3 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, e.g., timing channels. Recent work has identified ways in which these may be mitigated by language-based techniques [20]; but such techniques are outside the scope of this paper. We also do not directly address the robustness of Android’s abstractions: stronger component-level abstractions would permit robust, instead of best-effort, enforcement of information-flow policies within applications. Improving these abstractions, or complementing them by, e.g., static analysis, could thus further bolster the efficacy of our approach; we discuss these possibilities in Ch. 3.

Many security architectures are vulnerable to user error. On Android, a user can at installation time (or use time in recent versions) consent to giving an application more privileges than is wise. Our system does not address this; we design an infrastructure that supports rich, practically useful policies. Because our approach allows developers to better protect their applications, they may have an incentive to use it. However, we do not tackle the problem of preventing the user from making poor choices (e.g., about which applications to trust or which permission to grant). Furthermore, it has been shown that the vast majority of users do not usually understand or even pay much attention to these permissions [21]. Therefore, it’s not our intention that end users get directly involved in approving the policies in our system, which can be more complicated than simple permissions. These policies, which are set by app developers, should be vetted using an intermediary. For example, in a corporate environment where smartphones are managed by the IT department, the policies can be vetted by a security expert in the department. For average users, we envision that cyper-security companies or organizations could provide services, automated or otherwise, that uses high-level user preferences, among other criteria, to vet app policies. Approved policies should be fully enforced regardless of the existence of malicious or inadvertently misbehaving apps.

2.4 Related Work

In this section we first discuss related work on IFC and then works on Android security in general.

2.4.1 Information Flow Control

Enforcing information-flow policies has been an active area of research. Some develop novel information-flow type systems (cf. [22]) that enforce noninterference properties statically; others use run-time monitoring, or hybrid techniques (e.g., [19, 23, 24, 25, 26, 27]). These works track information flow at a much finer level of granularity than ours. In contrast, the goals of our design included minimally impacting legacy code and run-time performance on Android.

Our approach is most similar to work on enforcing information-flow policies in operating systems [8, 28, 29, 30]. 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 works do not prove any formal properties of their enforcement mechanisms. Krohn et al. [31] presented one of the first proofs of noninterference for practical DIFC-based operating systems. Our design is inspired by Flume [8], but has differences. For instance, Flume does not support floating labels. In Android, as we show through examples, floating labels are of practical importance.

2.4.2 Android Security

Many works have recently focused on Android security, analyzing Android's permission system [10, 32], developing tools to detect misbehaving applications [33, 34, 35], and proposing new protection mechanisms (e.g., [36, 37, 38, 39]). Android's permission system has been shown inadequate to protect against many attacks, including privilege-escalation attacks [3, 4] and information leaks [1, 2, 9, 40]. Closer to the goal of our work are projects such as TaintDroid [9] and AppFence [41], which automatically detect and prevent dangerous information leaks. They operate at a much finer granularity than our mechanism, tracking tainting at the level of variables, enforce fixed policies, and have not been formally analyzed. 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 [42]. 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 [43]. And more recently, Bugiel et al. proposed FlaskDroid, a mandatory access control solution for Android inspired by SELinux [44], and were able to instantiate previous mechanisms using it [45]. All 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. Most recently, a number of IFC based solutions has been proposed. Maxoid [46] confines apps, by blocking their access to the outside world or the file systems, when they receive secrets. Weir [47] instantiates different versions of the application whenever there is a conflict between the caller the callee's security labels. Both Maxoid and Weir lack formal guarantees. In addition, Maxoid policies are very limited in their expressiveness.

Formal analyses of Android-related security issues and language-based approaches to solving them have received less attention. Shin et al. [48] developed a formal model to verify functional correctness properties of Android, which revealed a flaw in the permission naming scheme and a possible attack [49]. Sorbet proposed a set of enhancements to Android's permission system designed to enforce information-flow-like policies, for which some correctness properties were also formally proved [50]. The work described in this thesis is different in several respects: we build on more well-understood theory of information flow; we support more flexible policies (e.g., in prior work it is not possible to specify that information should not leak to a component unless that component is protected by some permission); we make persistent state more explicit; and we formally model our enforcement system in much greater detail, thus providing much stronger correctness guarantees.

Chapter 3

Component-Level Robust Enforcment/Robust Enforcement at the Component Level

In this chapter we study the feasibility of robust enforcement of component-level IFC policies in Android. The high level goal is to support our thesis claim of provable security guarantees for the component-level enforcement.

Noninterference, the security property of our system, requires isolation and complete mediation of all communication channels between components and applications. Otherwise, unmediated communication channels can potentially be used to introduce information flows that violate noninterference. While Android provides isolation between applications, it is each developer's responsibility to ensure that components inside one application do not communicate except through monitored interfaces. As a result, our policy enforcement at the component level is best-effort only. For this reason we built a static analysis tool that will detect such unintended communication.

Android applications, except those intended to run natively, run on their own isolated virtual machine, and thus are isolated from other applications running on the system. They can only communicate through the monitored formal channels by calling Android APIs. However, components within the same application share the same memory space, and can communicate directly without going through any APIs.

Fortunately, Java does not allow arbitrary addressing of memory; memory can only be accessed using legitimate memory references. Therefore, two components can only directly communicate if they have access to a shared reference. There are only two ways this can happen: (1) through Java's static fields, and (2) through references passed from Android library code. We will only focus on the first scenario here, and trust the library code to not pass references between components or pass the same reference to more than one component. For the purpose of our analysis, we define a component's code slice as all statements reachable from at least one entry point that is associated with this component. We base our analysis on the following observation. In order for two components to communicate through shared memory, there must be a write from one of the components and a read from the other of the same memory location. By identifying all possible reads and writes to all reference and primitive variables, we can conservatively, but not precisely, detect possible information flows between components.

The immediate purpose for building the static analysis tool was to gain some insight into the use of these direct communication channels, or information flows, in real applications. For example, we wanted to know the prevalence of usage; whether usage is usually limited to few components; whether these flows are usually bi-directional. These statistics should help us judge the viability of future approaches. If the usage is too prevalent then it would be futile to try and control it. But if it's limited to a couple of components then trying to control it makes more sense. In addition, this tool can be used to aid developers in avoiding unintentional flows, which could violate their own component-level policies. Finally, we were aiming to leverage this tool to make the run-time enforcement of component-level policies more robust. One potential way to achieve our aim is by instrumenting the application with run-time checks to interrupt the application when potential policy-violating flows are exercised at run-time.

The remainder of this chapter is organized as follows. Sec. 3.1 gives an overview of our static analysis tool. We discuss implementation details in Sec. 3.2. Then, we list the limitations and assumption of the analysis in Sec. 3.3. Next, in Sec. 3.4, we describe our experiment of analyzing more than 2500 apps using the tool. Finally, Sec. 3.5 finishes this chapter with a discussion of the results and our conclusions.

3.1 The Static Analysis

We start this section with a brief overview of our static analysis. Then we list significant decisions that we made during the design of the analysis tool.

3.1.1 Overview

There are three main phases of our analysis. First, in Phase 0, we prepare for the actual analysis by decompiling the application and initializing the analysis framework. The next phase, Phase 1, builds an abstracted model of each component's code base. Phase 1 analysis starts with an initial set of code entry points. However, the analysis is iterated until all the other entry points, in the form of *callback methods*, are discovered. In each iteration, we first performs a *points-to* analysis to discover all potential objects that references in the code can point to, and then use the result to build a *call-graph*, which represent the calling relation between all methods in the code. In the last phase, Phase 2, we search the code models from Phase 1 to find all potential information flows through static fields. We declare a potential information flow whenever a static field is read by some component and written to by another. In Sec. 3.2, we share more details about theses phases and how they are implemented.

3.1.2 Noteworthy Design Decisions

Soundness An important criteria for this tool was for its analysis to be sound. This essentially meant that whenever there was a trade-off between soundness and either precision or performance, we chose soundness. For the tool to be sound, we decided that it needs to detect all potential (direct) flows between all components, even if some of them turned out to be impossible to exercise.

Entry points In order to build the call-graph and perform the points-to analysis, a set of entry points need to be specified. In a regular Java application the explicit entry point is the method `main`. We need to consider implicit entry points, such as class initializers and callbacks. However, for an Android application, and for the purpose of our analysis, the entry points are somewhat different. First, we did not want to include Android run-time code that runs before the application code even starts. Our assumption is that we trust the library code to not introduce direct information flows between components. Second, from the onset, multiple entry points that need to be considered: the Android component life-cycle methods (e.g., the methods to start, resume, or stop a component). In our analysis we considered any public or protected[1] method in the class that implements a specific component, or any of its superclasses, to be an entry point for this component.

Dynamic Broadcast Receivers Broadcast receivers can be registered dynamically without declaring them in the Android manifest. In order to include the code that runs when the `onReceive` method is called in our analysis, we need an approach similar to how we handle callbacks (see Sec. 3.2). In practice we consider broadcast receivers as special callback interfaces (though it does not necessarily implement any interface).

Equality of instance field accesses We consider two instance field accesses to be equal if they could refer to the same field, i.e., the same class and the same field name, and the intersection of the points-to sets of the references is not empty. This equality test is only used during set intersection computations in the last phase.

3.2 Implementation

In this section, we give an overview of how we implemented the phases of our analysis described in Sec. 3.1.

For carrying out our analysis we chose the Soot framework [2]. The main reason behind this choice was Soot's flexibility in terms of instrumentation, as opposes to, for example, IBM's WALA framework[3]. We wanted to have this option available for future work.

Phase 0: Pre-analysis There are three necessary steps to prepare for the actual analysis. First, the "Android Mainfest" file is extracted from the APK, using apktool [4], and a list of all

[1]Protected methods can be called by the library code in the super-class, e.g., the `onCreate` method of the `Activity` class

[2]http://sable.github.io/soot/

[3]https://github.com/wala/WALA

[4]https://code.google.com/p/android-apktool/

components in the application is compiled. Second, we use DARE [5], to translate the application's binary from Dalvik (Dex) to Java bytecode. Lastly, the tool initializes Soot's environment, which includes compiling a list of initial entry points for each component.

Phase 1: Points-to analysis and callback discovery Android applications are vastly interactive and GUI-driven. Thus, programming for Android involves using callback interfaces frequently. A callback interface is a Java interface in the Android run-time library, such that objects implementing it can be registered to receive callbacks via specific library functions. A callback is triggered by a certain asynchronous event, and when said event occurs the library code calls a designated method defined in the callback interface. Since it's not possible to determine all the methods that are registered as callbacks from one analysis pass over the code, an iterative process is needed. In each iteration of this phase, we first perform a points-to analysis and build the call-graph. Then, for each component, we search all the methods that are reachable from the current list of the component's entry points for references to potential callback objects, i.e., instances of classes that implement one or more of the (predefined) callback interfaces. We add the potential callback methods of those objects to the component's entry points list. We repeat this process until no new callback objects are found.

Phase 2: Detecting Potential Flows The goal of the final phase of the analysis is to actually find all the potential information flows. We begin by analyzing, for each component C_X, all reachable methods (starting from its entry points) to collect all static or instance fields written, W_X, and read, R_X. Then, for each component pair (C_A, C_B), we calculate $F_{A \to B}$, which is the intersection of the set of written fields in C_A and the set of read fields in C_B, i.e., $F_{A \to B} = W_A \cap R_B$. If the set $F_{A \to B}$ is empty, then there is no (direct) flow from C_A to C_B, otherwise the flow exists, and its *strength* is the number of elements in $F_{A \to B}$.

[5]http://siis.cse.psu.edu/dare/

Points-to Analysis A major component of our tool is the points-to analysis in Phase 1. During the initial development we used the faster, but less precise, context-insensitive analysis, by utilizing the Spark package from the Soot framework. Having fast analysis helped a lot with the development. However, toward the end we noticed that while the result were sound, too many false-positives were being reported by the tool. Therefore, we replaced Spark with the Paddle package, which implemented a context-sensitive analysis. This change led to more precise results at the expense of extra computation time.

3.3 Assumptions and Limitations

We note the important assumptions we make when interpreting the output of the tool and when considering the robustness of component-level enforcement.

We assume that the Android run-time library does not allow communication between components, except through the channels that are monitored at run-time (e.g., intent-based communication and the shared preferences storage). In particular, we assume that the library code does not pass references between components or pass the same reference to more than one component. This is reasonable in the context of our work because one can independently verify this assumption and, if necessary, modify the run-time library; and this only needs to be done once. In addition, we assume that all the application level abstractions are robust enough that components cannot communicate, say, by utilizing external colluding applications. In addition, our tool is limited to APKs that don't contain native code, which runs outside the virtual machine.

3.4 The Experiment

We applied the tool to the most popular applications in each category from the Google Play application store (March 2017). The tool was not able to process a significant percentage of the

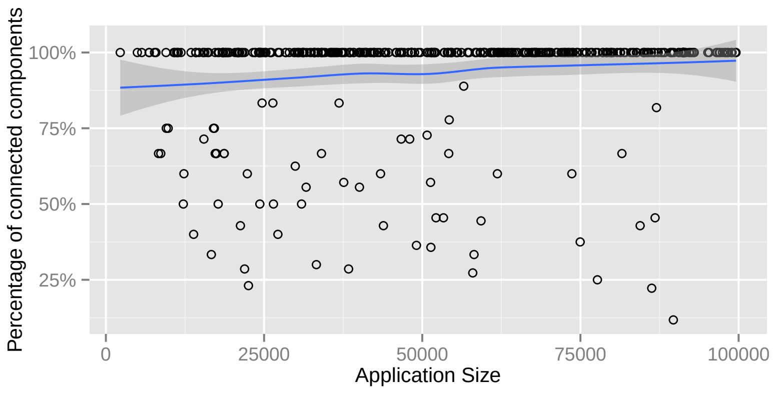

Figure 3.1: Percentage of connected components vs application size (in number of Jimple[7] statements). The line is obtained by using LOESS fitting, and the shaded area represents the 95% confidence interval.

applications in this set for several reasons. The majority of those failures resulted from either failures in the underlying libraries or tools we used, or miss-configured APKs (for example, many applications mention nonexistent classes in their Android manifest). The following results are from those applications that were successfully processed, that is 2,512 out of the 4,418 applications. To facilitate the analysis, we view applications as undirected graphs with components as the vertices and (potential) information flows as links.

About 17% of the applications are completely disconnected, i.e., there are no direct flows between components. This disconnection means that for a small portion of these applications, component-level policy enforcement will automatically be as robust as application-level enforcement. The average number of components in an application is 10.4 overall. However, for the completely disconnected applications, the average is only 3.1 components per application. This suggest that completely disconnected applications tend to be smaller in size. This does not suggest the converse, however, i.e., connected applications are not mostly larger in size. In fact, among connected applications, there is no strong relation between the percentage of connected components in an applications and its size (see Fig. 3.4).

In Fig. 3.4, a histogram of the connectedness of the applications components is shown. Connectedness is calculated as follows.

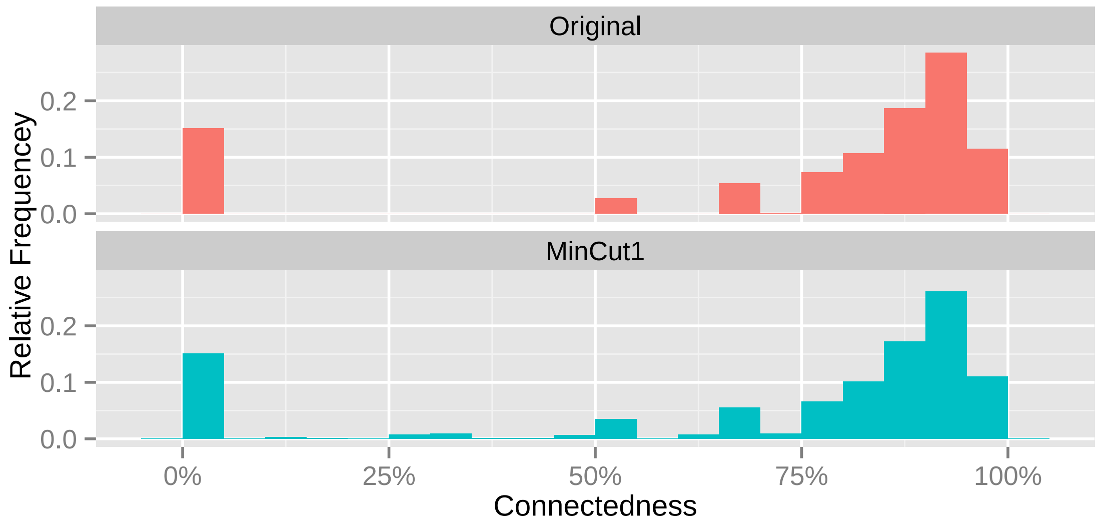

Figure 3.2: Connectedness of applications

$$connectedness = 100 * (1 - \frac{\text{number of disconnected subgraphs in the graph}}{\text{number of components}})$$

Thus, an application having 100% connectedness means that there is a path between any pair of components. On the other hand, 0% connectedness denotes a completely disconnected application. One notable observation is that the vast majority of the applications that are not completely disconnected have a connectedness of more than 60%.

Due to our concern with managing information flows, we asked "how well an approach, which groups strongly connected components together and monitors only the links between these groups, will work on existing applications?" Trying to answer this question, we devised a simple algorithm to partition the graph with the added capability of being able to cut a maximum of one link at a time. We apply a *Minimum Cut/Maximum Flow* algorithm on the application graph. If the Minimum Cut set has more than one link, we stop. Otherwise, we partition the graph by cutting this link, and then apply the algorithm recursively on the resulting parts. Fig. 3.4 shows the connectedness of the applications after this algorithm, which we call *MinCut1*, was applied.

The change was not dramatic, as for almost 90% of the applications the connectedness did not change. We also tried applying the same algorithm but with the ability to cut more than one link at a time. We call such an algorithm *MinCutX*, where X is the maximum number of cuts that can be applied in each iteration. The graphs in Fig. 3.4 show the result of applying the algorithm up to MinCut4. Again, the changes are still small due to the fact that most of the apps are well connected.

These results tell us that robustness of enforcement will not improve even if we try to more robustly enforce our DIFC policies at the component-level by run-time monitoring of potential information flow paths (e.g., using instrumentation), grouping tightly coupled components, or a combination of these two approaches. First, for about 17% of these applications (those with 0% connectedness), the enforcement is already robust. Second, the majority of these applications are well connected (connectedness $> 65\%$) and trying to monitor all the information flow paths will significantly degrade their performance or functionality.

3.5 Discussion and Conclusion

In order to better understand how our IFC enforcement mechanism will work at the component level, and to confirm the results of our experiment, we conducted a manual inspection of several open-source applications. This inspection led us to conclude that applications needed to be non-trivially modified in order to successfully develop a suitable security policy for existing Android applications at the component level, without sacrificing functionality. Moreover, we noticed that, overall, the need for such modification stems from the observation that the component boundaries are based on the GUI elements of the applications, and not on functionality. As an example, consider an application that has two Android activities, one to view a list of objects and the other to edit it. Both activities show advertisements using a third party library. Now if we need to assign permissions for reading and editing this list because the integrity of this list is sensitive, then we will need to remove the advertisements from the second activity. However, ideally, we

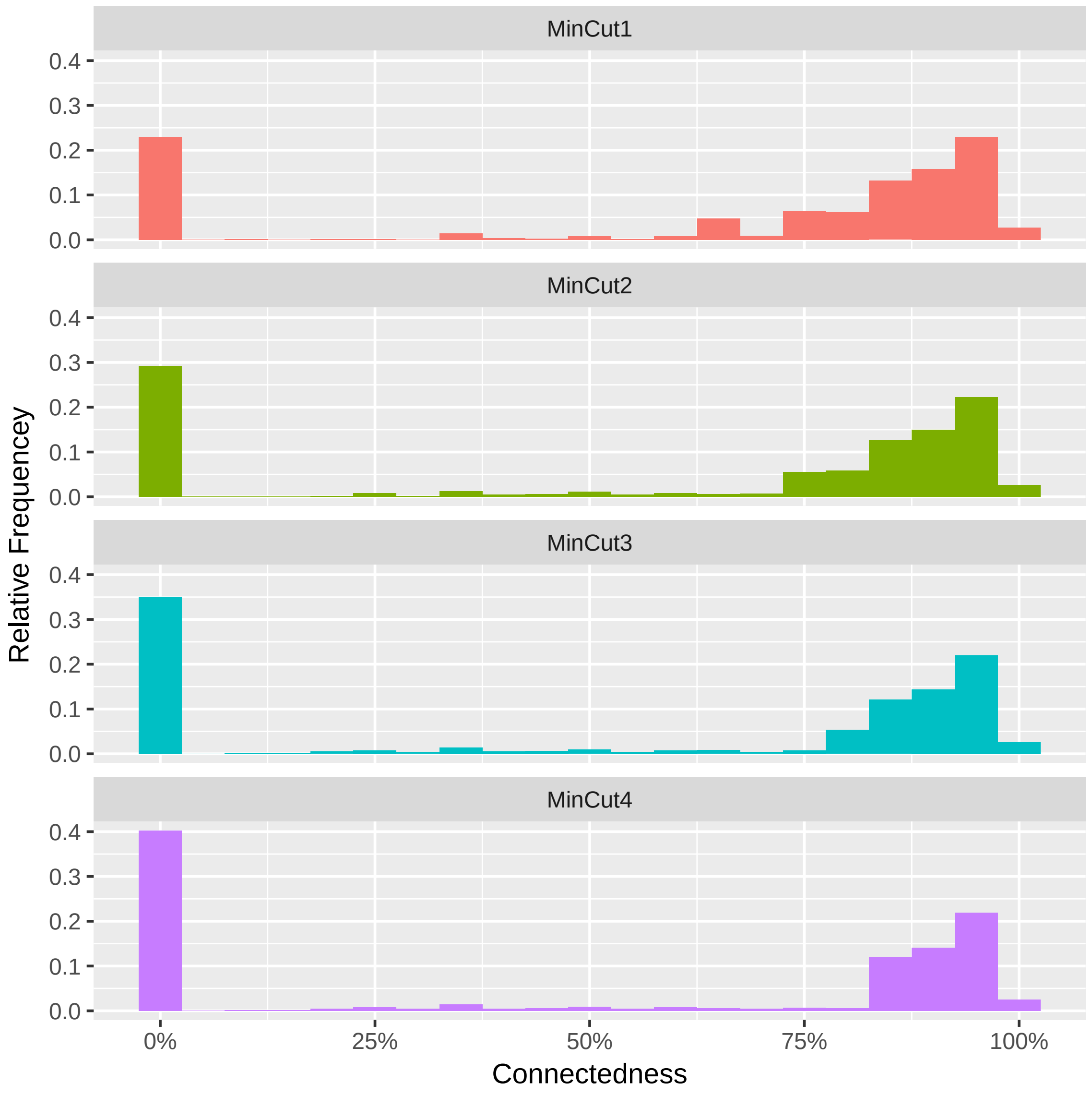

Figure 3.3: The connectedness of apps after applying the MinCut1 through MinCut4 algorithms.

want to assign separate permissions for the advertisement library and each of those activities while keeping the ability to show the advertisement on all the screens of an application. While the situation is better for non-activity components, we found that, for example, applications tend to implement many functionalities, which need long running computations in a single Android service component. This implementation means that this single service will be assigned many permissions that are not all needed for each single functionality.

While these observations do not imply that implementing suitable security policies at the component level is unattainable, they do indicate that significant manual modifications to the design of existing applications are needed. But even with such modification, implementing security policies can be awkward and the results might be sub-optimal. This finding reinforces the need to incorporate security consideration into the software development cycle from the beginning.

The other goal of these manual reviews was to assess the accuracy of our results. In our limited review (of 8 apps), we found that in the instances where the tool generated a warning of a potential flow through static fields, there was an actual data structure that was being accessed by multiple components in the app, i.e., no false positives. While this is not significant given the sample size, it does provide a degree of assurance in our results. A frequent example of such shared data structure was the use of Java's hashtables as key-value stores for data such as application settings and cached results. Another prevalent source of unmonitored flows is the use of advertisements libraries that are included in almost all the app's activities.

From the results of our analysis and the observations above, we conclude that for current Android applications, component-level enforcement is only suitable for a small portion of them. This portion includes the 17% which don't communicate internally using static fields and applications that have sparsely connected components, i.e., few (1-5) links between the components. For these applications, a (meaningful) component-level policy is likely enforceable without affecting their performance or functionality. However, the majority of applications are well connected such that application-level only policies are more appropriate.

For these applications, even if we group large sets of well connected components together for policy enforcement purposes, and monitor the rest of the direct communications, it is unlikely that the policy for those groups will be much different from the policy for the whole application, making the component-level policy redundant.

3.6 Related Work

There have been many works to study Android apps through the use of static analysis [51]. These works can be divided into two general categories: 1) targeted anaylsis to find a specific kind of vulnerabilities [10, 33, 34, 35, 52, 53, 54, 55, 56, 57, 58, 59, 60], or 2) frameworks that can be extended or modified to perform more specific analysis [61, 62, 63, 64].

The first body of work uses static analysis to detect specific secrity vulnerbility in Android apps. The first of such works is ComDroid, which uses static analysis to detect common misuses of Android intents and inter-app communications that could expose an app to attacks [10]. EPICC, which targets vulnerabilities similar to ComDroid, improves the accuracy of the analysis [52]. COPES detects apps with unused permissions that could be exploited by malware [53]. MalloDroid looks at ususes of SSL/TLS connections in Android apps and detects potentially vulnerable implementations [54]. Finally, ADrisk detects the use of risky advertisements libraries [55]. Our static analysis falls into this category. Unfortunately, these works are too specific to be used for our goal of detecting unmonitored information flows.

The second body of work either creates generalized analysis frameworks, that we might be able to specialize for our purposes [61], or works that are not truly extensible but detect general information flows and could theoretically be appropriated to detect the flows we are targeting [62, 64]. Both Scandal and FlowDroid, detect information flows from a list of sources (e.g., GPS location or IMEI number) to a list of sinks (e.g., networking APIs). Amandroid is a more general framework that can handle both control and data flows between components and is explicitly extensible [61]. The problem with these approaches is that they can take extremely long times to process apps. For example, apps that took about 10 to 20 minutes to process using our tool either finished after more than an hour or did not finish within two hours. Such times were not feasible for our goal of analyzing hundreds of apps.

Most of the tools mentioned above have a pre-specified set of sources and sinks that are considered security critical, and are focused on privacy. In contrast, our system allows for more expressive policies for both secrecy and integrity. Moreover, static analysis, by definition, does not consider actual run-time state and thus must disallow more apps than necessary. This limitation has the potential of offsetting the benefits of the extra computational power of static analysis that are not limited by running on users' smartphones. Moreover, these static analysis tools are not adequate for the purpose of our analysis in this chapter. While our analysis need to only consider information flows outside Android's APIs, these tools usually consider all potential flows. Moreover, in order to enhance the precision (i.e., decrease false positives) most of these tools track information at the instruction level at the cost of either efficiency or soundness. Ours only track information at the component level, which is more efficient[8], and does not compromise on soundness (except possibly at the level of the points-to analysis, which is provided by the framework and is an orthogonal problem).

[8]Analysis on most apps finish in less than 20 minutes. In comparison, with FlowDroid, which is also based on the Soot framework, only 7 out of a 24 apps finished within 2 hours [64].

Chapter 4

Verification of Security Properties

This chapter details our effort to formally verify the security properties of the system introduced in Ch. 2. This verification supports our thesis claim that our approach can provide provable security guarantees. By formally proving that a piece of software has specific security properties, we provide a much higher level of assurance in its security than informal arguments (e.g., manual reviews, use of ad hoc static analysis tools, and testing). The standard security property that IFC systems provide is noninterference, which guarantees that information flow is tightly controlled by ensuring that information from a part with a security label can only flow to parts with an equal or strictly higher label.

Sec. 4.1 introduces a formal model of the system, along with a proof that the model satisfies the noninterference property. In Sec. 4.2, we present a different simplified model with a detailed account of our work to encode this simplified model. In Sec. 4.3, we prove that the simplified model satisfies the noninterference property. Finally, we define the versioned-labels noninterference property, and prove it for the model, in Sec. 4.4.

The Coq model and proof We used the Coq proof assistant to encode the simplified model from Sec. 4.2, and then prove the noninterference theorem from Sec. 4.3. While the model described here and the Coq encoding should agree perfectly, there are some differences. The main reason for these difference is to make the presentation in the dissertation clearer. For example, the judgments names and argument orders can be different. Also, some of the trivial implementation details are omitted in the dissertation, such as the implementation of the label lookup judgment $(aid, iid)@lm$ `HasLabel` (sl, dl). You can find a listing of the Coq model encoding in Appx. A. Sec. 4.3 covers the main versioned-labels noninterference theorem, the judgements and function used to define it, and some of the high level lemmas used to prove it, omitting the reset of the proof. The actual proof is much longer at more than 12,000 lines of Coq source code.

The modeling gap Our model of Android, by design, omits certain (low level) details of Android; otherwise formal verification would be much more difficult. While we tried to design our model to be as faithful as possible while keeping the complexity manageable, there is potentially a lack of correspondence between the model and the actual Android implementation. Bridging this gap is out of the scope of this thesis. A similar gap exists in other formal verification research work and for similar reasons. For example, Leroy's work on compiler verification omits the verification of the assembler, linker, and parser [65]. Verification of cryptographic protocols also usually depends on idealized model that are verified in separate works [66]. One approach to tackle this problem is to design successively more detailed models of Android until we have an exact model of the actual implementation, where each model verifiably refines the previous one. This approach was used to verify the seL4 microkernel [67]. Our model of Android, because it was specifically designed as such, is closer to the actual implementation than the general models on which most the IFC security properties are designed. For example, some works defines noninterference for the generic, but powerful, process algebra models [68, 69]. In a more related work, noninterference is defined for a reactive system model, where apps are encoded with event-driven loops similar to Android, that otherwise lacks other Android specific features. In our model, we tried to capture important features of Android such as the separation of apps into single- and multi-instance, the intent driven nature of apps execution, and shared app state.

4.1 A π-Calculus Model and Its Noninterference Proof

In order to show that our enhanced policy specification scheme and enforcement mechanism for Android prevent information leakage and privilege escalation attacks, we prove a noninterference theorem. We start with a faithful process calculus model of Android's run-time and the enforcement mechanism in Sec. 4.1.1. Then, in Sec. 4.1.2, we define well-behaved configurations of the system model, along with a projection function that removes "high" labeled parts of the configurations (i.e., the parts that are not accessible by an attacker). We then show that any well-behaved configuration is equivalent (in the trace-based sense), up to the attacker observable actions, to a projection of the same configuration. Full details can be found in our technical report [70].

4.1.1 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 applications, persistent state within component instances, and shared state within an application.

To facilitate the proof of noninterference, our model also contains some elements that are helpful for proofs but are not part of or necessary for the enforcement mechanism.

4.1.1.1 Labels and label operations

As described in Sec. 2.2, we use labels to express information-flow policies and track flows at run time; here we define them formally, and in sufficient detail to facilitate our proofs. Labels express information-flow policies and are used to track flows at run time. A *label* is composed of sets of *tags*. We assume a universe of secrecy tags $\mathcal{S}$ and integrity tags $\mathcal{I}$. 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.

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

A simple label κ is a pair of a set of secrecy tags σ drawn from $\mathcal{S}$ and a set of integrity

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	$(Q(\sigma), Q(\iota), \delta) \uplus_{rz} (\sigma', \iota') = (Q(\sigma \cup \sigma'), Q(\iota \backslash \iota'), \delta)$
Declassify	$(C(\sigma), C(\iota), \delta) \uplus_d \delta_1 = (C(\sigma \backslash \{t \mid (-t) \in \delta_1\}), C(\iota \cup \{t \mid (+t) \in \delta_1\}), \delta)$
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 4.1: Summary of label operations.

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 components it can send intents to. Consequently, the partial order over simple labels is defined as follows:

$$(\sigma_1, \iota_1) \sqsubseteq (\sigma_2, \iota_2) \quad \text{iff} \quad \sigma_1 \subseteq \sigma_2, \text{ and } \iota_2 \subseteq \iota_1$$

Secrecy and integrity labels are annotated with C for concrete labels or F for floating labels, as described in Sec. 2.2. A process label K is composed of a secrecy label, an integrity label, and a set of declassification capabilities δ. An element in δ is of the form $-t_s$, where $t_s \in \mathcal{S}$, 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. Deleting se crecy tags or adding integrity tags allows the component more freedom to send intents to other components.

We define operations on labels to allow the reference monitor to compare labels, compute the effects of declassification, and instantiate floating labels (Fig. 4.1).

An *AM erasure* function K^- is used by the activity manager to reduce process labels to simple labels that can easily be compared to determine whether a call should be allowed. This function removes the declassification capabilities from K, and reduces a floating secrecy label to the top secrecy label. This captures the idea that declassification capabilities are not relevant to label comparison, and that a callee's floating secrecy label will never cause a call to be denied. The *PF* erasure function K^* is used in defining noninterference, and is explained in Sec. 4.1.2.

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 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.

4.1.1.2 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.

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

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.

Bellow we define standard structural congruence and labeled transition rules for the calculus.

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

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

Labeled transition rules A complete list of the labeled transition rules is shown in Fig. 4.2. 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 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. □

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

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

$$\frac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{P \,|\, Q \xrightarrow{\alpha} P \,|\, Q'} \qquad \frac{P \xrightarrow{\mathsf{in}\,x(y)} P' \qquad Q \xrightarrow{\mathsf{out}\,x(y)} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'}$$

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

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

$$\frac{P \xrightarrow{\mathsf{out}\,x(y)} P' \qquad x \neq y}{\nu y.P \xrightarrow{\nu y.\mathsf{out}\,x(y)} P'} \qquad \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(id) = \emptyset}{id[\,P\,] \xrightarrow{\alpha} id[\,P'\,]} \qquad \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \,|\, !P}$$

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

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

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

Figure 4.2: Labeled Transition Rules for the Core Calculus

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.

4.1.1.3 A model of android and our enforcement mechanism

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 and their label map A large part of the modeling effort is spent on ensuring that the process model faithfully reflects the life-cycles of the application and components, which is crucial to capturing information flows through persistent states within or across the life-cycles. The reference monitor maintains a label map Ξ, which reflects these life-cycles. Android supports single- and multi-instance components. Once created, a single-instance component can receive multiple calls; the instance body shares state across all these calls. A fresh instance of a single-instance component is created only when the previous instance has exited and the component is called again. 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 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 Fig. 4.3. We summarize special-purpose channels in Fig. 4.5. We delay explaining the label contexts $\ell[...]$ that surround processes until Sec. 4.1.2—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

$$\begin{array}{lll}
\textit{Parameters} & \mathit{arg} & = \mathit{aid}, \mathit{cid}, I, c_{AI}, c_{sv}, c_{ls}, c_{nI}, c_{lock}, \mathit{rt} \\
\textit{Comp. loop body} & CB(\mathit{arg}, s) & = \mathsf{in}\, c_{nI}(I).\langle \mathsf{out}\, I(\mathsf{self})\rangle.A(\mathit{arg}, \{s\}) \\
\textit{Comp. event loop} & CE(\mathit{arg}) & = !(\mathsf{in}\, c_{ls}(s).CB(\mathit{arg}, s)) \\
\textit{Component} & CP(\mathit{aid}, \mathit{cid}, c_{AI}, c_{sv}) & = !(\mathsf{in}\, \mathit{aid}\cdot \mathit{cid}\cdot c_{cT}(\_{=}c_{AI}, I, c_{nI}, c_{lock}, \mathit{rt}). \\
 & & \quad (c_{AI}, c_{nI})[\nu c_{ls}.(\mathsf{out}\, c_{ls}(\sigma_0).\mathsf{out}\, c_{nI}(I)\ '\vert'\ CE(\mathit{arg}))] \\
\textit{Shared store body} & \mathit{SVBody}(c_{svL}, c_{sv}) & = \mathsf{in}\, c_{sv}(\mathsf{rd}, r).\mathsf{out}\, r(x).\mathsf{out}\, c_{svL}(x) + \mathsf{in}\, c_{sv}(\mathsf{wt}, v).\mathsf{out}\, c_{svL}(v) \\
\textit{Shared store} & SV(c_{svL}, c_{sv}) & = !(\mathsf{in}\, c_{svL}(x).\mathit{SVBody}(c_{svL}, c_{sv})) \\
\textit{Application body} & \mathit{AppBody}(\mathit{aid}, c_{AI}) & = \nu c_{svL}.\nu c_{sv}.\mathsf{out}\, c_{svL}(s_0).(SV(c_{svL}, c_{sv})\ '\vert' \\
 & & \quad (c_{AI}, \mathit{cid}_1)[CP_1(\mathit{aid}, \mathit{cid}_1, c_{AI}, c_{sv})]\ '\vert'\ \cdots\ '\vert' \\
 & & \quad (c_{AI}, \mathit{cid}_n)[CP_n(\mathit{aid}, \mathit{cid}_n, c_{AI}, c_{sv})]) \\
\textit{Application} & \mathit{App}(\mathit{aid}) & = \mathit{aid}[!(\mathsf{in}\, \mathit{aid}\cdot c_L(c_{AI}).c_{AI}[\mathit{AppBody}(\mathit{aid}, c_{AI})])]
\end{array}$$

Figure 4.3: Encoding of applications and components.

$$\begin{array}{l}
\textit{Component body}\ \ A(\mathit{cid}, \mathit{aid}, I, c_{AI}, c_{sv}, c_{ls}, c_{nI}, c_{lock}, \mathit{rt}, V) ::= \\
\quad \ldots \\
\quad\ \vert\ \mathsf{out}\, t_m(\mathsf{raiseA}, \mathit{aid}, c_{AI}, \delta, \iota).A(\cdots) \\
\quad\ \vert\ \mathsf{out}\, t_m(\mathsf{raiseC}, \mathit{cid}, c_{nI}, \delta, \iota).A(\cdots) \\
\quad\ \vert\ \mathsf{out}\, t_m(\mathsf{dclassifyA}, c_{AI}, \delta).A(\cdots) \\
\quad\ \vert\ \mathsf{out}\, t_m(\mathsf{dclassifyC}, c_{nI}, \delta).A(\cdots) \\
\quad\ \vert\ \mathsf{out}\, a_m(\mathsf{call}_I, \mathit{rt}, \mathit{aid}, c_{AI}, \mathit{cid}_{ce}, I)\ '\vert'\ A(\cdots) \\
\quad\ \vert\ \mathsf{out}\, a_m(\mathsf{call}_E, \mathit{rt}, c_{AI}, c_{nI}, \mathit{aid}_{ce}, \mathit{cid}_{ce}, I)\ '\vert'\ A(\cdots) \\
\quad\ \vert\ \mathsf{out}\, a_m(\mathsf{exitA}, \mathit{aid}, \mathit{cid}, c_{AI}, c_{nI}, \mathit{rt}, e) \\
\quad\ \vert\ \mathsf{out}\, a_m(\mathsf{exitC}, \mathit{aid}, \mathit{cid}, c_{AI}, c_{nI}, \mathit{rt}, e) \\
\quad\ \vert\ \mathsf{out}\, c_{ls}(e).\mathsf{out}\, c_{lock}()\ \vert\ \mathsf{out}\, c_{sv}(\mathsf{wt}, e).A(\cdots) \\
\quad\ \vert\ \nu r.\mathsf{out}\, c_{sv}(\mathsf{rd}, r).\mathsf{in}\, r(x).A(\cdots, V \cup \{x\})
\end{array}$$

Figure 4.4: Partial encoding of component bodies.

noninterference; we will revisit this in Sec. 4.1.2. The parameters *arg*, together with s, appear free in A. They are generated by outer-layer processes, which we explain next. The event loop $CE(\mathit{arg})$ 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 $CP(\mathit{aid}, \mathit{cid}, c_{AI}, c_{sv})$ is launched from a designated creation channel $\mathit{aid}\cdot\mathit{cid}\cdot c_{cT}$. The message it receives on the creation channel is a tuple $(\_{=}c_{AI}, I, c_{nI}, c_{lock}, \mathit{rt})$ 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

	ID	launch ch.	instance ID	instance-specific channels
App	aid	$aid{\cdot}c_L$	c_{AI}	c_{svL}: shared variable launch channel c_{sv}: shared variable read-write channel
Comp.	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 4.5: 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.

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 the parallel composition of a shared state SV and components $CP_i(aid, cid_i, c_{AI}, c_{sv})$. Each application has a designated launch channel $aid{\cdot}c_L$. The channel c_{AI}, passed as an argument to the launch channel, serves as a unique identifier for an application instance. Once an application is launched, it 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 Fig. 4.4, 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 $\mathsf{out}\, a_m(\mathsf{call}_I, \mathit{rt}, \ldots)$ 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: ($\mathsf{out}\, c_{ls}(e).\mathsf{out}\, c_{lock}()$). 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. One reason for using such a lock is that an instance can change its label, and the activity manager needs to know an instance's most up-to-date label before sending the new intent.

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:

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

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.

The encoding of the label manager is shown in Fig. 4.6. 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.

The encoding of the label manager is shown in Fig. 4.6. 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.

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 instance.

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.

Note that, as we discussed in Sec. 2.3, the raise operation has to raise both the static and the effective label of single-instance components (or applications). In addition, we don't allow floating instances to raise their labels. 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.

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: $\mathsf{AM} = !(AM_I + AM_E + AM_{EX} + AM_R)$. 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.

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 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 Fig. 4.7 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 other parts of the encoding that deals with returns, exits, and cross-application calls where we need to enforce both the application-level policies and the component-level policies are illustrated in Figs. 4.8, 4.9, and 4.10.

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.

$$\mathsf{init} = \mathsf{out}\, t_t(\Xi^0).init[\mathsf{out}\, a_m(\mathsf{call}_E, \mathsf{NONE}, init_A, init_C, aid^0, cid^0, I^0)]$$

$init_A$ and $init_C$ represent the initial process's application instance and component instance. These are not meaningful except that Ξ^0 assigns them the lowest secrecy label and highest integrity label so that the initial process can start any other process.

We model the whole system as follows:

$$S = \mathsf{TM} \,|\, \mathsf{AM} \,|\, App_1(aid_1) \,|\, \cdots \,|\, App_n(aid_n) \,|\, \mathsf{init}$$

4.1.2 Noninterference

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

4.1.2.1 Choice between trace & bisimulation-based equivalence

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

Our decision about which notion of process equivalence to use for our noninterference definition is driven by the functionality required of the system so that practically reasonable policies can be implemented. As 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 (see our technical report [70] for an example).

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

4.1.2.2 High and low components

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

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

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

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

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

Now, given the label map Ξ, let $\Xi\langle c\rangle$ denote the label associated with a channel name c in Ξ. We say that a process of the form $aid[...(c_{AI}, c_{nI})[P]...]$ is a low process with regard to κ_L if $(aid, ((\Xi\langle c_{AI}\rangle)^*, (\Xi\langle c_{nI}\rangle)^*) \sqsubseteq_{aid_c} \kappa_L$; otherwise, it is a high process. Please see our tech report for a formal definition and additional details [70].

The function K^* (Fig. 4.1) removes the declassification capabilities in K, and reduces floating integrity labels to the lowest integrity label (on the lattice). This is because a call to a component with a floating integrity label may result in a new instance with a low integrity label, a low event observable by the attacker; hence, a floating component should always be considered a low component.

4.1.2.3 Traces

The actions relevant to our noninterference definitions are intent calls received by an instance, since the only explicit communication between the malicious components (applications) and other parts of the system is via intents. We model intents I as channels. The encoding of components includes a special output action $\langle \mathsf{out}\, I(\mathsf{self})\rangle$ right after the component receives a new intent (Fig. 4.3). This outputs to the intent channel the current labels of the component, denoted by self. Traces consist of these outputs $(\mathsf{out}\, I(aid, (kA, kC)))$, which contain information about both what the recipient has learned and the security label of the recipient. We call such an action low, if $(aid, (kA, kC)) \sqsubseteq_{aid_c} \kappa_L$, and high otherwise.

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

4.1.2.4 Noninterference

We define the projection of traces $t|^{aid_c}_{\kappa_L}$, which removes all high actions from t. The function $\mathsf{projT}(\Xi; \kappa_L; aid_c)$ removes from Ξ mappings from IDs or channel names to high labels. Similarly, $\mathsf{proj}(P, \kappa_L, aid_c, \Xi)$ removes high components, applications, and instances from P. The resulting configuration is the low system that does not contain secrets or sensitive interfaces.

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

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

Theorem 1 (Noninterference)**.**

For all κ_L, for all applications App$(aid_1), \cdots,$ App(aid_n),

given a aid_c ($aid_c = aid_i$), $i = 1 \dots n$, whose components do not access the shared variable,

let $S = A_M \,|\, T_M \,|\,$ App$(aid_1), \cdots,$ App(aid_n) be the initial system configuration, $S \Longrightarrow_A S'$,

$S' = A_M \,|\, T_M \,|\, \nu\vec{c}.(T_{MI}(\Xi) \,|\, AC(aid_c) \,|\, S'')$, where $T_{MI}(\Xi)$ is an instance of the tag manager, Ξ is the current label map, and $AC(aid_c)$ is an active launched instance of aid_c,

let $\Xi' = \mathsf{projT}(\Xi; \kappa_L; aid_c)$,

$S_L = A_M \,|\, T_M \,|\, \nu\vec{c}'.(T_{MI}(\Xi') \,|\, \mathsf{proj}(AC(aid_c) \,|\, S'', \kappa_L, aid_c, \Xi'))$

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

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

4.1.2.5 Proof sketch

Here we only give a general idea about our prove of Theorem 1. For more details, please refer to [70]. In order to prove Theorem 1, we prove that there is a simulation relation between the valid configurations of the system and the system with only low components (S and S_L, respectively).

The theorem then easily follow from the existence of such relation.

To prove that such simulation relation exists, we start by defining what are the stable (i.e., valid) configurations of the whole system by defining the stable configurations of the different parts of the system, i.e., the activity manager, label manager, components, etc. Then we define (Android-specific) transition rules for those stable configurations, and prove progress and preservation, ensuring the well-definedness of the these transition relations.

After that, we define a projection function, $\mathsf{proj}(P, \Xi, aid_c, \kappa_L)$, on those configurations, parameterized by κ_L and aid_c, which removes high process (processes whose behavior should not effect the attacker) and dead code (process guarded by channel names that do not exist in the label map). These projections are based on the label contexts. Similar projection functions for label maps are also defined. We prove that these projections preserver the stability of the configurations. In addition to the projection functions, we also define two relations between two label maps: $\Xi \precsim^{\kappa_L}_{aid_c} \Xi'$ and $\Xi \precsim^{\kappa_L}_{\mathsf{comp}} \Xi'$. Intuitively, these relations imply that although Ξ is the label map for the system with high components, and Ξ' is the label map for the system without high components, Ξ and Ξ' agree on low mappings. 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.

Operations, such as calling, declassification, and exiting, will cause the label map to be updated. We define classification of these label update, which either classify them as *high updates*, or *low updates*. Update classification is based on the effective label of the instance of the updated application (or component). The main idea is that high updates only affect high components. In addition, 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 1. $P \leq Q \overset{def}{=} Q = P \,|\, Q_1$

Finally, we 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 $P \neq l[P']$.

Now we are ready to define a candidate simulation relation:

Definition 2. *$P \ \mathcal{R}_{(aid_c, \kappa_L)} \ Q$ iff*

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

First, we utilize several key observations (lemmas):

- 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.
- 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.
- 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.
- 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.

After proving the lemmas above, we then proceed to prove that $\mathcal{R}$ is a Γ-simulation relation. 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).

1. Transitions that do not change the labels of any components:
   (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:
   (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.
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.
   (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.

(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.

```
$\mathit{Label\ Mngr}$ TM =
!( in $t_t(x)$.
      in $t_m(\mathsf{lookUp}, id, r)$. out $r(x(id))$.out $t_t(x)$
   + in $t_m(\mathsf{upd}, u)$.out $t_t(\mathit{updateMap}(x, u))$
   + in $t_m(\mathsf{cleanA}, aid, cid, c_{AI}, c_{nI})$.
      case $x(aid)$of
      $\mathsf{L}(c_{AI}, k) \Rightarrow$ out $t_t(x[aid \mapsto \mathsf{onEx}(c_{AI}, k)])$.
                        out $t_m(\mathsf{cleanC}, aid, cid, c_{AI}, c_{nI})$
      $|\_$             $\Rightarrow$ out $t_m(\mathsf{cleanC}, aid, cid, c_{AI}, c_{nI})$
   + in $t_m(\mathsf{cleanC}, aid, cid, c_{AI}, c_{nI})$.
      case $x(aid.cid)$of $\mathsf{SL}(c_{AI}, c_{nI}, c_{lock}, kC_s) \Rightarrow$
         if $x(c_{AI}) = (k, \{c_{nI}\}) \wedge x(aid) = \mathsf{onEx}(c_{AI}, kA_s)$
         then out $t_t(x\ [aid.cid \mapsto S(kC_s)]$
                        $[aid \mapsto \mathsf{U}(kA_s)] \backslash c_{AI} \backslash c_{nI})$
         else out $t_t(x[aid.cid \mapsto S(kC_s)]$
                       $[c_{AI} \mapsto x(c_{AI}) \backslash c_{nI}] \backslash c_{nI})$
      $|\ \mathsf{M}(kC_s) \Rightarrow$if $x(c_{AI}) = (kA_d, \{c_{nI}\})$
                       $\wedge\ x(aid) = \mathsf{onEx}(c_{AI}, kA_s)$
                      then out $t_t(x[aid \mapsto \mathsf{U}(kA_s)] \backslash c_{AI} \backslash c_{nI})$
                      else out $t_t(x[c_{AI} \mapsto x(c_{AI}) \backslash c_{nI}] \backslash c_{nI})$
   + in $t_m(\mathsf{dclassifyA}, c_{AI}, \delta)$.
      if $dL(x(c_{AI})) \supseteq \delta$
      then out $t_t(x[c_{AI} \mapsto x(c_{AI}) \uplus_d \delta])$
      else out $t_t(x)$
   + in $t_m(\mathsf{dclassifyC}, c_{nI}, \delta)$.
      if $dL(x(c_{nI})) \supseteq \delta$
      then out $t_t(x[c_{nI} \mapsto x(c_{nI}) \uplus_d \delta])$
      else out $t_t(x)$
   + in $t_m(\mathsf{raiseA}, aid, c_{AI}, \sigma, \iota)$.
      if $\mathit{floating}(x\langle aid \rangle)$
      then out $t_t(x)$
      else out $t_t(x[aid \mapsto x(aid) \uplus_{rz} (\sigma, \iota)]$
                     $[c_{AI} \mapsto x(c_{AI}) \uplus_{rz} (\sigma, \iota)])$
   + in $t_m(\mathsf{raiseC}, aid, cid, c_{nI}, d)$.
      case $x(aid{\cdot}cid)$ of $\mathsf{M}(k) \Rightarrow$
        out $t_t(x[c_{nI} \mapsto x(c_{nI}) \uplus_{rz} (\sigma, \iota)])$
      $|\ \mathsf{SL}(\_) \Rightarrow$
        if $\mathit{floating}(x\langle aid{\cdot}cid \rangle) \vee \mathit{floating}(x\langle aid \rangle)$
        then out $t_t(x)$
        else out $t_t(x[aid{\cdot}cid \mapsto x(aid{\cdot}cid) \uplus_{rz} (\sigma, \iota)]$
                       $[c_{nI} \mapsto x(c_{nI}) \uplus_{rz} (\sigma, \iota)])$
```

Figure 4.6: Encoding of the label manager.

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

Figure 4.7: Encoding of the activity manager subprocess that handles calls between components of the same application.

Activity Manager Exit AM_{EX} =

```
24    in $a_m(\mathsf{exitA}, kA_{ce}, kC_{ce}, aid_{ce}, cid_{ce}, c_{AI_{ce}}, c_{nI_{ce}}, rt, I)$.
25       out $t_m(\mathsf{cleanA}, aid_{ce}, cid_{ce} c_{AI_{ce}}, c_{nI_{ce}})$.
26       case $rt$ of NONE $\Rightarrow \mathbf{0}$
27       | SOME$(aid_{cr}, c_{AI}, c_{nI}, c_{lock}) \Rightarrow$
28          $(aid_{ce}, (kA_{ce}, kC_{ce}))[$out $a_m(\mathsf{return}, kA_{ce}, kC_{ce}, aid_{ce},$
29                                              $c_{AI_{ce}}, aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)]$
30 +  in $a_m(\mathsf{exitC}, kA_{ce}, kC_{ce}, aid_{ce}, cid_{ce}, c_{AI_{ce}}, c_{nI_{ce}} rt, I)$.
31       out $t_m(\mathsf{cleanC}, aid_{ce}, cid_{ce}, c_{AI_{ce}}, c_{nI_{ce}})$.
32       case $rt$ of NONE $\Rightarrow \mathbf{0}$
33       | SOME$(aid_{cr}, c_{AI}, c_{nI}, c_{lock}) \Rightarrow$
34          $(aid_{ce}, (kA_{ce}, kC_{ce}))[$out $a_m(\mathsf{return}, kA_{ce}, kC_{ce}, aid_{ce},$
35                                              $aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)]$
```

Activity Manager Return AM_R =

```
36  in $a_m(\mathsf{return}, kA_{ce}, kC_{ce}, aid_{ce}, c_{AI_{ce}}, aid_{cr}, c_{AI}, c_{nI}, c_{lock}, I)$.
37     in $c_{lock}()$.
38     if $aid_{ce} = aid_{cr}$
39     then $\nu r$.out $t_m(\mathsf{lookUp}, c_{nI}, r)$.in $r(s)$.
40            case $s$ of $(\_, kC_d)$ $\Rightarrow$
41              if $kC_{ce}^{-} \sqsubseteq kC_d^{-}$
42              then $(aid_{cr}, (c_{AI}, c_{nI}))[$ out $c_{nI}(I)$ $]$
43              else $(aid_{cr}, (c_{AI}, c_{nI}))[$ out $c_{lock}()$ $]$
44            $| \Rightarrow \mathbf{0}$
45     else $\nu r$.out $t_m(\mathsf{lookUp}, c_{AI}, r)$.in $r(s1)$.
46           out $t_m(\mathsf{lookUp}, c_{nI}, r)$.in $r(s2)$.
47           case $(s1, s2)$ of $((\mathsf{DA}(kA_d), \_), (\_, kC_d)) \Rightarrow$
48             if $kA_{ce}^{-} \sqsubseteq kA_d^{-} \wedge kA_{ce}^{-} \sqsubseteq kC_d^{-}$
49               $\wedge\, kC_{ce}^{-} \sqsubseteq kA_d^{-} \wedge kC_{ce}^{-} \sqsubseteq kC_d^{-}$
50             then $(aid_{cr}, (c_{AI}, c_{nI}))[$ out $c_{nI}(I)$ $]$
51             else $(aid_{cr}, (c_{AI}, c_{nI}))[$ out $c_{lock}()$ $]$
52           $| \Rightarrow \mathbf{0}$
```

Figure 4.8: Encoding of the subprocesses of the activity manager that handle exit and return calls.

Activity Manager Inter-app Call AM_E =

```
53 in $a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)$.
54    $\nu r$.out $t_m(\mathsf{lookUp}, aid_{ce})$.in $r(s1)$. out $t_m(\mathsf{lookUp}, aid_{ce}{\cdot}cid_{ce}, r)$.in $r(s2)$.
55       case $(s1, s2)$ of $(\mathsf{NE}, \_) \Rightarrow \mathbf{0} \mid (\_, \mathsf{NE}) \Rightarrow \mathbf{0}$
56       $\mid (\mathsf{U}(kA_{ce}), \mathsf{S}(kC_{ce}))$
57          $\Rightarrow$ if $kA_{cr}^- \sqsubseteq kA_{ce}^- \wedge kA_{cr}^- \sqsubseteq kC_{ce}^- \wedge kC_{cr}^- \sqsubseteq kA_{ce}^- \wedge kC_{cr}^- \sqsubseteq kC_{ce}^-$
58             then $\nu c_{AI}.\nu c_{nI}.\nu c_{lock}$.
59                   out $t_m(\mathsf{upd}, \{\, (aid_{ce}, \mathsf{L}(kA_{ce}, c_{AI})), (aid{\cdot}cid_{ce}, \mathsf{SL}(c_{nI}, kC_{ce})),$
60                                        $(c_{AI}, \mathsf{DA}(kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr}), \mathsf{LC}(\{c_{nI}\}))),$
61                                        $(c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\})$.
62                   $(aid_{ce}, c_{AI})[$out $aid_{ce}{\cdot}c_L(c_{AI})]$
63                   $\mid\ (aid_{ce}, (c_{AI}, c_{nI}))[$out $aid_{ce}{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, \mathsf{NONE})]$
64             else $\mathbf{0}$
65       $\mid (\mathsf{U}(kA_{ce}), \mathsf{M}(kC_{ce}))$
66          $\Rightarrow$ if $kA_{cr}^- \sqsubseteq kA_{ce}^- \wedge kA_{cr}^- \sqsubseteq kC_{ce}^- \wedge kC_{cr}^- \sqsubseteq kA_{ce}^- \wedge kC_{cr}^- \sqsubseteq kC_{ce}^-$
67             then $\nu c_{AI}.\nu c_{nI}.\nu c_{lock}$.
68                   out $t_m(\mathsf{upd}, \{\, (aid_{ce}, \mathsf{L}(kA_{ce}, c_{AI})),$
69                                        $(c_{AI}, \mathsf{DA}(kA_{ce} \lhd (kA_{cr} \uplus_M kC_{cr}), \mathsf{LC}(\{c_{nI}\}))),$
70                                        $(c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\})$.
71                   $(aid_{ce}, c_{AI})[$out $aid{\cdot}c_L(c_{AI})]$
72                   $\mid\ (aid_{ce}, (c_{AI}, c_{nI}))[$out $aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, rt)]$
73             else $\mathbf{0}$
74       $\mid (\mathsf{L}(c_{AI}, kA_{ce}), \mathsf{S}(kC_{ce}))$
75          $\Rightarrow$ out $t_m(\mathsf{lookUp}, c_{AI}, r)$.in $r(\mathsf{DA}(kA_d, \_))$.
76             if $kA_{cr}^- \sqsubseteq kA_{ce}^- \wedge kA_{cr}^- \sqsubseteq kC_{ce}^- \wedge kC_{cr}^- \sqsubseteq kA_{ce}^- \wedge kC_{cr}^- \sqsubseteq kC_{ce}^-$
77             then if $(\mathit{conrete}(kA_{ce}) \wedge\ kA_{cr}^- \sqsubseteq kA_d^- \wedge\ kC_{cr}^- \sqsubseteq kA_d^-\ )$
78                      $\vee(\mathit{floating}(kA_{ce}) \wedge\ kA_{cr}^- = kA_d^- \wedge\ kC_{cr}^- = kA_d^-\ )$
79                  then $\nu c_{nI}.\nu c_{lock}$.out $t_m(\mathsf{upd}, \{\, (c_{AI}, \{c_{nI}\}), (aid{\cdot}cid_{ce}, \mathsf{SL}(c_{nI}, kC_{ce})),$
80                                                   $(c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\})$.
81                       $(aid_{ce}, (c_{AI}, c_{nI}))[\,$out $aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, \mathsf{NONE})\,]$
82                  else $(aid_{cr}, (kA_{cr}, kC_{cr}))[\,$out $a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)\,]$
83             else $\mathbf{0}$
84       $\mid (\mathsf{L}(c_{AI}, kA_{ce}), \mathsf{SL}(c_{nI}, kC_{ce}))$
85          $\Rightarrow$ out $t_m(\mathsf{lookUp}, c_{AI}, r)$.in $r(\mathsf{DA}(kA_d, \_))$.
86             out $t_m(\mathsf{lookUp}, c_{nI}, r)$.in $r(\mathsf{DC}(c_{lock}, \_))$. in $c_{lock}()$.
87             out $t_m(\mathsf{lookUp}, c_{nI}, r)$.in $r(\mathsf{DC}(\_, kC_d))$.
88             if $kA_{cr}^- \sqsubseteq kA_{ce}^- \wedge kA_{cr}^- \sqsubseteq kC_{ce}^- \wedge kC_{cr}^- \sqsubseteq kA_{ce}^- \wedge kC_{cr}^- \sqsubseteq kC_{ce}^-$
89             then if $(\mathit{conrete}(kA_{ce}) \supset (kA_{cr}^- \sqsubseteq kA_d^- \wedge kC_{cr}^- \sqsubseteq kA_d^-))$
90                      $\wedge\ (\mathit{floating}(kA_{ce}) \supset (kA_{cr}^- = kA_d^- \wedge kC_{cr}^- = kA_d^-))$
91                      $\wedge\ (\mathit{conrete}(kC_{ce}) \supset (kA_{cr}^- \sqsubseteq kC_d^- \wedge kC_{cr}^- \sqsubseteq kC_d^-))$
92                      $\wedge\ (\mathit{floating}(kC_{ce}) \supset (kA_{cr}^- = kC_d^- \wedge kC_{cr}^- = kC_d^-))$
93                  then $(aid_{ce}, (c_{AI}, c_{nI}))[\,$out $c_{nI}(I)\,]$
94                  else $(aid_{ce}, (c_{AI}, c_{nI}))[\,$out $c_{lock}()\,]$
95                       $\mid (aid_{cr}, (kA_{cr}, kC_{cr}))[\,$out $a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)\,]$
96             else $(aid_{ce}, (c_{AI}, c_{nI}))[\,$out $c_{lock}()\,]$
```

Figure 4.9: Encoding of the subprocess of the activity manager that handles calls between applications (part 1 of 2).

```
97    $\mid (\mathsf{L}(c_{AI}, kA_{ce}), \mathsf{M}(kC_{ce}))$
98      $\Rightarrow \mathsf{out}\, t_m(\mathsf{lookUp}, c_{AI}, r).\mathsf{in}\, r(\mathsf{DA}(kA_d)).$
99        $\mathsf{if}\ \ kA_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kA_{cr}{}^- \sqsubseteq kC_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_{ce}{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kC_{ce}{}^-$
100       $\mathsf{then}\ \ \mathsf{if}\ \ (conrete(kA_{ce}) \wedge\ kA_{cr}{}^- \sqsubseteq kA_d{}^- \wedge\ kC_{cr}{}^- \sqsubseteq kA_d{}^-\ )$
101             $\vee(floating(kA_{ce}) \wedge\ kA_{cr}{}^- = kA_d{}^- \wedge\ kC_{cr}{}^- = kA_d{}^-\ )$
102           $\mathsf{then}\ \nu c_{nI}.\nu c_{lock}.\mathsf{out}\, t_m(\mathsf{upd}, \{\ (c_{AI}, \{c_{nI}\}),$
103                   $(c_{nI}, (c_{lock}, kC_{ce} \lhd (kA_{cr} \uplus_M kC_{cr})))\}).$
104              $(aid_{ce}, (c_{AI}, c_{nI}))[\ \mathsf{out}\, aid{\cdot}cid{\cdot}c_{cT}(c_{AI}, I, c_{nI}, c_{lock}, rt)\ ]$
105           $\mathsf{else}\ \ (aid_{cr}, (kA_{cr}, kC_{cr}))[\ \mathsf{out}\, a_m(\mathsf{call}_E, kA_{cr}, kC_{cr}, rt, aid_{cr}, aid_{ce}, cid_{ce}, I)\ ]$
106       $\mathsf{else}\ \mathbf{0}$
107   $\mid (\mathsf{onEx}(), \_) \qquad \Rightarrow \mathbf{0}$
```

Figure 4.10: Encoding of the subprocess of the activity manager that handles calls between applications (part 2 of 2).

4.2 A Simplified Model $\mathcal{M}(\mathcal{L})$

In this section we introduce a model of our system that is simpler than the initial, π-Calculus based, one from Sec. 4.1. This simpler, labeled-transition-system (LTS) based, model represents apps only and not their components. Moreover, we restrict the policy to concrete labels, i.e., no floating labels. We started our work with the model from Sec. 4.1, which was adequate for the purpose of formally verifying that the system is secure (in the noninterference sense). However, for the purposes of defining and proving the versioned-labels noninterference property, and introducing controlled declassification later (see Ch. 5), we wanted to start with a simpler model that will enable us to focus on those aspects without the distraction of all the details from the full model. These omitted details are important, and we intend to add them back to model in future work. Furthermore, the transition from π-Calculus to an LTS model makes it more convenient to encode the model and both the noninterference property and proof in Coq, mainly because we avoid encoding π-Calculus itself.

We start by defining the security lattice in Sec. 4.2.1. Next, we describe the language for apps' code in Sec. 4.2.2. Then, we detail the syntax and operational semantics of the whole system model in Sec. 4.2.3 and Sec. 4.2.4, respectively.

4.2.1 The Security Lattice

The security lattice represents the security level or labels in the systems and the relation between them. Actually, the lattice construct here is not a strict requirement, we only need a *partially ordered set*, or a *poset*. Lattices are a special kind of posets that guarantee the existence of a unique least upper bound and a unique greatest lower bound for any two elements in the set. However, we don't need such requirement for our system. Some IFC systems, such as JFlow [18], do require it because they use *meet* and *join* operations on labels and need the answer to be unique. Moreover, the tag-based labels described in Ch. 2 do form a lattice with the subset relation. For these reasons we use the name security lattice, or simply lattice, throughout this dissertation.

A lattice, $\mathcal{L}$, is a pair $(L, \sqsubseteq)$, where L is a set of elements, and $\sqsubseteq$ is partial ordering relation for the elements of L.

4.2.2 App Language

We define a simple imperative programming language to encode the applications in our system. This language has standard side-effect free *expressions* that are defined in Fig. 4.11. They include generic binary operations, variables, and and memory referencing. The only final values an expression can yield are booleans, integers, and memory references. Memory references are just encapsulated integers that index into memory. The `ARG` variable, is a special variable used to access the intent (i.e., argument) received by the app instance.

Next, we define *statements* (Fig. 4.12), which includes assignments, conditionals, memory allocation, and variable declaration. We did not include loops for simplicity, but they won't affect any of the later results since we don't try to prove termination for apps or progress-sensitive security [71]. In addition, we define *commands*, which are special statements that represent inter-app calls or actions that affect the app's label or life cycle.

Memory We model memory as a linear list of values. Memory locations, l, are natural numbers that index into this list. Once a new memory location is allocated, through the **`new`** statement, its type is set and cannot be changed.[1] Only values of the same type can be written to an allocated memory location. This memory model approximates how Java, the language in which Android apps are written, abstracts memory. The main difference between this memory model and Java is that Java's *garbage collector* reclaims allocations that are not needed anymore.

Operational Semantics We describe the semantics of the language using small-step operational semantics rules (Fig. 4.13). The substitution used in the operational semantics of

[1]This is enforced by the typing system.

$$
\begin{array}{ll}
id & ::= x \mid \texttt{ARG} \\
B & ::= \texttt{true} \mid \texttt{false} \\
N & ::= 0 \mid 1 \mid 2 \mid \ldots \\
val & ::= B \mid N \mid \texttt{LOC}\, N \\
exp & ::= id \mid val \mid exp \oplus exp \mid *(exp)
\end{array}
$$

Figure 4.11: App language syntax: expressions.

$$
\begin{array}{lll}
AppID & ::= & N \\
label & ::= & \langle \textit{ any label } l \textit{, such that } l \in L \,\rangle \\
cmd & ::= & \textbf{RAISE}\ label \mid \textbf{DECLASSIFY}\ label \mid \textbf{CALL}(AppID, exp, B) \\
 & & \mid \textbf{RETURN}\ \text{exp} \mid \textbf{EXIT}\ exp \\
stm & ::= & (stm;\ stm) \\
 & & \mid \texttt{if}\ exp\ \texttt{then}\ stm\ \texttt{else}\ stm \\
 & & \mid exp := exp \\
 & & \mid \texttt{new}\ x := exp\ \texttt{in}\ \{stm\} \\
 & & \mid cmd \mid \texttt{null}
\end{array}
$$

Figure 4.12: App language syntax: statements.

the **new** statement (and later in Fig. 4.23) is described in Fig. 4.14. Note that there are no rules at this level to describe the execution of commands, as those will be handled at a higher level (see Sec. 4.2.4.2). Commands are handled at a higher level because they require interactions with external constructs, such as the *label map*, or other apps or instances. The **RAISE** l and **DECLASSIFY** l commands are requests to change the instance's dynamic label to l. The **CALL**(aid, n, er) command sends an intent with argument n to the app whose ID is aid. The third argument, er, specifies if the calling instance is expecting a return or not. The difference between **RETURN** n and **EXIT** n is that the later will remove the instance from the system, while the former will not. Both of these commands will initiate a return call, with argument n, if it was expected.

Typing In order to assure that apps are well behaved, we require that all apps are type checked using the typing system specified in the Coq encoding (see Appendix A). This is a straight forward type system that follows the treatment by Pierce et al. [72]. For example, the typing system ensures that all memory accesses are of the correct type. We write $\Delta; \Gamma \vdash_s s$ to state that statement s is well-typed given variable type assignments Γ and memory type assignments Δ. We also write $\Delta \vdash_{mem} m$ to indicate that all the memory m contain values of the corresponding types in Δ. In addition, we prove that the language is *safe*, i.e., we prove preservation and progress.[2]

4.2.3 Modeling the Android System: Syntax

The model of the Android system includes four components: (1) a set of *apps*, (2) a set of app *instances*, (3) a *label map*, and (4) a *pending queue*. An app instance in the model represents an active app process in Android.

The apps set contains the actual code of each app in the system. The instances set contains all current app instances in the system. The label map associates with each app and each app instance their security labels and state. The pending queue contains all pending requests that could not have been processed immediately. The exact syntax for the system is in Fig. 4.15. Below we will explain the different parts of this syntax.

Lists We use the *list* construct to model various aspects of the system. To facilitate our modeling we will use a standard inductively defined abstract list type. Figure 4.16 summarizes the notation and operations we use for lists. In addition, when we write, for example, $Apps ::=$ `list` stm, it's a syntactic sugar for $Apps ::= [] \mid (stm::Apps)$. In our Coq modeling we use Coq's native `list` datatype.

App IDs App IDs are essentially integers which index into the apps set, the label map, and the instances set. All three constructs are lists that must have the same length. The length of these lists will never change throughout the execution of the system ensuring that app IDs consistently refer to the same app.

[2]The proof is in our Coq development.

Expressions

$$\frac{e_1/m \Longrightarrow_{\mathrm{e}} e_1'}{e_1 \oplus e_2/m \Longrightarrow_{\mathrm{e}} e_1' \oplus e_2} \qquad \frac{e_2/m \Longrightarrow_{\mathrm{e}} e_2'}{val \oplus e_2/m \Longrightarrow_{\mathrm{e}} val \oplus e_2'} \qquad \frac{val_1 \oplus val_2 = val}{val_1 \oplus val_2/m \Longrightarrow_{\mathrm{e}} val}$$

$$\frac{e/m \Longrightarrow_{\mathrm{e}} e'}{*(e)/m \Longrightarrow_{\mathrm{e}} *(e')} \qquad \frac{n \leq \texttt{length}(m)}{*(\texttt{LOC}\,n)/m \Longrightarrow_{\mathrm{e}} \texttt{lookup}(m, n)}$$

Commands

$$\frac{e/m \Longrightarrow_{\mathrm{e}} e'}{\textbf{CALL}(aid, e, b)/m \Longrightarrow_{\mathrm{c}} \textbf{CALL}(aid, e', b)} \qquad \frac{e/m \Longrightarrow_{\mathrm{e}} e'}{\textbf{RETURN}\; e/m \Longrightarrow_{\mathrm{c}} \textbf{RETURN}\; e'}$$

$$\frac{e/m \Longrightarrow_{\mathrm{e}} e'}{\textbf{EXIT}\; e/m \Longrightarrow_{\mathrm{c}} \textbf{EXIT}\; e'}$$

Statements

$$\frac{}{(\texttt{null};\ s)/m \Longrightarrow_{\mathrm{stm}} s/m} \qquad \frac{s_1/m \Longrightarrow_{\mathrm{stm}} s_1'/m'}{(s_1;\ s_2)/m \Longrightarrow_{\mathrm{stm}} (s_1';\ s_2)/m'}$$

$$\frac{e/m \Longrightarrow_{\mathrm{e}} e'}{\texttt{if}\; e\; \texttt{then}\; s_1\; \texttt{else}\; s_2/m \Longrightarrow_{\mathrm{stm}} \texttt{if}\; e'\; \texttt{then}\; s_1\; \texttt{else}\; s_2/m}$$

$$\frac{}{\texttt{if}\; \mathtt{true}\; \texttt{then}\; s_1\; \texttt{else}\; s_2/m \Longrightarrow_{\mathrm{stm}} s_1/m}$$

$$\frac{}{\texttt{if}\; \mathtt{false}\; \texttt{then}\; s_1\; \texttt{else}\; s_2/m \Longrightarrow_{\mathrm{stm}} s_2/m} \qquad \frac{e/m \Longrightarrow_{\mathrm{e}} e'}{e := e_1/m \Longrightarrow_{\mathrm{stm}} e' := e_1/m}$$

$$\frac{e/m \Longrightarrow_{\mathrm{e}} e'}{\texttt{LOC}(n) := e/m \Longrightarrow_{\mathrm{stm}} \texttt{LOC}(n) := e'/m} \qquad \frac{}{\texttt{LOC}(n) := val/m \Longrightarrow_{\mathrm{stm}} \texttt{null}/m[n] \leftarrow val}$$

$$\frac{e/m \Longrightarrow_{\mathrm{e}} e'}{\texttt{new}\; x := e\; \texttt{in}\; \{s\}/m \Longrightarrow_{\mathrm{stm}} \texttt{new}\; x := e'\; \texttt{in}\; \{s\}/m}$$

$$\frac{}{\texttt{new}\; x := val\; \texttt{in}\; \{s\}/m \Longrightarrow_{\mathrm{stm}} [\texttt{LOC}(\texttt{length}(m) + 1)/x]s/(val{::}m)} \qquad \frac{c/m \Longrightarrow_{\mathrm{c}} c'}{c/m \Longrightarrow_{\mathrm{stm}} c'/m}$$

Figure 4.13: App language operational semantics.

$$
\begin{array}{lcll}
[v/x]y & = & y & x \neq y \\
[v/x]x & = & v & \\
[v/x]e_1 \oplus e_2 & = & [v/x]e_1 \oplus [v/x]e_2 & \\
[v/x]\,{*}(e) & = & {*}([v/x]e) & \\
\hline
[v/x]\textbf{RAISE}\ l & = & \textbf{RAISE}\ l & \\
[v/x]\textbf{DECLASSIFY}\ l & = & \textbf{DECLASSIFY}\ l & \\
[v/x]\textbf{CALL}(aid, e, b) & = & \textbf{CALL}(aid, [v/x]e, b) & \\
[v/x]\textbf{RETURN}\ e & = & \textbf{RETURN}\ [v/x]e & \\
[v/x]\textbf{EXIT}\ e & = & \textbf{EXIT}\ [v/x]e & \\
\hline
[v/x](s_1;\ s_2) & = & ([v/x]s_1;\ [v/x]s_2) & \\
[v/x]\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 & = & \texttt{if}\ [v/x]e\ \texttt{then}\ [v/x]s_1\ \texttt{else}\ [v/x]s_2 & \\
[v/x]e_1 := e_2 & = & [v/x]e_1 := [v/x]e_2 & \\
[v/x]\texttt{new}\ y := e\ \texttt{in}\ \{s\} & = & \texttt{new}\ y := [v/x]e\ \texttt{in}\ \{[v/x]s\} & x \neq y \\
[v/x]\texttt{new}\ x := e\ \texttt{in}\ \{s\} & = & \texttt{new}\ x := [v/x]e\ \texttt{in}\ \{s\} &
\end{array}
$$

Figure 4.14: Definition of variable substitution.

$$
\begin{array}{lcl}
IID & ::= & N \\
Apps & ::= & \texttt{list}\ stm \\
ret & ::= & \texttt{None} \mid \texttt{Some}\ (AppID, IID) \\
mem & ::= & \texttt{list}\ val \\
Instance & ::= & (IID, mem, stm, ret) \\
IS & ::= & \texttt{list}(\texttt{list}\ Instance) \\
LM & ::= & \texttt{list}\ AppLabelMap \\
PendingType & ::= & \texttt{pCall}\ ret \mid \texttt{pRet}\ IID \\
PendingRq & ::= & (AppID, val, label, PendingType) \\
PQ & ::= & \texttt{list}\ PendingRq \\
Sys & ::= & (Apps, IS, LM, PQ)
\end{array}
$$

Figure 4.15: The syntax for the Android system model.

$[]$	The empty list
$[a]$	A list with a single element a
$(a{::}l)$	The list construction operator :: appends element a to list l
$(l_1 \mathbin{++} l_2)$	This operator appends the list $l1$ to the list $l2$
$\texttt{len}(l)$	This function returns the number of elements in the list
$l\lceil n \rfloor_d$	This function returns the n'th element of the list l; returning d if $\texttt{len}(l) \leq n$
$\texttt{map}(f, l)$	Apply the function f to each element in l and return the resulting list
$a \in l$	Test whether the a is an element of l or not

Figure 4.16: Summary of list notation and operations.

$$\begin{array}{lcl} \mathit{DynamicLabelList} & ::= & \texttt{list}(\mathit{IID}, \mathit{label}) \\ \mathit{AppLabelMap} & ::= & \text{SingleInstanceUnlaunched } \mathit{label}\ \mathit{IID} \mid \\ & & \text{SingleInstanceLaunched } \mathit{label}\ \mathit{label}\ \mathit{IID} \mid \\ & & \text{MultiInstance } \mathit{label}\ \mathit{DynamicLabelList} \mid \text{NA} \end{array}$$

Figure 4.17: The syntax for the app label map.

Return address When a call is made the caller has the option to expect a return and the system will register the *return address* as part of the callee state. This address will be used whenever the callee issues the **RETURN** command. The address is a 2-tuple: the app ID and the instance ID. The app ID is needed because we only guarantee that instance IDs are unique within a single app. This assumption makes it easier to assign an ID to a new instance when we design the operational semantics later. If the caller does not expect a return the return address will be `None`.

Label map The label map keeps track of each app's status and security labels. Figure 4.17 encode the syntax of an app's label map entry. For single-instance apps, the entry indicates whether an app is launched or not. An unlaunched, single-instance app has a static security label and the instance ID, IID, of the previous instance. A launched app has, in addition to the static label, the dynamic, or effective, label and the ID of the currently running instance. For multi-instance apps, we record the static label and a list of instance IDs and dynamic label pairs, which inlcudes a pair for each active instance of the app. We call this list the app's *dynamic-label-list*. The NA mapping is reserved for apps that should not be available, and is only used when we use *projection* on a system for the purpose of the noninterference proofs (refer to Sec. 4.3).

Pending Queue The pending queue (PQ) contains all the call or return requests that could not be processed before. Each request consists of the ID of the destination app, the dynamic label of the source app instance at the time of the original request, and either the destination instance ID for return requests, or the return address for call requests.

Instance set The set of all active instances, IS, in the system is implemented as a list of lists. Each entry in the top-level list, indexed by the app ID, contains the list of active instances of the corresponding app; we call such list the *app-instance-set*. Unlike the instances set, app-instance-sets are not indexed by the instance IDs and can change length throughout the execution of the system. Each app instance consists of an instance ID, the instance memory, the program statement, and the return address.

4.2.4 Modelling the Android System: Semantics

In this section we detail the operational semantics of our Android model. For reference, all the judgments used in this section are listed in Tables 4.1 and 4.2 with a brief description of each judgment.

We write $S \Longrightarrow^{a} S'$ to state that a system instance, sys, takes a step with label a and that S' is what becomes of S after the step is taken. In other words, instance S transitions to instance S', and a is the transition label. A system step can produce an *event*, which is recorded using the transition label. If the system step produces no event, the transition label will have a the special value of τ, and we call it a τ step ($\Longrightarrow^{\tau}$). For now, the only events are output events (we add declassification events later). We write $\Longrightarrow^{(n,dl)}$ to indicate that the output event had a value of n and security label dl.

An instance of the system can take a step in three main ways: (1) $\Longrightarrow_{\text{is}}$ an internal step where an app instance program takes a regular (i.e., not a command) step, (2) $\Longrightarrow_{\text{pq}}$ a pending request being processed from the pending queue, or (3) $\rhd^{a}$ an execution step where a command that has been issued by an app instance is executed. Output events record the values sent between app instances and the security labels of the sending instances. Thus only command execution steps for the **CALL** and **RETURN** commands can result in output steps, all other system steps are τ steps. Since the sent value and the security label are fixed at the time the command is issued we can always record the output then, even if the request will be placed in the pending

$\boxed{S_1 \overset{\texttt{OUT}(n,l)}{\Longrightarrow} S_2}$	System instance S_1 takes a step that produces an output event of value n and label l, and results in instance S_2
$\boxed{S_1 \overset{\tau}{\Longrightarrow} S_2}$	System instance S_1 takes a silent, or τ, step resulting in instance S_2
$\boxed{is_1 \Longrightarrow_{\text{is}} is_2}$	Internal step: Exactly one app instance in instance set is_1 takes an internal (i.e., not a command) step with the resulting instance set being is_2
$\boxed{(lm_1, is_1, pq_1)/apps \Longrightarrow_{\text{pq}} (lm_2, is_2, pq_2)}$	PQ processing: starting with is_1, lm_1, and pq_1 (with an app set *App*), one request from the pq_1 is processed resulting in is_2, lm_2, and pq_2
$\boxed{(lm_1, is_1, pq_1)/apps \overset{\texttt{OUT}(n,l)}{\rhd} (lm_2, is_2, pq_2)}$	Execution step: Starting with is_1, lm_1, and pq_1 (with an app set *App*) an app instance in is_1 issues a command and and the command is executed resulting in a final state is_2, lm_2, and pq_2, and producing an output of value n and label l
$\boxed{(lm_1, is_1, pq_1)/apps \overset{\tau}{\rhd} (lm_2, is_2, pq_2)}$	Execution Step: similar to the judgment above, except that there is no output
$\boxed{ais_1 \Longrightarrow_{\text{as}} ais_2}$	Exactly one app instance in the app-instance-set ais_1 takes an internal step with the end result being ais_2
$\boxed{(aid, iid)@is_1 \succ_{is} \langle c, is_2, ra\rangle}$	Command issuing: Exactly one app instance, identified by the pair (aid, iid), issues command c and the instance set is_1 will step to is_2 in the process; ra exposes the return address saved in the issuing instance
$\boxed{iid@ais_1 \succ_{as} \langle c, ais_2, ra\rangle}$	Command issuing: Exactly one app instance with ID iid issues command c and the app-instance-set ais_1 will step to ais_2 in the process; ra exposes the return address saved in the issuing instance
$\boxed{s \mapsto (c, s')}$	Indicates that immediate next step in running statement s is the execution of command c followed by s'
$\boxed{\text{IsReady}(c)}$	All the arguments of command c are fully evaluated and its ready to be executed
$\boxed{(lm_1, is_1, pq_1)/apps \underset{\textsc{Call}}{\rhd} \langle dl, aid_{callee}, n, ra\rangle\ (lm_2, is_2, pq_2)}$	Executing a call from an instance whose dynamic label is *dl* to app aid_{callee} with argument n and return address ra, starting from state (lm_1, is_1, pq_1) ends with state (lm_2,is_2,pq_2)
$\boxed{lm_1 \underset{\textsc{Raise}}{\rhd} \langle aid, iid, l\rangle\ lm_2}$	Raising the label of app instance (aid, iid) in the label map lm_1 to the label l results in label map lm_2

Table 4.1: A description of all judgments used to define the system operational semantics (part 1 of 2).

$\boxed{(lm, is_1, pq_1)/apps \underset{\textsc{Return}}{\rhd} \langle dl, aid_{callee}, iid_{callee}, n \rangle\ (is_2, pq_2)}$ Starting from state (lm, is_1, pq_1) and executing a return call from an app instance whose dynamic label is dl to the instance $(aid_{callee}, iid_{callee})$ with argument n results in state (lm, is_2, pq_2)

$\boxed{(lm_1, is_1, pq_1)/apps \underset{\textsc{Exit}}{\rhd} \langle aid, iid \rangle\ (lm_2, is_2, pq_2)}$ Starting from state (lm, is_1, pq_1) and ending app instance (aid, iid) by removing it and reference to it in the system, results in state (lm, is_2, pq_2)

$\boxed{(aid, iid)@lm\ \texttt{HasLabel}\ (sl, dl)}$ The label map lm assigns app instance (aid, iid) static label sl dynamic label dl

---

$\boxed{aid@lm\ \texttt{HasAppLabel}\ (alm)}$ The label map lm assigns app aid the app-label-map alm

---

$\boxed{\texttt{Update}\ aid@lm_1\ \texttt{with}\ (alm)\ \mapsto\ lm_2}$ Replacing the app-label-map of app aid in label map lm_1 with alm yields label map lm_2

$\boxed{iid@dll\ \texttt{HasInstanceLabel}\ dl}$ The instance with ID iid is assigned dynamic label dl in the dynamic-label-list dll

---

$\boxed{\texttt{Update}\ iid@dll_1\ \texttt{with}\ l\ \mapsto\ dll_2}$ Updating the dynamic-label-list dll_1 to reassign instance ID iid label to be l result in dll_2

$\boxed{\texttt{BusyApp}\ iid@ais}$ The app instance with ID iid in the app-instance-set ais is still running, i.e., has a program statement the is not **`null`**

$\boxed{\texttt{Busy}\ (aid, iid)@is}$ The app instance (aid, iid) in the instances set is is still running, i.e., has a program statement the is not **`null`**

$\boxed{\texttt{Send}\ \langle n, ra, s \rangle\ \texttt{ToExisting}\ iid@(ais_1)\ \mapsto\ ais_2}$ Replacing the instance with ID iid program with statement s and its return address with ra, in ais_1 yields ais_2

$\boxed{\texttt{Send}\ \langle n, ra, s \rangle\ \texttt{ToNew}\ iid@(ais_1)\ \mapsto\ ais_2}$ Adding a new instance with ID iid, empty memory, program s, and return address ra to ais_1 yields ais_2

$\boxed{\texttt{RemoveRA}\ (aid, iid)\ \texttt{from}\ is_1\ \mapsto\ is_2}$ Instances set is_2 is the result of removing any return address pointing to (aid, iid) from all instances in is_1

$\boxed{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ais_1\ \mapsto\ ais_2}$ App-instance-set ais_2 is the result of removing any return address pointing to (aid, iid) from all instances in ais_1

$\boxed{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2}$ Pending queue pq_2 is the result of removing any request or return address pointing to (aid, iid) from pq_1

Table 4.2: A description of all judgments used to define the system operational semantics (part 2 of 2).

$$\textsc{Sys-Internal}\quad \frac{is_1 \Longrightarrow_{\text{is}} is_2}{(apps, lm, is_1, pq) \overset{\tau}{\Longrightarrow} (apps, lm, is_2, pq)}$$

$$\textsc{Sys-PQ}\quad \frac{(lm_1, is_1, pq_1)/apps \Longrightarrow_{\text{pq}} (lm_2, is_2, pq_2)}{(apps, lm_1, is_1, pq_1) \overset{\tau}{\Longrightarrow} (apps, lm_2, is_2, pq_2)}$$

$$\textsc{Sys-Exe-Out}\quad \frac{(lm_1, is_1, pq_1)/apps \overset{\texttt{OUT}(n,l)}{\rhd} (lm_2, is_2, pq_2)}{(apps, lm_1, is_1, pq_1) \overset{\texttt{OUT}(n,l)}{\Longrightarrow} (apps, lm_2, is_2, pq_2)}$$

$$\textsc{Sys-Exe-Tau}\quad \frac{(lm_1, is_1, pq_1)/apps \overset{\tau}{\rhd} (lm_2, is_2, pq_2)}{(apps, lm_1, is_1, pq_1) \overset{\tau}{\Longrightarrow} (apps, lm_2, is_2, pq_2)}$$

Figure 4.18: The top-level system steps operational semantics.

queue. Therefore, processing pending requests is always a τ step. Figure 4.18 formalizes these semantics. Next, we will cover in more details the semantics for internal steps, command execution, and pending queue processing.

4.2.4.1 Internal steps $is_1 \Longrightarrow_{\text{is}} is_2$

Figure 4.19 encodes the semantics for internal steps. Rules I-IS-STEP and I-IS-CON state that exactly one app takes an internal step. An app takes an internal step if exactly one of its instances takes a step; this fact is encoded by rules I-AIS-STEP and I-AIS-CON. Finally, as I-AIS-STEP states, an app instance takes a step if its program (i.e., statement) takes a step.

4.2.4.2 Command execution $(lm_1, is_1, pq_1)/apps \overset{a}{\rhd} (lm_2, is_2, pq_2)$

A command is executed by making specific changes in the instances set, label map, and pending queue. In addition, a command can optionally produce an output. Thus, we write
$(lm_1, is_1, pq_1)/apps \overset{a}{\rhd} (lm_2, is_2, pq_2)$ to indicate that an app instance in is_1 executed some

I-AIS-CON
$$\frac{ais_1 \Longrightarrow_{\text{as}} ais_2}{(inst{::}ais_1) \Longrightarrow_{\text{as}} (inst{::}ais_2)}$$

I-AIS-STEP
$$\frac{s_1/m_1 \Longrightarrow_{\text{stm}} s_2/m_2}{((iid, m_1, s_1, ra){::}ais) \Longrightarrow_{\text{as}} ((iid, m_2, s_2, ra){::}ais)}$$

I-IS-CON
$$\frac{is_1 \Longrightarrow_{\text{is}} is_2}{(ais{::}is_1) \Longrightarrow_{\text{is}} (ais{::}is_2)}$$

I-IS-STEP
$$\frac{ais_1 \Longrightarrow_{\text{as}} ais_2}{(ais_1{::}is) \Longrightarrow_{\text{is}} (ais_2{::}is)}$$

Figure 4.19: The operational semantics for internal steps.

command resulting in changing lm_1 to lm_2, is_1 to is_2, and pq_1 to pq_2, and that there was an event a. Note that a could be τ, meaning that there was no output. In addition, it could be the case that there were no changes to the label map, $lm_1 = lm_2$, or to the pending queue, $pq_1 = pq_2$.

The operational semantics for command execution are in Fig. 4.20. The first premise of all these rules is a command issuing step ($\succ_{is}$) where a command is dispatched from the issuing app instance. Most rules also utilize the $(iid, lm)@aid$ `HasLabel` (sl, dl) judgment, which retrieves the static and dynamic labels of the specified app instance. The actual execution of a command is done using a corresponding execution judgment, with the general format of $((\textit{initial state})\ \triangleright_{\textit{command name}}\langle\textit{command arguments}\rangle\ (\textit{resulting state}))$; for example the judgment for the execution of the **RAISE** command is $lm_1\ \triangleright_{\textbf{RAISE}}\langle aid, iid, l\rangle\ lm_2$. We define command issuing and the specific command execution rules below. One detail worth pointing to is the second premise of the last two rules concerning the **EXIT** command. These conditions are needed to make sure that an instance that is exiting, and hence will be removed from the system, will not return to itself. Because the basic noninterference property requires that no declassification steps are taken by the system, we exclude the operational semantics for the **DECLASSIFY** command from this model (We add them in Sec. 4.4).

Issuing commands $\succ_{is}$ The first step of executing a command is to extract the command from the issuing app instance. Exactly one of the app instances must be in a state where the next statement to be executed is a command, and there is no further evaluation needed for

$$\mathrm{getRA}(er, aid, iid) = \begin{cases} \texttt{None} & er = \texttt{false} \\ \texttt{Some}\ ((aid, iid)) & er = \texttt{true} \end{cases}$$

$$\frac{\begin{array}{c}(aid, iid)@is_1 \succ_{is} \langle \textbf{CALL}(aid_{callee}, n, er), is', ra\rangle \qquad (aid, iid)@lm_1\ \texttt{HasLabel}\ (sl, dl) \\ (lm_1, is', pq_1)/apps \underset{\textbf{CALL}}{\rhd} \langle dl, aid_{callee}, n, \mathrm{getRA}(er, aid, iid)\rangle\ (lm_2, is_2, pq_2)\end{array}}{(lm_1, is_1, pq_1)/apps \overset{\texttt{OUT}(n,dl)}{\rhd} (lm_2, is_2, pq_2)}\ \textsc{Exe-C}$$

$$\frac{(aid, iid)@is_1 \succ_{is} \langle \textbf{RAISE}\ l, is_2, ra\rangle \qquad lm_1 \underset{\textbf{RAISE}}{\rhd} \langle aid, iid, l\rangle\ lm_2}{(lm_1, is_1, pq)/apps \overset{\tau}{\rhd} (lm_2, is_2, pq)}\ \textsc{Exe-Ra}$$

$$\frac{\begin{array}{c}(aid, iid)@is_1 \succ_{is} \langle \textbf{RETURN}\ n, is', \texttt{Some}\ ((aid_{callee}, iid_{callee}))\rangle \\ (aid, iid)@lm\ \texttt{HasLabel}\ (sl, dl) \\ (lm, is', pq_1)/apps \underset{\textbf{RETURN}}{\rhd} \langle dl, aid_{callee}, iid_{callee}, n\rangle\ (is_2, pq_2)\end{array}}{(lm, is_1, pq_1)/apps \overset{\texttt{OUT}(n,dl)}{\rhd} (lm, is_2, pq_2)}\ \textsc{Exe-R1}$$

$$\frac{(aid, iid)@is_1 \succ_{is} \langle \textbf{RETURN}\ n, is_2, \texttt{None}\rangle \qquad (aid, iid)@lm\ \texttt{HasLabel}\ (sl, dl)}{(lm, is_1, pq)/apps \overset{\texttt{OUT}(n,dl)}{\rhd} (lm, is_2, pq)}\ \textsc{Exe-R2}$$

$$\frac{\begin{array}{c}(aid, iid)@is_1 \succ_{is} \langle \textbf{EXIT}\ n, is', \texttt{Some}\ ((aid_{callee}, iid_{callee}))\rangle \\ aid \neq aid_{callee} \vee iid \neq iid_{callee} \\ (aid, iid)@lm_1\ \texttt{HasLabel}\ (sl, dl) \qquad (lm_1, is', pq_1)/apps \underset{\textbf{EXIT}}{\rhd} \langle aid, iid\rangle\ (lm_2, is'', pq') \\ (lm_2, is'', pq')/apps \underset{\textbf{RETURN}}{\rhd} \langle dl, aid_{callee}, iid_{callee}, n\rangle\ (lm_2, is_2) pq_2\end{array}}{(lm_1, is_1, pq_1)/apps \overset{\texttt{OUT}(n,dl)}{\rhd} (lm_2, is_2, pq_2)}\ \textsc{Exe-E1}$$

$$\frac{\begin{array}{c}(aid, iid)@is_1 \succ_{is} \langle \textbf{EXIT}\ n, is', ra\rangle \qquad ra = \texttt{Some}\ ((aid, iid)) \vee ra = \texttt{None} \\ (aid, iid)@lm_1\ \texttt{HasLabel}\ (sl, dl) \qquad (lm_1, is', pq_1)/apps \underset{\textbf{EXIT}}{\rhd} \langle aid, iid\rangle\ (lm_2, is_2, pq_2)\end{array}}{(lm_1, is_1, pq_1)/apps \overset{\texttt{OUT}(n,dl)}{\rhd} (lm_2, is_2, pq_2)}\ \textsc{Exe-E2}$$

Figure 4.20: The operational semantics for command execution.

the command arguments. We write $(aid, iid)@is_1 \succ_{is} \langle c, is_2, ra\rangle$ to say that exactly one app instance, identified by the pair (aid, iid) issues command c and the instances set is_1 will step to is_2 in the process. ra, which exposes the return address saved in the issuing instance is used for executing the **RETURN** and **EXIT** commands.

Figure 4.21 formalizes these requirements. The last two rules, ISS-IS-CON and ISS-IS-STEP, define the judgment $(aid, iid)@is_1 \succ_{is} \langle c, is_2, ra\rangle$ ensuring that exactly one app-instance-set issues a command. The first five rules in Fig .4.21 define the judgment $iid@ais_1 \succ_{as} \langle c, ais_2, ra\rangle$, which state that the app instance in ais_1, with instance ID iid and return address ra, takes a step dispatching the command c. The first four rules define the actual dispatching of the ($\succ_{as}$) commands, i.e., the app instance (at the head of the app-instance-set) takes a step by removing the command statement leaving only the residual program. The residual program is just the rest of the app instance code that follows the issued command and is determined by the $s \mapsto (c, r)$ judgment, which states that a program s with an active command c will step to program r after dispatching the command. For example, $(\textbf{RAISE}\, l;\ (s_2;\ s_3)) \mapsto (\textbf{RAISE}\, l, (s_2;\ s_3))$ and $\textbf{RETURN}\, 0 \mapsto (\textbf{RETURN}\, 0, \texttt{null})$. The fifth rule, ISS-AIS-CON, completes the definition of this judgment by allowing instances that are not at the head of list to issue commands.

The RAISE l command This command allows app instances to change their (dynamic) security label to a more strict label, i.e., a label that is higher in the security lattice. This operation is always safe, thus we should only need to check if the label is actually higher. That is, assuming that the current dynamic label is dl, then we need to check that $dl \sqsubseteq l$. However, as we found out when trying to prove the noninterference theorem for the system, this will lead to potential leaks in the case of single-instance apps (refer to Sec. 2.3.2). Therefore, for single-instance apps, we will make sure that we also raise the static label if necessary to prevent it from ever being lower than the dynamic label. The operational semantics for the **RAISE** command execution are defined in Fig. 4.22. We do not define this judgment for nonexistent

Define active commands and find residual program

$$\text{IsReady}(c) = \begin{cases} \text{True} & c \text{ is } \textbf{RAISE} \text{ or } \textbf{DECLASSIFY} \\ \text{True} & c \text{ is } \textbf{CALL}, \textbf{RETURN}, \text{ or } \textbf{EXIT} \text{ with fully evaluated arguments} \\ \text{False} & c \text{ is } \textbf{CALL}, \textbf{RETURN}, \text{ or } \textbf{EXIT} \text{ with none fully evaluated arguments} \end{cases}$$

$$\frac{s \text{ is a command } c \qquad \text{IsReady}(c)}{s \mapsto (c, \texttt{null})} \qquad \frac{s \mapsto (c, r)}{(s;\ s_2) \mapsto (c, (r;\ s_2))}$$

Command issuing

Iss-AIS-RaD
$$\frac{s \mapsto (c, r) \qquad c \text{ is a } \textbf{RAISE} \text{ or } \textbf{DECLASSIFY} \text{ command}}{iid@((iid, m, s, ra)::ais) \succ_{as} \langle c, ((iid, m, r, ra)::ais), \texttt{None}\rangle}$$

Iss-AIS-C
$$\frac{s \mapsto (\textbf{CALL}(aid, \text{exp}, er), r)}{iid@((iid, m, s, ra)::ais) \succ_{as} \langle \textbf{CALL}(aid, \text{exp}, er), ((iid, m, r, ra)::ais), ra\rangle}$$

Iss-AIS-R
$$\frac{s \mapsto (\textbf{RETURN}\ \text{exp}, r)}{iid@((iid, m, s, ra)::ais) \succ_{as} \langle \textbf{RETURN}\ \text{exp}, ((iid, m, \texttt{null}, ra)::ais), \texttt{None}\rangle}$$

Iss-AIS-E
$$\frac{s \mapsto (\textbf{EXIT}\ \text{exp}, r)}{iid@((iid, m, s, ra)::ais) \succ_{as} \langle \textbf{EXIT}\ \text{exp}, ((iid, [], \texttt{null}, ra)::ais), \texttt{None}\rangle}$$

Iss-AIS-Con
$$\frac{iid@ais_1 \succ_{as} \langle c, ais_2, ra\rangle}{iid@(inst::ais_1) \succ_{as} \langle c, (inst::ais_2), ra\rangle}$$

Iss-IS-Con
$$\frac{(aid, iid)@is_1 \succ_{is} \langle c, is_2, ra\rangle}{(aid+1, iid)@(ais::is_1) \succ_{is} \langle c, (ais::is_2), ra\rangle}$$

Iss-IS-Step
$$\frac{iid@ais_1 \succ_{as} \langle c, ais_2, ra\rangle}{(0, iid)@(ais_1::is) \succ_{is} \langle c, (ais_2::is), ra\rangle}$$

Figure 4.21: The operational semantics for issuing commands.

RA-S1

$$\frac{\begin{array}{c} aid@lm_1 \texttt{ HasAppLabel } (\text{SingleInstanceLaunched } iid\ sl\ dl) \\ dl \sqsubseteq l \qquad l \not\sqsubseteq sl \qquad \texttt{Update } aid@lm_1 \texttt{ with } (\text{SingleInstanceLaunched } iid\ l\ l) \ \mapsto\ lm_2 \end{array}}{lm_1 \underset{\textbf{RAISE}}{\rhd} \langle aid, iid, l\rangle\ lm_2}$$

RA-S2

$$\frac{\begin{array}{c} aid@lm_1 \texttt{ HasAppLabel } (\text{SingleInstanceLaunched } iid\ sl\ dl) \\ dl \sqsubseteq l \qquad l \sqsubseteq sl \qquad \texttt{Update } aid@lm_1 \texttt{ with } (\text{SingleInstanceLaunched } iid\ sl\ l) \ \mapsto\ lm_2 \end{array}}{lm_1 \underset{\textbf{RAISE}}{\rhd} \langle aid, iid, l\rangle\ lm_2}$$

RA-S-B

$$\frac{aid@lm \texttt{ HasAppLabel } (\text{SingleInstanceLaunched } iid\ sl\ dl) \qquad dl \not\sqsubseteq l}{lm \underset{\textbf{RAISE}}{\rhd} \langle aid, iid, l\rangle\ lm}$$

RA-M

$$\frac{\begin{array}{c} aid@lm_1 \texttt{ HasAppLabel } (\text{MultiInstance } sl\ dll_1) \\ iid@dll_1 \texttt{ HasInstanceLabel } dl \qquad dl \sqsubseteq l \\ \texttt{Update } iid@dll_1 \texttt{ with } l \ \mapsto\ dll_2 \qquad \texttt{Update } aid@lm_1 \texttt{ with } (\text{MultiInstance } sl\ dll_2) \ \mapsto\ lm_2 \end{array}}{lm_1 \underset{\textbf{RAISE}}{\rhd} \langle aid, iid, l\rangle\ lm_2}$$

RA-M-B

$$\frac{aid@lm \texttt{ HasAppLabel } (\text{MultiInstance } sl\ dll) \qquad iid@dll \texttt{ HasInstanceLabel } dl \qquad dl \not\sqsubseteq l}{lm \underset{\textbf{RAISE}}{\rhd} \langle aid, iid, l\rangle\ lm}$$

Figure 4.22: The operational semantics for the **RAISE** command.

instances since only existing instances can issue commands. The rules RA-S* are for single-instance apps and the rules RA-M* are for multi-instance-apps. Rules RA-S-B and RA-S-B cover the cases when the raise request is denied because it is not to a label that is higher in the lattice. Rule RA-S1 is for the case where the new label is not the same as or below the static label of a single-instance app and we need to raise both the dynamic and static labels. Rule RA-S2 cover the case where we only need to raise the dynamic label. The simpler case for multi-instance apps is covered by the rule RA-M.

$$\text{app-stm}(apps, aid) = apps\lceil aid \rfloor_{\texttt{null}}$$

Busy app instance

Bus-AIS

$$\frac{s \neq \texttt{null}}{\texttt{BusyApp}\ iid@((iid, m, s, ra)::ais)}$$

Bus-AIS-Con

$$\frac{\texttt{BusyApp}\ iid@ais}{\texttt{BusyApp}\ iid@(inst::ais)}$$

Bus-IS

$$\frac{\texttt{BusyApp}\ iid@(is\lceil aid \rfloor_{[]})}{\texttt{Busy}\ (aid, iid)@is}$$

Sending an intent

Snd-Ext-Con

$$\frac{\texttt{Send}\ \langle n, ra, s\rangle\ \texttt{ToExisting}\ iid@(ais_1) \ \mapsto\ ais_2}{\texttt{Send}\ \langle n, ra, s\rangle\ \texttt{ToExisting}\ iid@((inst::ais_1)) \ \mapsto\ (inst::ais_2)}$$

Snd-Ext

$$\frac{}{\begin{array}{l}\texttt{Send}\ \langle n, ra, s\rangle\ \texttt{ToExisting}\ iid@(((iid, m, \texttt{null}, ra_{old})::ais)) \\ \qquad\qquad \mapsto\ ((iid, m, [n/\texttt{ARG}]s, ra)::ais)\end{array}}$$

Snd-New

$$\frac{}{\texttt{Send}\ \langle n, ra, s\rangle\ \texttt{ToNew}\ iid@(ais) \ \mapsto\ ((iid, [], [n/\texttt{ARG}]s, ra)::ais)}$$

Figure 4.23: Helper definitions for the **Call** and **Return** command operational semantics.

Sending intents and Busy Instances To complete an allowed call or return request, the argument value needs to be substituted into a fresh copy of the app's code that will be put in a new or existing instance. The exact details are encoded in the judgments in Fig. 4.23. The judgment Send $\langle n, ra, s\rangle$ ToExisting $iid@(ais_1) \;\mapsto\; ais_2$ states that the result of sending an intent, identified by iid with argument n and return address ra, to the app-instance-set ais_1 is ais_2. The second judgment is similar except that the intent is sent to a newly created instance that is added to the app-instance-set. It's assumed that s is a fresh copy of the app's code. The argument value, n, is substituted in place of the ARG variable in the app's code. If we are sending an intent to an existing app instance (SEND-EXT and SEND-EXT-CON) we need to match the instance ID in the request with that of the instance. In this case, the instance's code will be replaced by the fresh copy, but its memory will be kept intact. On the other hand, if we are sending the intent to a new instance (SEND-NEW), we need to add a new entry at the head of the app-instance-set with the requested instance ID, an empty memory, and the fresh copy of the code. An important detail to notice here is that, when sending an intent to an existing instance, we require that this instance be ready to accept new intents, i.e., not busy processing a previous request. This is achieved by requiring that the instance's current program be **null**. The Busy $(aid, iid)@is$ judgment state that the instance identified by the app ID aid and instance ID iid in the instance set is is busy processing a request, i.e., its program is not **null**. This judgment along with those for sending intents will be used for the operational semantics of call and return commands below.

The CALL(aid, n, er) command A call request is executed by blocking it, putting it in the pending queue, or by allowing it and sending the intent to the callee app. The first rule in Fig. 4.24 (C-NA) deals with unavailable apps. Since we don't put restrictions on what app IDs are allowed in the **CALL** command, they could refer to nonexistent apps. Also, while the NA app label should not occur in normal operations, it needs to be handled to cover *projected systems* where this label could appear (see Sec. 4.3). The next three rules handle calls to already launched single-instance apps. If the current instance's dynamic label allows the call to be made without violating the security policy (C-S), the execution is handled normally by sending the intent to the instance and thus updating the instances set. If the call is allowable but the callee's current instance is busy, we put the request in the pending queue (C-S-P). We save the caller's dynamic label because we need to compare it again with the callee's when we process the pending request later, as it could have changed. We ignore, i.e., block, the request if the callee's dynamic label does not allow it (C-S-B).[3] The last four rules deal with calls to unlaunched single-instance apps or multi-instance apps. In both cases, we only need to compare the caller's dynamic label with the callee's static label and either allow the request (C-SU and C-M) or block it (C-SU-B and C-M-B). For single instance apps, we increment the previous instance ID by one to get the ID for the new instance. But for multi-instance apps, we need to find the maximum ID in the apps list of active instances and increment it, which is the role of the $\text{next-IID}(dll)$ function.

The RETURN n command A return is executed by sending an intent to the specified app instance (see Fig. 4.25). We need not be concerned with returning to a nonexistent app instance because once an app instance exits we will remove all pending return requests and all saved return addresses pointing to the exited instance in the system. Any request that violates the security policy is blocked (R-S-B and R-M-B). Otherwise, the request is put in the pending queue if the instance is busy (R-S-P and R-M-P), or processed immediately if it's not (R-S and R-M). Note that the label map never changes when processing return requests.

Removing expired return address and pending requests When an instance exits we need to clean up the system by removing any references to it. There are two places where we find such references: the requests in the pending queue and the saved return addresses in the instances set. We write `RemoveRA` (aid, iid) `from` $is_1 \mapsto is_2$ to state that is_1 becomes is_2 after removing all return address to the app instance identified by aid and iid. Similarly, we

[3]Basing the decision on the static label will prevent us from proving noninterference.

C-NA

$$\frac{aid \geq \texttt{len}(apps) \vee aid@lm\ \texttt{HasAppLabel}\ (\text{NA})}{(lm, is, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm, is, pq)}$$

C-S

$$\frac{\begin{array}{c} aid@lm_1\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl) \\ dl_{caller} \sqsubseteq dl \qquad \texttt{Send}\ \langle n, ra, \text{app-stm}(apps, aid)\rangle\ \texttt{ToExisting}\ iid@(is_1\lceil aid \rfloor_{[]})\ \mapsto\ ais_{new} \\ \texttt{Update}\ aid@(lm_1, is_1)\ \texttt{with}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl, ais_{new})\ \mapsto\ (lm_2, is_2) \end{array}}{(lm_1, is_1, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm_2, is_2, pq)}$$

C-S-P

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl) \\ dl_{caller} \sqsubseteq dl \qquad \texttt{Busy}\ (aid, iid)@is \end{array}}{(lm, is, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm, is, ((aid, n, dl_{caller}, \texttt{pCall}\ ra)::pq))}$$

C-S-B

$$\frac{aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl) \qquad dl_{caller} \not\sqsubseteq dl}{(lm, is, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm, is, pq)}$$

C-SU

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceUnlaunched}\ sl\ iid_{prev}) \qquad dl_{caller} \sqsubseteq sl \\ \texttt{Send}\ \langle n, ra, \text{app-stm}(apps, aid)\rangle\ \texttt{ToNew}\ (iid_{prev}+1)@(is_1\lceil aid \rfloor_{[]})\ \mapsto\ ais_{new} \\ \texttt{Update}\ aid@(lm_1, is_1)\ \texttt{with}\ (\text{SIL}\ (iid_{prev}+1)\ sl\ sl, ais_{new})\ \mapsto\ (lm_2, is_2) \end{array}}{(lm_1, is_1, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm_2, is_2, pq)}$$

C-SU-B

$$\frac{aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceUnlaunched}\ sl\ iid_{prev}) \qquad dl_{caller} \not\sqsubseteq sl}{(lm, is, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm, is, pq)}$$

C-M

$$\frac{\begin{array}{c} aid@lm_1\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll) \qquad dl \sqsubseteq sl \qquad iid = \text{next-IID}(dll) \\ \texttt{Send}\ \langle n, ra, \text{app-stm}(apps, aid)\rangle\ \texttt{ToNew}\ iid@(is_1\lceil aid \rfloor_{[]})\ \mapsto\ ais_{new} \\ \texttt{Update}\ aid@(lm_1, is_1)\ \texttt{with}\ (\text{MultiInstance}\ sl\ ((iid, sl)::dll), ais_{new})\ \mapsto\ (lm_2, is_2) \end{array}}{(lm_1, is_1, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm_2, is_2, pq)}$$

C-M-B

$$\frac{aid@lm_1\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll) \qquad dl \not\sqsubseteq sl}{(lm, is, pq)/apps\ \underset{\textsc{Call}}{\rhd}\ \langle dl_{caller}, aid, n, ra\rangle\ (lm, is, pq)}$$

Figure 4.24: The operational semantics for the CALL command.

R-S

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}) \qquad \mathit{dl}_{\mathit{caller}} \sqsubseteq \mathit{dl} \\ \texttt{Send}\ \langle n, \texttt{None}, \text{app-stm}(\mathit{apps}, \mathit{aid})\rangle\ \texttt{ToExisting}\ \mathit{iid}@(is_1\lceil \mathit{aid}\rfloor_{[]})\ \mapsto\ \mathit{ais}_{new} \\ \texttt{Update}\ \mathit{aid}@(lm, is_1)\ \texttt{with}\ (\text{SingleInstanceLaunched}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}, \mathit{ais}_{new})\ \mapsto\ (lm, is_2) \end{array}}{(lm, is_1, pq)/\mathit{apps}\ \underset{\textbf{Return}}{\rhd}\ \langle \mathit{dl}_{\mathit{caller}}, \mathit{aid}, \mathit{iid}, n\rangle\ (is_2, pq)}$$

R-S-P

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}) \\ \mathit{dl}_{\mathit{caller}} \sqsubseteq \mathit{dl} \qquad \texttt{Busy}\ (\mathit{aid}, \mathit{iid})@is \end{array}}{(lm, is, pq)/\mathit{apps}\ \underset{\textbf{Return}}{\rhd}\ \langle \mathit{dl}_{\mathit{caller}}, \mathit{aid}, \mathit{iid}, n\rangle\ (is, ((\mathit{aid}, n, \mathit{dl}_{\mathit{caller}}, \texttt{pRet}\ \mathit{iid})::pq))}$$

R-S-B

$$\frac{aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}) \qquad \mathit{dl}_{\mathit{caller}} \not\sqsubseteq \mathit{dl}}{(lm, is, pq)/\mathit{apps}\ \underset{\textbf{Return}}{\rhd}\ \langle \mathit{dl}_{\mathit{caller}}, \mathit{aid}, \mathit{iid}, n\rangle\ (is, pq)}$$

R-M

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ \mathit{sl}\ \mathit{dll}) \\ \mathit{iid}@\mathit{dll}\ \texttt{HasInstanceLabel}\ \mathit{dl} \qquad \mathit{dl}_{\mathit{caller}} \sqsubseteq \mathit{dl} \\ \texttt{Send}\ \langle n, \texttt{None}, \text{app-stm}(\mathit{apps}, \mathit{aid})\rangle\ \texttt{ToExisting}\ \mathit{iid}@(is_1\lceil \mathit{aid}\rfloor_{[]})\ \mapsto\ \mathit{ais}_{new} \\ \texttt{Update}\ \mathit{aid}@(lm_1, is_1)\ \texttt{with}\ (\text{MultiInstance}\ \mathit{sl}\ ((\mathit{iid}, \mathit{sl})::\mathit{dll}), \mathit{ais}_{new})\ \mapsto\ (lm_2, is_2) \end{array}}{(lm, is_1, pq)/\mathit{apps}\ \underset{\textbf{Return}}{\rhd}\ \langle \mathit{dl}_{\mathit{caller}}, \mathit{aid}, \mathit{iid}, n\rangle\ (is_2, pq)}$$

R-M-P

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ \mathit{sl}\ \mathit{dll}) \\ \mathit{iid}@\mathit{dll}\ \texttt{HasInstanceLabel}\ \mathit{dl} \qquad \mathit{dl}_{\mathit{caller}} \sqsubseteq \mathit{dl} \qquad \texttt{Busy}\ (\mathit{aid}, \mathit{iid})@is \end{array}}{(lm, is, pq)/\mathit{apps}\ \underset{\textbf{Return}}{\rhd}\ \langle \mathit{dl}_{\mathit{caller}}, \mathit{aid}, \mathit{iid}, n\rangle\ (is, ((\mathit{aid}, n, \mathit{dl}_{\mathit{caller}}, \texttt{pRet}\ \mathit{iid})::pq))}$$

R-M-B

$$\frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ \mathit{sl}\ \mathit{dll}) \\ \mathit{iid}@\mathit{dll}\ \texttt{HasInstanceLabel}\ \mathit{dl} \qquad \mathit{dl}_{\mathit{caller}} \not\sqsubseteq \mathit{dl} \end{array}}{(lm, is, pq)/\mathit{apps}\ \underset{\textbf{Return}}{\rhd}\ \langle \mathit{dl}_{\mathit{caller}}, \mathit{aid}, \mathit{iid}, n\rangle\ (is, pq)}$$

Figure 4.25: The operational semantics for the **Return** command.

RRA-AIS-EM

$$\frac{}{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ []\ \mapsto\ []}$$

RRA-AIS-RM

$$\frac{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ais_1\ \mapsto\ ais_2}{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ((iid', m, s, \texttt{Some}\ (aid, iid))::ais_1)\ \mapsto\ ais_2}$$

RRA-AIS-SKIP

$$\frac{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ais_1\ \mapsto\ ais_2 \qquad aid \neq aid_2 \vee iid \neq iid_2}{\begin{array}{l}\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ((iid', m, s, \texttt{Some}\ (aid_2, iid_2))::ais_1) \\ \qquad \mapsto\ ((iid', m, s, \texttt{Some}\ (aid_2, iid_2))::ais_2)\end{array}}$$

RRA-AIS-NONE

$$\frac{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ais_1\ \mapsto\ ais_2}{\texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ((iid', m, s, \texttt{None})::ais_1)\ \mapsto\ ((iid', m, s, \texttt{None})::ais_2)}$$

RRA-IS-EM

$$\frac{}{\texttt{RemoveRA}\ (aid, iid)\ \texttt{from}\ []\ \mapsto\ []}$$

RRA-IS

$$\frac{\texttt{RemoveRA}\ (aid, iid)\ \texttt{from}\ is_1\ \mapsto\ is_2 \qquad \texttt{RemoveRA}_{as}\ (aid, iid)\ \texttt{from}\ ais_1\ \mapsto\ ais_2}{\texttt{RemoveRA}\ (aid, iid)\ \texttt{from}\ (ais_1::is_1)\ \mapsto\ (ais_2::is_2)}$$

Figure 4.26: The rules for removing return addresses to a specific instance from all instances in an instance set.

write $\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2$ for removing pending return requests and return addresses of an app instance from the pending queue. The definitions of these judgments are listed in Figs. 4.26 and 4.27.

The EXIT n command When an app instance exits, we need to remove it from the instances set, adjust the app's label map, and remove all references to it from the system. For single-instance apps the label map entry is switched from launched to unlaunched and the app-instance-set is set to an empty list. As for multi-instance apps, the instance's entry in the app's dynamic-label-list and app-instance-set must be removed. Figure 4.28 list the operational semantics rules for executing an **EXIT** command. Note that these rules only handle the cleanup, not the return

$$\textsc{RP-Em}\quad \frac{}{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ []\ \mapsto\ []}$$

$$\textsc{RP-C-None}\quad \frac{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2}{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ ((aid', n, dl, \texttt{pCall None})::pq_1)\ \mapsto\ ((aid', n, dl, \texttt{pCall None})::pq_2)}$$

$$\textsc{RP-C-Skip}\quad \frac{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2 \qquad aid \neq aid_2 \lor iid \neq iid_2}{\begin{array}{c}\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ ((aid', n, dl, \texttt{pCall}\ (\texttt{Some}\ (aid_2, iid_2)))::pq_1)\\ \mapsto\ ((aid', n, dl, \texttt{pCall}\ (\texttt{Some}\ (aid_2, iid_2)))::pq_2)\end{array}}$$

$$\textsc{RP-C-Rm}\quad \frac{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2}{\begin{array}{c}\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ ((aid', n, dl, \texttt{pCall}\ (\texttt{Some}\ (aid, iid)))::pq_1)\\ \mapsto\ ((aid', n, dl, \texttt{pCall None})::pq_2)\end{array}}$$

$$\textsc{RP-R-Skip}\quad \frac{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2 \qquad aid \neq aid_2 \lor iid \neq iid_2}{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ ((aid_2, n, dl, \texttt{pRet}\ iid_2)::pq_1)\ \mapsto\ ((aid_2, n, dl, \texttt{pRet}\ iid_2)::pq_2)}$$

$$\textsc{RP-R-Rm}\quad \frac{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1\ \mapsto\ pq_2}{\texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ ((aid, n, dl, \texttt{pRet}\ iid)::pq_1)\ \mapsto\ pq_2}$$

Figure 4.27: The rules for removing return addresses and pending return requests of a specific instance from the pending queue.

$$\textsc{Rem-Con}\quad \frac{\texttt{Remove}\ iid@(dll_1, ais_1) \ \mapsto\ (dll_2, ais_2)}{\texttt{Remove}\ iid@((d{::}dll_1), (inst{::}ais_1)) \ \mapsto\ ((d{::}dll_2), (inst{::}ais_2))}$$

$$\textsc{Rem}\quad \frac{}{\texttt{Remove}\ iid@(((iid, l){::}dll_1), ((iid, m, s, ra){::}ais_1)) \ \mapsto\ (dll_2, ais_2)}$$

$$\text{E-S}\quad \frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl) \\ \texttt{RemoveRA}\ (aid, iid)\ \texttt{from}\ is_1 \ \mapsto\ is' \\ \texttt{Update}\ aid@(lm_1, is')\ \texttt{with}\ (\text{SingleInstanceUnlaunched}\ sl\ iid, []) \ \mapsto\ (lm_2, is_2) \\ \texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1 \ \mapsto\ pq_2 \end{array}}{(lm_1, is_1, pq_1)/apps \underset{\textbf{Exit}}{\rhd} \langle aid, iid\rangle\ (lm_2, is_2, pq_2)}$$

$$\text{E-M}\quad \frac{\begin{array}{c} aid@lm\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll) \\ \texttt{RemoveRA}\ (aid, iid)\ \texttt{from}\ is_1 \ \mapsto\ is' \qquad \texttt{Remove}\ iid@(dll, is'\lceil aid\rfloor_{[]}) \ \mapsto\ (dll', ais') \\ \texttt{Update}\ aid@(lm_1, is')\ \texttt{with}\ (\text{MultiInstance}\ sl\ dll', ais') \ \mapsto\ (lm_2, is_2) \\ \texttt{RemovePQ}\ (aid, iid)\ \texttt{from}\ pq_1 \ \mapsto\ pq_2 \end{array}}{(lm_1, is_1, pq_1)/apps \underset{\textbf{Exit}}{\rhd} \langle aid, iid\rangle\ (lm_2, is_2, pq_2)}$$

Figure 4.28: The operational semantics for the **Exit** command.

part of the **Exit**, which is handled as a normal return (refer to the second to rule Exec-E1 in Fig. 4.20).

4.2.4.3 Processing pending queue requests $(lm_1, is_1, pq_1)/apps \Longrightarrow_{\text{pq}} (lm_2, is_2, pq_2)$

A pending request, from the pending queue, is processed by invoking the corresponding command execution semantics detailed in Sec. 4.2.4.2. We define this step, $\Longrightarrow_{\text{pq}}$, using an auxiliary judgment `Process` $(lm_1, is_1, pq_1', pq_1)/apps \ \mapsto\ (lm_2, is_2, pq_2)$, which states that exactly one entry in the pending queue pq_1 is processed, and that the pending queue composed of $(pq_1' \ {+}{+}\ pq_1)$ will step to pq_2 as a result of this processing. pq_1' is essentially the portion of the original pending queue that we *skipped* before getting to the entry we are processing.[4] The

[4]We had to resort to this indirect approach in order to have the whole pending queue as an input and output to the command execution judgment. This restriction facilitate the noninterference proof by enabling the reuse of

$$\textsc{PP-Next}\quad \frac{\texttt{Process}\ (lm_1, is_1, (pq_1' \mathbin{++} [p]), pq_1)/apps \ \mapsto\ (lm_2, is_2, pq_2)}{\texttt{Process}\ (lm_1, is_1, pq_1', (p{::}pq_1))/apps \ \mapsto\ (lm_2, is_2, pq_2)}$$

$$\textsc{PP-Call}\quad \frac{(lm_1, is_1, (pq' \mathbin{++} pq))/apps \ \underset{\textsc{Call}}{\rhd} \langle dl, aid, n, ra\rangle\ (lm_2, is_2, (pq' \mathbin{++} pq))}{\texttt{Process}\ (lm_1, is_1, pq', ((aid, n, dl, \texttt{pCall}\ ra){::}pq))/apps \ \mapsto\ (lm_2, is_2, (pq' \mathbin{++} pq))}$$

$$\textsc{PP-Ret}\quad \frac{(lm, is_1, (pq' \mathbin{++} pq))/apps \ \underset{\textsc{Return}}{\rhd} \langle dl, aid, n, iid\rangle\ (lm, is_2)(pq' \mathbin{++} pq)}{\texttt{Process}\ (lm, is_1, pq', ((aid, n, dl, \texttt{pRet}\ iid){::}pq))/apps \ \mapsto\ (lm, is_2, (pq' \mathbin{++} pq))}$$

$$\textsc{PQ}\quad \frac{\texttt{Process}\ (lm_1, is_1, [], pq_1)/apps \ \mapsto\ (lm_2, is_2, pq_2)}{(lm_1, is_1, pq_1)/apps \Longrightarrow_{\text{pq}} (lm_2, is_2, pq_2)}$$

Figure 4.29: The operational semantics for processing the pending queue.

exact operational semantics are in Fig. 4.29. The first rule (PP-NEXT) simply states that we can always skip one more entry from the original pending queue by moving it to the skipped queue. Rules PP-CALL and PP-RET define how the entry at the head of the remaining queue is processed by invoking the appropriate command execution judgment.

4.3 Noninterference Proof

In order to validate our model, in terms of IFC security, we will prove that it satisfies the noninterference property. The noninterference property, in general, states that if we start with two systems that are exactly the same, except that one contains protected secrets not present in the other system, they should be able to take similar steps such that they remain equivalent as viewed by an attacker. In other words, the attacker, who can only observe or influence the low security parts of a system, won't be able to differentiate between the two systems. Therefore, when the same attacker interacts with a single system, it won't be able to know its protected

previously proven lemmas as is.

secrets or whether these secrets even exists. In our noninterference definition, we start with an arbitrary system that may or may not contain secrets. Then, we use a special function that removes the high parts of the system, i.e., the parts that according to the security policy, this attacker should not be able to access. We call this special function a *projection*, and we call the resulting system the *projected system*.

The attacker We cannot define any security property without defining an attacker model. In our setting, we need to define the attacker's *security level*, i.e., its label k from the security lattice. When we say that the attacker label is a k, we mean that the attacker can access any secret or privilege with label l if and only if $l \sqsubseteq k$. We call apps with such labels *attacker-accessible* apps. In practice that means the attacker is in control of one or more of these apps. That being said, and since we allow apps to raise their labels, an attacker could raise the label of one of the apps under his or her control and access any secret. However, the attacker should never be able to declassify these secrets and, for example, send them through the Internet APIs.

We will start this section by introducing the projection function in Sec. 4.3.1. Then, in Sec. 4.3.2, we will describe the *well-configured* system property, which ensures that a system instance is configured as expected by the operational semantics and the noninterference proof. In Sec. 4.3.3, we define the noninterference property and prove it for the simplified model.

4.3.1 The Projection $\bowtie^{k}_{sys}(S)$

The goal of the projection function is to *purge* the system of any parts that should not be accessible by the attacker.[5] It's a very critical part of the noninterference definition, as we use it to define the projected system. If ill-defined, the projection function can result in an unsecure or less flexible property. A projection function that does not remove all attacker-accessible parts of a system instance will result in a less secure definition. In the extreme, a projection that does not remove anything will result in a noninterference property that admits any system because any

[5]It's often called the *purge function* in the literature.

system should be able to simulate itself. On the other hand, a projection function that removes more parts than necessary might result in a definition that is either secure but not quite useful, as it might not admit some of the secure systems; or insecure because it misrepresents the attacker capabilities, i.e., models a weaker attacker model[6].

The projection function, $\bowtie^k_{sys}(S)$, which is parametrized with the attacker label k, takes a system instance as an input and returns the same instant but with all the high parts removed. We define the projection function in Figs. 4.30 and 4.31. The definitions in these two figures use pattern-matching where the first pattern to match the arguments is used. For example, when defining the function $\bowtie^k_{is}$, the first pattern will be used whenever both the second and third arguments are not empty lists; otherwise, the second, catch all, pattern will be used. In general, the projection will remove all high (i.e., not attacker-accessible) instances from the system, including their label map entries and any references to them in the system. For multi-instance apps, this means that any instance with a dynamic label dl, where $dl \not\sqsubseteq k$, will be removed from both the app-instance-set and the dynamic-label-list of the app. For single-instance apps, we look at the static label of unlaunched apps and the dynamic label of launched ones, say l. If $l \not\sqsubseteq k$, the app will be removed from the system: its app-instance-set will be emptied, and the its label map entry will be switched to NA. The helper functions, h_1, h_2, h_3, and h_4 are used to transform the output of underlying functions to the format needed by the calling function. For example, the function $\bowtie^k_{pq}\langle lm, lm\rangle(pq)$ calls itself and the function $\bowtie^k_{pendR}$. The helper function h_4 is used to combine the two outputs and return the resulting queue.

The projection function will rename the IDs of all instances of multi-instance apps. This renaming is needed to unify the IDs when comparing the projected system and the original one in the simulation lemma proof (Sec. 4.3.3), as the comparison will reapply the projection to both systems (see the definition of $\approx_{\mathrm{k}}$ in Fig. 4.34). Otherwise, if the two systems diverge in terms of IDs, because of the activity of high instances (i.e., instances that are not accessible to the attacker), they won't be able to simulate each other. The IDs are numbered sequentially, starting

[6]Weaker in the sense that it has access to less parts of the system.

from 0, and ignoring high instances. For example, if app A, has four instances numbered 1, 2, 4, and 5, with instance 4 being high, they will be renamed as follows: $1 \to 0$, $2 \to 1$, $5 \to 2$. Instance 4 does not need to be renamed because it will be removed by the projection.

The projection function $\bowtie^k_{sys}(apps, lm, is, pq)$ is defined using the three functions $\bowtie^k_{lm}(lm)$, $\bowtie^k_{is}\langle lm\rangle(lm, is)$, and $\bowtie^k_{pq}\langle lm, lm\rangle(pq)$, which are applied to the label map, the instance set, and the pending queue, respectively.

The label map projection function, $\bowtie^k_{lm}(lm)$, filters the high label map entries by applying the function $\bowtie^k_{alm}(alm)$ to each entry. If an entry is not available, i.e, NA, nothing is done. For unlaunched single-instance apps, the static label is compared to the attacker's to decide whether the entry should be removed (by returning NA). The dynamic label is used if the single-instance app is launched, which is fine because we make sure that the static label is never lower than the dynamic label (see Sec. 4.3.2). Therefore, it will never be the case that a single-instance app with a low label is removed by the projection function because its dynamic label is high. For multi-instance apps, the static label is ignored by the projection function because we never remove multi-instance apps completely. An individual instance of a multi-instance app is removed if the instance's dynamic label is high. The function $\bowtie^k_{dll}\langle n\rangle(dll)$ perform this removal and rename the instance IDs as discussed above.

The instance set projection function, $\bowtie^k_{is}\langle \hat{lm}\rangle(lm, is)$, pairs each app-instance-set with the corresponding app-label-map and then applies $\bowtie^k_{ais}\langle \hat{\hat{lm}}\rangle(alm, ais)$ to each pair. The parameter $\hat{lm}$ of the function $\bowtie^k_{is}\langle \hat{lm}\rangle(lm, is)$, is just a copy of the complete label map that is needed for the projection return address. It's treated as separate parameter to keep it whole as the function recursively breaks down label map. The function $\bowtie^k_{ais}\langle \hat{\hat{lm}}\rangle(alm, ais)$ projects an app-instance-set based on the corresponding app-label-map similar to $\bowtie^k_{alm}(alm)$ function discussed above. If an app-instance-set needs to be removed, an empty list is returned; otherwise the result of applying the function $\bowtie^k_{inst}\langle lm\rangle(ais)$ to all the instances in it is returned. For multi-instance apps, the function $\bowtie^k_{ais,mi}\langle n+1, lm\rangle(dll, ais)$ is used to filter high instances and both rename the IDs and apply $\bowtie^k_{inst}\langle lm\rangle(ais)$ to the remaining instances. The function $\bowtie^k_{inst}\langle lm\rangle(inst)$

returns the instance after applying $\Join_{ra}^{k}\langle lm\rangle(ra)$ to its return address. The function $\Join_{ra}^{k}\langle lm\rangle(ra)$ removes the return address (by returning `None`) if it points to an instance with a high dynamic label, and rename its instance ID otherwise.

The projection function also filters the pending queue by removing requests destined to low instances and the return address in call requests if it refers to a high instance. A request made to a high instance must be removed because such a request either has been blocked, if it originated from a low instance, or should have never existed if it originated from a high instance. Note that the label map is passed twice to the $\Join_{pq}$ and $\Join_{pend}$ functions to facilitate the proof only. One copy, $\hat{lm}$, is only used to project the return addresses, and the other, lm, is used to project the actual requests. In fact, we could have it as one argument only but with a somewhat longer proof.

4.3.2 Well-Configured System Instances

A *well-configured* system instance is one that satisfies specific conditions. These conditions are an *invariant* of the system. We write $\models_{sys}\ S$ to state that the system instance S is well-configured. This invariant is designed such that it will always be satisfied as the system takes steps according to the operational semantics. While the syntax does restrict the configuration of the system instances, not all these configurations are valid. For example, a launched single-instance app must have exactly one instance in its app-instance-set, but this is not enforced by the syntax rules. In addition, the well-configured invariant, defined in Figs. 4.32 and 4.33, includes conditions that are not necessary for the stability of the system but needed to ensure the security of the system, thus allowing us to prove noninterference. For example, we require that for single-instance apps, in a well-configured system, the static label must never be “lower” than the dynamic label, i.e, $dl \sqsubseteq sl$.

$$
\begin{array}{ll}
\text{getALM}(\mathit{emptyL}, \mathit{aid}) & = \texttt{None} \\
\text{getALM}((\mathit{alm}{::}\mathit{lm}), 0) & = \texttt{Some}\ (\mathit{alm}) \\
\text{getALM}((\mathit{alm}{::}\mathit{lm}), \mathit{aid}) & = \text{getALM}(\mathit{lm}, \mathit{aid} - 1) \\
\hline
\bowtie^k_{\mathit{iid,dll}}\langle n, []\rangle(\mathit{iid}) & = \texttt{None} \\
\bowtie^k_{\mathit{iid,dll}}\langle n, ((\mathit{iid}_2, l){::}\mathit{dll})\rangle(\mathit{iid}) & = \texttt{if}\ \mathit{iid} = \mathit{iid}_2\ \texttt{then} \\
 & \quad\quad \texttt{if}\ l \sqsubseteq k\ \texttt{then}\ \texttt{Some}\ (n)\ \texttt{else}\ \texttt{None} \\
 & \quad \texttt{else} \\
 & \quad\quad \texttt{if}\ l \sqsubseteq k\ \texttt{then} \\
 & \quad\quad\quad \bowtie^k_{\mathit{iid,dll}}\langle n+1, \mathit{dll}\rangle(\mathit{iid}) \\
 & \quad\quad \texttt{else} \\
 & \quad\quad\quad \bowtie^k_{\mathit{iid,dll}}\langle n, \mathit{dll}\rangle(\mathit{iid}) \\
\hline
\bowtie^k_{\mathit{iid,alm}}\langle \text{SIL}\ \mathit{iid}'\ \mathit{sl}\ \mathit{dl}\rangle(\mathit{iid}) & = \texttt{if}\ \mathit{iid} = \mathit{iid}' \wedge \mathit{dl} \sqsubseteq k\ \texttt{then}\ (\texttt{Some}\ (\mathit{iid}))\ \texttt{else}\ \texttt{None} \\
\bowtie^k_{\mathit{iid,alm}}\langle \text{MI}\ \mathit{sl}\ \mathit{dll}\rangle(\mathit{iid}) & = \bowtie^k_{\mathit{iid,dll}}\langle 0, \mathit{dll}\rangle(\mathit{iid}) \\
\bowtie^k_{\mathit{iid,alm}}\langle _\rangle(\mathit{iid}) & = \texttt{None} \\
\hline
h_1(\texttt{None}, \mathit{aid}) & = \texttt{None} \\
h_1(\texttt{Some}\ (\mathit{iid}), \mathit{aid}) & = \texttt{Some}\ (\mathit{aid}, \mathit{iid}) \\
\bowtie^k_{\mathit{iid,lm}}\langle \mathit{lm}\rangle(\mathit{aid}, \mathit{iid}) & = h_1(\bowtie^k_{\mathit{iid,alm}}\langle \text{getALM}(\mathit{aid}, \mathit{lm})\rangle(\mathit{iid})) \\
\hline
\bowtie^k_{\mathit{ra}}\langle \mathit{lm}\rangle(\texttt{None}) & = \texttt{None} \\
\bowtie^k_{\mathit{ra}}\langle \mathit{lm}\rangle(\texttt{Some}\ (\mathit{aid}, \mathit{iid})) & = \bowtie^k_{\mathit{iid,lm}}\langle \mathit{lm}\rangle(\mathit{aid}, \mathit{iid}) \\
\hline
\bowtie^k_{\mathit{inst}}\langle \mathit{lm}\rangle((\mathit{iid}, m, s, \mathit{ra})) & = (\mathit{iid}, m, s, \bowtie^k_{\mathit{ra}}\langle \mathit{lm}\rangle(\mathit{ra})) \\
\hline
r((\mathit{iid}, m, s, \mathit{ra}), n) & = (n, m, s, \mathit{ra}) \\
\bowtie^k_{\mathit{ais,mi}}\langle n, \mathit{lm}\rangle((_, l{::}\mathit{dll}), (r(\mathit{inst}){::}\mathit{ais})) & = \texttt{if}\ l \sqsubseteq k\ \texttt{then} \\
 & \quad\quad (\bowtie^k_{\mathit{inst}}\langle \mathit{lm}\rangle(r(\mathit{inst}, n)){::}\bowtie^k_{\mathit{ais,mi}}\langle n+1, \mathit{lm}\rangle(\mathit{dll}, \mathit{ais})) \\
 & \quad \texttt{else} \\
 & \quad\quad \bowtie^k_{\mathit{ais,mi}}\langle n+1, \mathit{lm}\rangle(\mathit{dll}, \mathit{ais}) \\
\bowtie^k_{\mathit{ais,mi}}\langle n, \mathit{lm}\rangle(_, _) & = [] \\
\hline
\bowtie^k_{\mathit{ais}}\langle \mathit{lm}\rangle(\text{SIU}\ \mathit{sl}\ \mathit{iid}_{\mathit{prev}}, \mathit{ais}) & = \texttt{if}\ \mathit{sl} \sqsubseteq k\ \texttt{then}\ \texttt{map}(\bowtie^k_{\mathit{inst}}\langle \mathit{lm}\rangle, \mathit{ais})\ \texttt{else}\ [] \\
\bowtie^k_{\mathit{ais}}\langle \mathit{lm}\rangle(\text{SIL}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}, \mathit{ais}) & = \texttt{if}\ \mathit{dl} \sqsubseteq k\ \texttt{then}\ \texttt{map}(\bowtie^k_{\mathit{inst}}\langle \mathit{lm}\rangle, \mathit{ais})\ \texttt{else}\ [] \\
\bowtie^k_{\mathit{ais}}\langle \mathit{lm}\rangle(\text{MI}\ \mathit{sl}\ \mathit{dll}, \mathit{ais}) & = \bowtie^k_{\mathit{ais,mi}}\langle n, \mathit{lm}\rangle(\mathit{dll}, \mathit{ais}) \\
\bowtie^k_{\mathit{ais}}\langle \mathit{lm}\rangle(\text{NA}, \mathit{ais}) & = [] \\
\hline
\bowtie^k_{\mathit{is}}\langle \hat{\mathit{lm}}\rangle((\mathit{alm}{::}\mathit{lm}), (\mathit{ais}{::}\mathit{is})) & = (\bowtie^k_{\mathit{ais}}\langle \hat{\mathit{lm}}\rangle(\mathit{alm}, \mathit{ais}){::}\bowtie^k_{\mathit{is}}\langle \hat{\mathit{lm}}\rangle(\mathit{lm}, \mathit{is})) \\
\bowtie^k_{\mathit{is}}\langle \hat{\mathit{lm}}\rangle(_, _) & = [] \\
\hline\hline
\bowtie^k_{\mathit{dll}}\langle n\rangle([]) & = [] \\
\bowtie^k_{\mathit{dll}}\langle n\rangle(((\mathit{iid}, l){::}\mathit{dll})) & = \texttt{if}\ l \sqsubseteq k\ \texttt{then}\ ((n, l){::}\bowtie^k_{\mathit{dll}}\langle n+1\rangle(\mathit{dll}))\ \texttt{else}\ \bowtie^k_{\mathit{dll}}\langle n\rangle(\mathit{dll}) \\
\hline
\bowtie^k_{\mathit{alm}}(\text{SIU}\ \mathit{sl}\ \mathit{iid}_{\mathit{prev}}) & = \texttt{if}\ \mathit{sl} \sqsubseteq k\ \texttt{then}\ \text{SIU}\ \mathit{sl}\ \mathit{iid}_{\mathit{prev}}\ \texttt{else}\ \text{NA} \\
\bowtie^k_{\mathit{alm}}(\text{SIL}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}) & = \texttt{if}\ \mathit{dl} \sqsubseteq k\ \texttt{then}\ \text{SIL}\ \mathit{iid}\ \mathit{sl}\ \mathit{dl}\ \texttt{else}\ \text{NA} \\
\bowtie^k_{\mathit{alm}}(\text{MI}\ \mathit{sl}\ \mathit{dll}) & = \text{MI}\ \mathit{sl}\ \bowtie^k_{\mathit{dll}}\langle 0\rangle(\mathit{dll}) \\
\bowtie^k_{\mathit{alm}}(\text{NA}) & = \text{NA} \\
\hline
\bowtie^k_{\mathit{lm}}(\mathit{lm}) & = \texttt{map}(\bowtie^k_{\mathit{alm}}, \mathit{lm})
\end{array}
$$

Figure 4.30: The projection function (part 1 of 2).

$$
\begin{array}{ll}
h_2(\texttt{None}) & = \texttt{None} \\
h_2(\texttt{Some }(iid)) & = \texttt{Some }(\texttt{pRet }iid) \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{SIU } sl\ \_, \texttt{pCall } ra) & = \texttt{if } sl \sqsubseteq k \texttt{ then Some } (\texttt{pCall } \bowtie^k_{ra}\langle\hat{lm}\rangle(ra)) \texttt{ else None} \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{SIL } \_\ dl\ \_, \texttt{pCall } ra) & = \texttt{if } dl \sqsubseteq k \texttt{ then Some } (\texttt{pCall } \bowtie^k_{ra}\langle\hat{lm}\rangle(ra)) \texttt{ else None} \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{MI } sl\ \_, \texttt{pCall } ra) & = \texttt{if } sl \sqsubseteq k \texttt{ then Some } (\texttt{pCall } \bowtie^k_{ra}\langle\hat{lm}\rangle(ra)) \texttt{ else None} \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{SIU } \_\ \_, \texttt{pRet } \_) & = \texttt{None} \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{SIL } \_\ dl\ iid', \texttt{pRet } iid) & = \texttt{if } iid' = iid \wedge dl \sqsubseteq k \texttt{ then} \\
 & \quad\ \texttt{Some } (\texttt{pRet } iid) \\
 & \ \ \texttt{else} \\
 & \quad\ \texttt{None} \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{MI } sl\ dll, \texttt{pRet } iid) & = h_2(\bowtie^k_{iid,dll}\langle 0, dll\rangle(iid)) \\
\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{NA}, \_) & = \texttt{None} \\
\hline
h_3(\texttt{None}, \_, \_, \_) & = \texttt{None} \\
h_3(\texttt{Some }(pt), aid, n, dl) & = \texttt{Some }(aid, n, dl, pt) \\
\bowtie^k_{pendR}\langle\hat{lm}, lm\rangle((aid, n, dl, pt)) & = h_3(\bowtie^k_{pend}\langle\hat{lm}\rangle(\text{getALM}(lm, aid), pt)) \\
\hline
h_4(\texttt{None}, pq) & = pq \\
h_4(\texttt{Some }(p), pq) & = (p{::}pq) \\
\bowtie^k_{pq}\langle\hat{lm}, lm\rangle([]) & = [] \\
\bowtie^k_{pq}\langle\hat{lm}, lm\rangle((p{::}pq)) & = h_4(\bowtie^k_{pendR}\langle\hat{lm}, lm\rangle(p), \bowtie^k_{pq}\langle\hat{lm}, lm\rangle(pq)) \\
\hline\hline
\bowtie^k_{sys}(apps, lm, is, pq) & = (apps, \bowtie^k_{lm}(lm), \bowtie^k_{is}\langle lm\rangle(lm, is), \bowtie^k_{pq}\langle lm, lm\rangle(pq))
\end{array}
$$

Figure 4.31: The projection function (part 2 of 2).

$$\frac{}{iid\ \texttt{InDLL}\ ((iid, l){::}dll)} \qquad \frac{iid\ \texttt{InDLL}\ dll}{iid\ \texttt{InDLL}\ (d{::}dll)}$$

$$\frac{}{iid\ \texttt{InALM}\ (\text{SingleInstanceLaunched } sl\ dl\ iid)} \qquad \frac{iid\ \texttt{InDLL}\ dll}{iid\ \texttt{InALM}\ (\text{MultiInstance } iid\ dll)}$$

$$\frac{iid\ \texttt{InALM}\ alm}{(0, iid)\ \texttt{InLM}\ (alm{::}lm)} \qquad \frac{(aid, iid)\ \texttt{InLM}\ lm}{(aid+1, iid)\ \texttt{InLM}\ (alm{::}lm)} \qquad \frac{}{iid\ \texttt{NotIn}\ []}$$

$$\frac{iid\ \texttt{NotIn}\ dll \qquad iid \neq iid'}{iid\ \texttt{NotIn}\ (iid', l{::}dll)}$$

Figure 4.32: Helper definitions for the well-configured system judgments.

$$\frac{}{lm \models_{ra} \texttt{None}} \qquad \frac{(aid, iid)\ \texttt{InLM}\ lm}{lm \models_{ra} \texttt{Some}\ (aid, iid)}$$

$$\frac{lm \models_{ra} ra \qquad \exists\Delta \cdot \Delta \vdash_{mem} m \wedge \Delta; [] \vdash_s s}{lm \models_{inst} (iid, m, s, ra)} \qquad \frac{}{lm \models_{ais} []}$$

$$\frac{lm \models_{inst} inst \qquad lm \models_{ais} ais}{lm \models_{ais} (inst::ais)} \qquad \frac{}{lm \models_{is} []} \qquad \frac{lm \models_{ais} ais \qquad lm \models_{is} is}{lm \models_{is} (ais::is)}$$

$$\frac{}{\models_{apps} []} \qquad \frac{[]; [\texttt{ARG : Tnat}] \vdash_s s \qquad \models_{apps} apps}{\models_{apps} (s::apps)} \qquad \frac{}{\models_{dll} ([], [])}$$

$$\frac{iid\ \texttt{NotIn}\ dll \qquad \models_{dll} (dll, ais)}{\models_{dll} ((iid, l::dll), ((iid, m, s, ra)::ais))} \qquad \frac{}{\models_{alm} ((\text{SingleInstanceUnlaunched}\ sl\ iid_{prev}), [])}$$

$$\frac{dl \sqsubseteq sl}{\models_{alm} ((\text{SingleInstanceLaunched}\ sl\ dl\ iid), [(iid, m, s, ra)])}$$

$$\frac{\models_{dll} (dll, ais)}{\models_{alm} ((\text{MultiInstance}\ sl\ dll), ais)} \qquad \frac{}{\models_{alm} (\text{NA}, [])} \qquad \frac{}{\models_{lm} ([], [])}$$

$$\frac{\models_{alm} (alm, ais) \qquad \models_{lm} (lm, is)}{\models_{lm} ((alm::lm), (ais::is))} \qquad \frac{}{\hat{lm}; lm \models_{pend} (aid, n, dl, \texttt{pCall None})}$$

$$\frac{(aid, iid)\ \texttt{InLM}\ \hat{lm}}{\hat{lm}; lm \models_{pend} (aid', n, dl, \texttt{pCall Some}\ (aid, iid))} \qquad \frac{(aid, iid)\ \texttt{InLM}\ lm}{\hat{lm}; lm \models_{pend} (aid, n, dl, \texttt{pRet}\ iid)}$$

$$\frac{}{\hat{lm}; lm \models_{pq} []} \qquad \frac{\hat{lm}; lm \models_{pend} p \qquad \hat{lm}; lm \models_{pq} pq}{\hat{lm}; lm \models_{pq} (p::pq)}$$

$$\frac{\models_{apps} apps \qquad \models_{lm} (lm, is) \qquad lm \models_{is} is \qquad lm; lm \models_{pq} pq \qquad \texttt{len}(is) = \texttt{len}(lm)}{\models_{sys} (apps, lm, is, pq)}$$

Figure 4.33: Well-configured system.

4.3.3 The Noninterference Property

We formally define the noninterference property in Definition 3. The definition essentially states that for if a well-configured system instance S takes several steps becoming S' and producing trace t, then it must be the case that the projected system, $\bowtie^k_{sys}(S)$, can take similar steps producing a trace t_L that is equivalent to the projection of the original trace t.

Definition 3 (Noninterference)**.** *A model $\mathcal{M}(-)$ satisfies noninterfrence if, for any security lattice $\mathcal{L} = (L, \sqsubseteq)$, any attacker label $k \in L$, any system instances $S, S' \in \mathcal{M}(\mathcal{L})$, and any trace t:*

If $\models_{sys}$ S and $S \xRightarrow{t}* \ S'$ then

exists $S'_L \in \mathcal{M}(\mathcal{L})$ and t_L s.t.,

$\bowtie^k_{sys}(S) \xRightarrow{t_L}* \ S'_L$ and $\bowtie^k_t(t) \equiv t_L$.

In order to prove the noninterference theorem, we first define a bisimulation relation between two system instances. The relation $\approx_\mathrm{k}$, defined in Fig.4.34, essentially says that the two systems are well-configured and appear equivalent to an attacker with security level k. Fig. 4.34 also defines system traces and the multi-step judgment. A trace t is a list of output and label pairs produced when a system instance takes some number of steps. Remember that the label represents the security level of the instance that produced the output at the time of the output event. We also define a projection function for traces, which filters a trace by omitting any none attacker-accessible outputs.

We next prove the k-bisimulation lemma (Lemma 1), which states that the $\approx_\mathrm{k}$ is a weak k-bisimulation relation in our system [73]. This lemma is critical for our noninterference proof, as the proof follows easily from it. However, proving it requires the examination of a large number of cases and sub-cases, which is a long error-prone process. This limitation is the main reason for our use of the Coq proof-assistant to encode our model and prove this lemma. Note that the relation $\approx_\mathrm{k}$ is symmetric, and thus it is enough to prove that it is a k-simulation to prove that it's

Traces and their projection and equivalence

$$TRACE ::= \texttt{list}(N, label)$$

$$\begin{aligned} \bowtie_t^k([]) &= emptyL \\ \bowtie_t^k(((n,l)::t)) &= \texttt{if } l \sqsubseteq k \texttt{ then } (n, l::\bowtie_t^k(t)) \texttt{ else } \bowtie_t^k(t) \end{aligned}$$

$$\frac{}{[] \equiv []} \qquad \frac{t_1 \equiv t_2}{(a::t_1) \equiv (a::t_2)}$$

The bisimulation relation and system multi-step

$$\textsc{MS-nil}\ \frac{}{S \xRightarrow{[]}* S} \qquad \textsc{MS-tau}\ \frac{S_1 \xRightarrow{t}* S' \qquad S' \xRightarrow{\tau} S_2}{S_1 \xRightarrow{t}* S_2} \qquad \textsc{MS-out}\ \frac{S_1 \xRightarrow{t}* S' \qquad S' \xRightarrow{a} S_2}{S_1 \xRightarrow{(a::t)}* S_2}$$

$$\textsc{Sim}\ \frac{\models_{sys} S_1 \qquad \models_{sys} S_2 \qquad \bowtie_{sys}^k(S_1) = \bowtie_{sys}^k(S_2)}{S_1 \approx_{\mathrm{k}} S_2}$$

Figure 4.34: The bisimulation relation, multi-step, and trace equivalence and projection.

a k-bisumulation. This lemma and other notable lemmas about the behaviour of the projection function are listed below. We won't show their proofs here, as they have been proved in Coq. Lemma 2 states that the projection function preservers well-configuredness. Lemma 3 states that the projection function is idempotent, i.e., applying it more than once yields the same results as applying once.

Lemma 1 (Weak k-simulation)**.** *For any security lattice $(L, \sqsubseteq)$; any attacker label $k \in L$; any system instances S_1,S_1', and S_2; and any trace t_1, If $S_1 \approx_{\mathrm{k}} S_2$ and $S_1 \xRightarrow{t_1}* S_1'$ then exists t_2 and S_2' s.t., $S_2 \xRightarrow{t_2}* S_2'$ and $\bowtie_t^k(t_1) \equiv t_2$ and $S_1 \approx_{\mathrm{k}} S_2$.*

Lemma 2 (Projection preservation)**.** *For any security lattice $(L, \sqsubseteq)$; any attacker label $k \in L$ and any system instance S, if $\models_{sys} S$ then $\models_{sys} \bowtie_{sys}^k(S)$.*

Lemma 3 (Idempotent projection)**.** *For any security lattice $(L, \sqsubseteq)$; any attacker label $k \in L$ and any system instance S, $\bowtie_{sys}^k(S) = \bowtie_{sys}^k(\bowtie_{sys}^k(S))$.*

Now we are ready to state and proof the noninterference theorem.

Theorem 4 (Noninterference)**.** *The system model $\mathcal{M}(-)$ satisfies noninterference.*

Proof. Using Lemmas 2 and 3 we get $\models_{sys} \bowtie^{k}_{sys}(S)$ and $\bowtie^{k}_{sys}(S) = \bowtie^{k}_{sys}(\bowtie^{k}_{sys}(S))$, respectively. Then, by the definition of $\approx_{\mathrm{k}}$, we get $S \approx_{\mathrm{k}} \bowtie^{k}_{sys}(S)$. Finally, we can use Lemma 1 to conclude the theorem. $\square$

While the noninterference theorem is based on trace-equivalence, due to how we started our development, there is no obstacle, other than consistency, preventing it from being based on weak bisimilarity. In fact, the proof of Lemma 1 can be easily adapted to prove a bimsimulation-based noninterference theorem. The reason we chose trace-equivalence in Sec. 4.1.2 is to accommodate floating labels, which are omitted in this simplified model. Moreover, the way we define output events in the simplified model is slightly different from the pi-Calculus model from Sec. 4.1. In the simplified model all calls or returns produce output events that are included in the trace. In the pi-Calculus model only successful calls or returns produce outputs because we only omit the output when an intent is received. However, this difference is not consequential since, as we mentioned, we are able to prove the stronger bisimulation-based noninterference, which compares whole instances instead of just output events.

4.4 The Versioned-Labels Noninterference Property

A shortcoming of (standard) noninterference theorems for systems, like the one we proved for our enhanced Android run-time in Sec. 4.3, is that they can only be proved for the system between declassification operations. This shortcoming means that when a declassification operation is executed, the theorem says nothing about the information flow between the differently labeled parts of the system before and after the declassification. This lack of guarantees is true, even if the actual system does provide control for information flows not affected by declassification. Furthermore, such theorems will not be sufficient for our endeavor in Ch. 5, since our access control checks happen exactly at the declassification points.

Fortunately, researchers have suggested alternative definitions for the noninterference property that can account for declassification [13, 74, 75]. One example is *intransitive noninterference*, which treats declassification operations as normal operations but only allows information to flow from label A to label B if explicitly allowed by the current policy[7], as opposed to allowing any flow as long as it is not to less restricted labels, i.e., up the lattice [74].

The reason we cannot apply regular noninterference when a high app is declassified to low is that the projected system never had the high app in the first place, and thus cannot simulate the declassification operation. In general, the solution is to amend the projection function to not remove the instance (or action in the case of language-level IFC systems) that have been declassified. However, this amendment is not enough; we need to also keep all the high instances that affected ,i.e., sent information flows to, the declassified instance. In our approach, we keep the projection function as is, but modify the lattice such that the needed app instances are not removed.

In this section, we discuss the different approaches for lattice modification in Sec. 4.4.1. We then present a motivating example for our approach in Sec. 4.4.2. Next, we give the operational semantics for the declassification command in Sec. 4.4.3. Section 4.4.4 defines an extended system model to be used in the proof of the versioned-labels noninterference theorem in Sec. 4.4.5. Finally, we briefly discuss how to make an even more precise noninterference property in Sec. 4.4.6.

4.4.1 Morphing the Security Lattice

For our approach, we want to extend our noninterference definition by morphing our security levels and the partial ordering relation each time a declassification operation is performed. The morphed relation does not play any role in the execution of the actual system, it is only used within the noninterference definition. The resulting levels and relation, when used in the

[7]The policy specification language is modified to allow these explicit flow controls.

projection function, should allow the declassified instance, and any contributing instance, to persist in the projected system. Next we discuss three, increasingly stronger (i.e., provide better security guarantees), lattice morphing strategies.

Tag removal This coarse grained strategy only applies to tag-based lattices. In this strategy, when an app uses its capability to declassify a certain tag, say s, to change its label's secrecy tags set from $\{s, s_1, s_2, \ldots\}$ to $\{s_1, s_2, \ldots\}$, we remove the tag s from all labels in the system. The intuition behind this strategy is that declassification could potentially reveal any secret from anywhere in the system that was protected by this tag, and hence, it should be removed from all labels. This assumption is not precise. For example, if app A, with label $(\{s, s_1\}, \{\}, \{-s\})$ declassified itself to $(\{s_1\}, \{\}, \{-s\})$, then app B, with label $(\{s, s_2\}, \{\}, \{\})$, will be considered to have a label $(\{s_2\}, \{\}, \{\})$. App B could not have sent any information to the declassified app, and ideally should not have been affected by this particular declassification. However, this definition does not prevent app B from calling app C with the label $(\{s_2\}, \{\}, \{\})$ because their labels are considered the same after the declassification. In other words, a system will satisfy the noninterference poperty with this strategy even if it allows leaks from B to C after A declassify its label as above.

Single flow Whenever we declassify label l_1 to label l_2, we add a single additional flow or "arrow" in the lattice from l_1 to l_2. More concretely, we amend the relation $\sqsubseteq$ such that $l_1 \sqsubseteq l_2$. The intuition is that the declassification operation added, albeit temporarily, a flow from all the apps labeled l_1 to those labeled l_2. The problem with this approach is that the resulting relation is not a partial ordering, more specifically, it's not transitive nor anti-symmetric. One way to cope with this problem is to use one of the intransitive noninterference definitions. In the tag-based lattice settings, and compared to the tag removal strategy, this is a more precise definition that only adds one extra flow to the lattice. For example, if l_1 was $\{s_1, s_2, s_3\}$ and l_2 was $\{s_2, s_3\}$, this strategy will not allow for a flow from $\{s_1\}$ to $\{\}$, while the tag removal strategy will allow it (because it will be from $\{\}$ to $\{\}$, after removing s_1). The problem with this strategy is that once a certain label is declassified it will remain so indefinitely. In our system, we allow apps to raise their labels back after declassification. However, with this strategy, the raise operation will be ignored as far as the noninterference property is concerned because the label is still declassified in the morphed lattice.

Versioned Labels The versioned lattice approach captures the intuition that, when an app instance declassifies its label from l_1 to l_2, the declassification applies to only the current l_1 label but not to future instantiations of it. When an app later raises its label from, say, l_3 to l_1, the information protected by the new instance of l_1 should not flow to l_2 anymore. To satisfy this requirement, a new version of the original lattice is created whenever a declassification occurs. All labels in the system are upgraded to the new version at the time of declassification. To keep the connection between versions from the same label, a flow is inserted between the consecutive versions of a label, i.e., for any label l, there is a flow from version n of l and version $n+1$ of l. Moreover, a flow is inserted between version n of l_1 and version $n+1$ of l_2, assuming this the n^{th} declassification operation from the initial state. For example, if the original lattice was the set $\{$H,M,L$\}$ with $L \leq M$ and $M \leq H$, then the morphed lattice will as depicted in Fig. 4.35 after declassification from H to M then M to L.

In this work we will be using the versioned labels approach.

4.4.2 Motivating Example

In this section, we revisit the scenario from Sec. 2.1 to motivate our versioned labels approach. Suppose the user requests that secret file A be declassified and forwarded to the email client. The secret-file manager will retrieve the file, declassify itself from $(\{\textsf{secret-file}\}, \{\}, \{\textsf{-secret-file}\})$ to $(\{\}, \{\}, \{\textsf{-secret-file}\})$, and then send the file to the email client. After that, adhering to the important least-privilege principal, the manger should raise itself back to the original label. Later the user asks the manager to create a new secret file B. Finally, suppose that a malicious

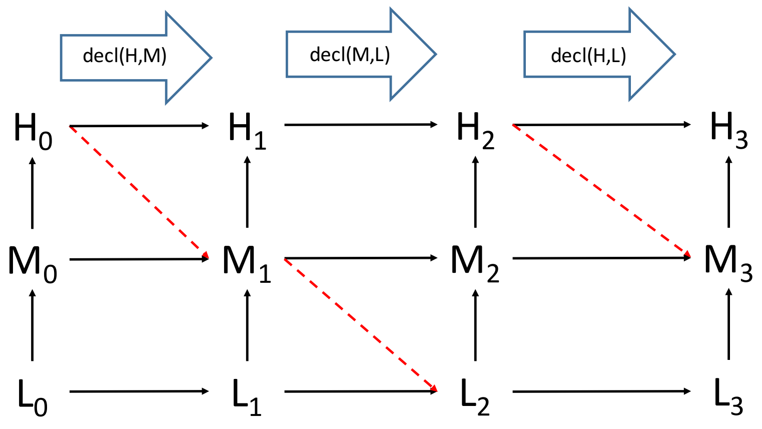

Figure 4.35: An example of a versioned lattice after two declassification operations.

app, which does not have a secret-file tag in its label, tries to retrieve file B and send it through Internet. A reasonable enforcement mechanism will prevent the malicious app from exfiltrating file B.

Consider a faulty system that "remembers" any declassification operation from l_{old} to l_{new}, and then whenever any app in the system with label l_{old} tries to call another app, it uses l_{new} instead of l_{old} to make the enforcement decisions. This system will allow the malicious app from the scenario above to access file B because it will "remember" that $(\{\textsf{secret-file}\}, \{\}, \{\textsf{-secret-file}\})$ should be treated as $(\{\}, \{\}, \{\textsf{-secret-file}\})$. Such a system does not satisfy the versioned-labels noninterference property that is based on versioned labels (defined in Sec. 4.4.5) because the new version of the label will be high. Specifically, the projection function will remove the manager app and file B, both of which now have the high new version of the label, when comparing the projected and original systems after the declassification, and the two systems could diverge from each other while still agreeing on the low parts. Thus, we won't be able to guarantee that the projected systems will be able to simulate the original when the malicious app accessed file B. For example, file B might not have been created in the projected system.

Previously proposed extensions to noninterference, such as intransitive noninterference [74] and relaxed noninterference [13], do not guarantee correct policy enforcement for this scenario probably because they don't consider label raising operations.

4.4.3 A Model with Declassification $\mathcal{M}_d(\mathcal{L})$

Here, we define the operational semantics for executing declassification (**DECLASSIFY**) commands, which were omitted from the model in Section 4.2. However, we first need to redefine the syntax and semantics to add declassification capabilities to apps. We do this, in Fig. 4.36, by redefining the apps' label map entries, $AppLabelMap$, to include declassification capabilities, $DeclCap$. A declassification capability is just a list of label pairs. Each pair, (l_1, l_2), permits the app to declassify itself from l_1 to l_2. We also add the rule SYS-EXEC-DECL, which allows for steps with declassification events, to the system step definition in Fig. 4.18.

Now, we can introduce, in Fig. 4.37, the rules to define $\rhd_{\textbf{DECLASSIFY}}$. The two rules, D-S-B and D-M-B, cover the cases where a declassification request is denied (or blocked) for the single-instance and multi-instance apps. A request is denied if either the app does not have the required declassification capability or if the new label is already below the current label in the lattice. The first rule, D-S, defines declassification for single-instance apps. It makes sure that the app has the required capability, i.e., $(dl, l) \in \delta$, and updates the app-label-map to have the new label. Rule D-M does the same but for multi-instance apps. The last two rules augment the rules in Fig. 4.20 to define the command execution for the case of the **DECLASSIFY** command whether the declassification was successful (EXEC-D1) or denied (EXEC-D2).

In addition, we extend the notion of traces to include not just output events but also declassification events in Fig. 4.38. A declassification event, $\texttt{DECL}(l_1, l_2)$, is added to the trace when an app declassifies itself from l_1 to l_2. Finally, we need to define the projection function

$$\begin{array}{lcl} \mathit{DeclCap} & ::= & \texttt{list}(\mathit{label}, \mathit{label}) \\ \mathit{AppLabelMap} & ::= & \text{SingleInstanceUnlaunched } \mathit{label}\ \mathit{IID}\ \mathit{DeclCap}\ \mid \\ & & \text{SingleInstanceLaunched } \mathit{label}\ \mathit{label}\ \mathit{IID}\ \mathit{DeclCap}\ \mid \\ & & \text{MultiInstance } \mathit{label}\ \mathit{DynamicLabelList}\ \mathit{DeclCap}\ \mid \text{NA} \end{array}$$

$$\textsc{Sys-Exec-Decl}\quad \frac{(lm_1, is_1, pq_1)/apps \overset{\texttt{DECL}(l_{old}, l_{new})}{\rhd} (lm_2, is_2, pq_2)}{(apps, lm_1, is_1, pq_1) \overset{\texttt{DECL}(l_{old}, l_{new})}{\Longrightarrow} (apps, lm_2, is_2, pq_2)}$$

Figure 4.36: Amending system syntax and semantics to add declassification capabilities.

for traces with declassification events (see Fig. 4.38). For a declassification event, the projection function will only keep the event if the new label is low (i.e., attacker-accessible). If the new label is high (i.e., not attacker-accessible), then so is the old label, which means the whole event should not be accessible to the attacker and must be removed by the projection function.

4.4.4 The Versioned System Model $\check{\mathcal{M}}_d(\mathcal{L}^n)$

Here, we introduce a slightly modified system model that incorporates versioned labels. This modified model serves as temporary representation of the system for the purpose of proving noninterference only. There are no versioned labels during normal execution. A versioned label, l^n, is a normal label that has been paired with a version number, n in this case. We start by defining the function $\omega(\mathcal{L}, t)$, which takes the original lattice and a trace to generate the versioned lattice (see Fig. 4.39). We also need relabeling functions $\Theta(\check{S})$, which takes a versioned system instance and advance all its labels to the next version, and $\Theta^n(S)$, which takes a normal system instance and convert it to a versioned instance with version n labels. Finally, we need to amend the operational semantics for declassification (from Fig. 4.37) as follows to advance the labels' version with each successful declassification operation.

$$\textsc{V-Exec-D1}\quad \frac{(lm_1, is_1, pq_1)/apps \overset{\texttt{DECL}(l_{old}, l_{new})}{\rhd} (lm_2, is_2, pq_2)}{(apps, lm_1, is_1, pq_1) \overset{\texttt{DECL}(l_{old}, l_{new})}{\Longrightarrow} \Theta(apps, lm_2, is_2, pq_2)}$$

$$\frac{\begin{array}{c}\text{D-S}\\ aid@lm_1\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl\ \delta) \qquad l \not\sqsubseteq dl \\ (dl,l) \in \delta \qquad \texttt{Update}\ aid@lm_1\ \texttt{with}\ (\text{SingleInstanceLaunched}\ iid\ sl\ l\ \delta)\ \mapsto\ lm_2\end{array}}{lm_1 \underset{\textbf{DECLASSIFY}}{\overset{(dl,l)}{\rhd}} \langle aid, iid, l\rangle\ lm_2}$$

$$\frac{\begin{array}{c}\text{D-S-B}\\ aid@lm\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl\ \delta) \qquad l \sqsubseteq dl \vee (dl,l) \notin \delta\end{array}}{lm \underset{\textbf{DECLASSIFY}}{\overset{\tau}{\rhd}} \langle aid, iid, l\rangle\ lm}$$

$$\frac{\begin{array}{c}\text{D-M}\\ aid@lm_1\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll_1\ \delta) \qquad iid@dll_1\ \texttt{HasInstanceLabel}\ dl \\ l \not\sqsubseteq dl \qquad (dl,l) \in \delta \qquad \texttt{Update}\ iid@dll_1\ \texttt{with}\ l\ \mapsto\ dll_2 \\ \texttt{Update}\ aid@lm_1\ \texttt{with}\ (\text{MultiInstance}\ sl\ dll_2\ \delta)\ \mapsto\ lm_2\end{array}}{lm_1 \underset{\textbf{DECLASSIFY}}{\overset{(dl,l)}{\rhd}} \langle aid, iid, l\rangle\ lm_2}$$

$$\frac{\begin{array}{c}\text{D-M-B}\\ aid@lm\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll\ \delta) \\ iid@dll\ \texttt{HasInstanceLabel}\ dl \qquad l \sqsubseteq dl \vee (dl,l) \notin \delta\end{array}}{lm \underset{\textbf{DECLASSIFY}}{\overset{\tau}{\rhd}} \langle aid, iid, l\rangle\ lm}$$

$$\frac{\begin{array}{c}\text{EXEC-D1}\\ (aid,iid)@is_1\ \succ_{is}\ \langle \textbf{DECLASSIFY}\ l, is_2, ra\rangle \qquad lm_1 \underset{\textbf{DECLASSIFY}}{\overset{(dl,l)}{\rhd}} \langle aid, iid, l\rangle\ lm_2\end{array}}{(lm_1, is_1, pq)/apps \overset{\texttt{DECL}(dl,l)}{\rhd} (lm_2, is_2, pq)}$$

$$\frac{\begin{array}{c}\text{EXEC-D2}\\ (aid,iid)@is_1\ \succ_{is}\ \langle \textbf{DECLASSIFY}\ l, is_2, ra\rangle \qquad lm_1 \underset{\textbf{DECLASSIFY}}{\overset{\tau}{\rhd}} \langle aid, iid, l\rangle\ lm_2\end{array}}{(lm_1, is_1, pq)/apps \overset{\tau}{\rhd} (lm_2, is_2, pq)}$$

Figure 4.37: The operational semantics for the **DECLASSIFY** command.

$$\begin{array}{lcl} EVENT & ::= & \texttt{OUT}(N, label) \mid \texttt{DECL}(label, label) \\ TRACE & ::= & \texttt{list}\, EVENT \end{array}$$

$$\begin{array}{ll} \bowtie_t^k([]) & = [] \\ \bowtie_t^k((\texttt{OUT}(n,l)::t)) & = \texttt{if}\ l \sqsubseteq k\ \texttt{then}\ (\texttt{OUT}(n,l)::\bowtie_t^k(t))\ \texttt{else}\ \bowtie_t^k(t) \\ \bowtie_t^k((\texttt{DECL}(l_{old}, l_{new})::t)) & = \texttt{if}\ l_{new} \sqsubseteq k\ \texttt{then}\ (\texttt{DECL}(l_{old}, l_{new})::\bowtie_t^k(t))\ \texttt{else}\ \bowtie_t^k(t) \end{array}$$

Figure 4.38: Extending traces to include declassification events.

$\theta^n(L)$	$=$	$\{l^n \mid l \in L\}$
$\Omega((L, \sqsubseteq), ((L^n, \sqsubseteq^n), n), l_{old}, l_{new})$	$=$	$((\theta^{n+1}(L) \cup L^n, \hat{\Omega}(\sqsubseteq, \sqsubseteq^n, l_{old}, l_{new}, n)), n+1)$
$\omega((L, \sqsubseteq), [])$	$=$	$((\theta^0(L), \sqsubseteq), 0)$
$\omega(\mathcal{L}, (\texttt{OUT}(n, l)::t))$	$=$	$\omega(\mathcal{L}, t)$
$\omega(\mathcal{L}, (\texttt{DECL}(l_{old}, l_{new})::t))$	$=$	$\Omega(\mathcal{L}, \omega(\mathcal{L}, t), l_{old}, l_{new})$
$\Theta_t([], n)$	$=$	$[]$
$\Theta_t((\texttt{OUT}(n, l)::t), n)$	$=$	$\Theta_t(t, n)$
$\Theta_t((\texttt{DECL}(l_{old}, l_{new})::t), n)$	$=$	$(\texttt{DECL}(l^n_{old}, l^n_{new})::\Theta_t(t, n-1))$

where $\hat{\Omega}(\sqsubseteq, \sqsubseteq^n, l_{old}, l_{new}, n)$ is defined as the *transitive closure* of $\sqsubseteq^{n+1}$, which is defined by

- $l_1^{m_1} \sqsubseteq^{n+1} l_2^{m_2} \iff l_1^{m_1} \sqsubseteq^n l_2^{m_2}$ for any $m_1, m_2 \leq n$
- $l_1^{n+1} \sqsubseteq^{n+1} l_2^{n+1} \iff l_1 \sqsubseteq l_2$
- $l^n \sqsubseteq^{n+1} l^{n+1}$
- $l^n_{old} \sqsubseteq^{n+1} l^{n+1}_{new}$

Figure 4.39: Generation of versioned lattices.

Note that for a versioned system model, the attacker label should be k^n, where n is the number of declassification operations in the trace and k is the original attacker label.

4.4.5 The Versioned-Labels Noninterference Theorem and Proof

At a high level, the versioned-labels noninterference property requires that we first look at all the declassification events in the system trace and generate the versioned lattice based on this trace. The second step is to use this versioned lattice to define the versioned model (i.e., $\check{\mathcal{M}}_d(\mathcal{L}^n)$). In the versioned model, the initial system instance with version zero labels can takes the exact same steps as in the original model. The next step is to use the projection function, with the last version of the attacker label, to obtain the projected system. The projection will compare last version of the attacker label with version zero of the instances labels. Therefore, any app instance in the initial system with possible information flow, through declassification or otherwise, to the attacker at the end of the trace will be kept in the in the projected system. Finally, the projected system is required to simulate the original system by producing a trace that is equivalent in the low events to the trace produced by the original system.

This definition means that if a high instance declassifies to low, it will be included in the projected system. Moreover, if the same instance raises it label after the declassification back to high, it will remain in the projected system but will no longer be required to simulate, i.e., produce the same events as or be equivalent at each step to, the corresponding instance in the original system. As an example, consider if this instance that just became high, call it A, receives a call from a newly created high instance in the original system, say B. The projected system does not include B, and thus won't be able to simulate the this call. This is perfectly fine, since A is now high and does not need to match the corresponding instance in the projected system. Next, A in the original system tries to call a low instance, say C, sending the value A received from B. The call will be blocked, and projected system does not need to simulate it (since A is now high). However, if due to faulty enforcement A's call to C is allowed. The projected system will not be able to simulate this call, since its copy of A does not have the value received from B.

We formally define the versioned-labels noninterference property in Definition 4. The definition essentially states that for if a well-configured system instance S takes several steps becoming S' and producing trace t with n declassification operations, then it must be the case that 1) S relabeled to version zero, $\Theta^0(S)$, can take the exact same steps to the S' relabeled to version n, $\Theta^n(S')$ and produce the relabeled version of the trace t; 2) the projection of $\Theta^0(S)$ using version n of the attacker label, $\bowtie_{sys}^{k^n}(\Theta^0(S))$, can take similar steps producing a trace $\check{t}$; and 3) $\check{t}$ is equivalent to the projection of the relabeled trace $\Theta_t(t, n-1)$. This analogous to the basic noninterference theorem, except that simulation is in the versioned model of the system.

Definition 4 (Versioned-Labels Noninterference)**.** *A model $\mathcal{M}_d(-)$ satisfies noninterfrence if, for any security lattice $\mathcal{L} = (L, \sqsubseteq)$, any attacker label $k \in L$, any system instances $S, S' \in \mathcal{M}_d(\mathcal{L})$, and any trace t:*

If $\models_{sys} S$ and $S \overset{t}{\Longrightarrow}* \; S'$ then exists $\check{S} \in \check{\mathcal{M}}_d(\omega(\mathcal{L}, t))$ and $\check{t}$ s.t.,

1. $\Theta^0(S) \xRightarrow{\Theta_t(t,n\text{-}1)}{}^{*}\ \Theta^n(S')$
2. $\bowtie_{sys}^{k^n}(\Theta^0(S)) \xRightarrow{\check{t}}{}^{*}\ \check{S}$, *and*
3. $\bowtie_t^{K^n}(\Theta_t(t, n-1)) \equiv \check{t}$.

Before we prove our versioned-labels noninterference theorem, we need some lemmas. First, Lemma 5 states that the ordering relation of all versioned lattices is transitive. Lemma 6 states that a label must be high if it is above a high label. Lemma 7 states that executing the versioned system is equivalent to executing the original system, and that we can get the final versioned system instance by simply versioning the original final instance. Finally, Lemma 8 states that for any declassification operation in the execution of the original system, a corresponding declassification operation will appear in the trace of the versioned system and that the declassified and new labels are either both low or both high in the final versioned lattice.

Lemma 5 (Transitivity of versioned ordering relation)**.** *For any lattice $\mathcal{L}$ and any trace t, if $((L^n, \sqsubseteq^n), n) = \omega(\mathcal{L}, t)$, then $\sqsubseteq^n$ is transitive.*

Proof. The proof proceeds by induction on the number of declassification operations in t and using the definition of $\hat{\Omega}$. □

Lemma 6. *For any lattice $(L, \sqsubseteq)$ and any labels $k, l_1, l_2 \in L$, if $l_1 \sqsubseteq l_2$ and $l_1 \not\sqsubseteq k$ then $l_2 \not\sqsubseteq k$*

Proof. Assume that $l_2 \sqsubseteq k$, then by transitivity we get that $l_1 \sqsubseteq k$, which contradicts our premise. Therefore it must be the case that $l_2 \not\sqsubseteq k$ □

Lemma 7. *For any lattice $\mathcal{L}$, system instances $S, S' \in \mathcal{M}_d(\mathcal{L})$ and any trace t, if $S \xRightarrow{t}{}^{*}\ S'$ and $((L^n, \sqsubseteq^n), n) = \omega(\mathcal{L}, t)$ then $\Theta^0(S) \xRightarrow{\Theta_t(t,n\text{-}1)}{}^{*}\ \Theta^n(S')$.*

Proof. The proof proceeds by induction on the number of declassification operations in t. We need to only consider declassification steps since the only difference between S and the relabeled system is the versioned labels, which have no effect on other operations. Thus, let us assume that t has only declassification events. The base case, where there is no declassification operation, is trivial since there are no steps to be taken. In the inductive case, we are adding one more declassification step, say $\texttt{DECL}(l_{old}, l_{new})$ to the trace. The only difference between S' and $\Theta(n)S'$ is the versioned labels. From the definition of ω, we know that $l^n_{old} \sqsubseteq^n l^n_{new} \iff l_{old} \sqsubseteq l_{new}$. Therefore, we can execute the similar $\texttt{DECL}(l^n_{old}, l^n_{new})$ step in $\Theta^n(S')$. In addition, it is trivial to prove that $\Theta(\Theta^n(S')) = \Theta^{n+1}(S')$ and $\Theta_t((\texttt{DECL}(l_{old}, l_{new})::t), n) = (\texttt{DECL}(l^n_{old}, l^n_{new})::\Theta_t(t, n-1))$. □

Lemma 8. *For any lattice $\mathcal{L}$ and any trace t, let $((L^n, \sqsubseteq^n), n) = \omega(\mathcal{L}, t)$ and $\check{t} = \Theta_t(t, n-1)$ then if $\texttt{DECL}(l_{old}, l_{new}) \in t$ then $\texttt{DECL}(l^m_{old}, l^m_{new}) \in \check{t}$, for some $m < n$, and Either*

- *$l^m_{old} \sqsubseteq^n k^n$ and $l^m_{new} \sqsubseteq^n k^n$ (both are low) , or*
- *$l^m_{old} \not\sqsubseteq^n k^n$ and $l^m_{new} \not\sqsubseteq^n k^n$ (both are high)*

Proof. By induction on t. The base case, $t = []$, is trivial. The inductive case has two subcases. The first subcase, when we add an output event, follows easily from the definition of ω and Θ_t. The second subcase is when we add a declassification event $\texttt{DECL}(l'_{old}, l'_{new})$. If $m < n+1$ then we can use the inductive hypotheses plus the fact that $\sqsubseteq n \subset \sqsubseteq n+1$.

Otherwise, if $m = n$, then we can easily show that $\texttt{DECL}({l'_{old}}^n, {l'_{new}}^n) \in \Theta^{(\texttt{DECL}(l'_{old}, l'_{new})::t)}(n)$ from the definition of Θ_t. In addition, from the definition of ω, we know that (1) $l^n_{old} \sqsubseteq^{n+1} l^{n+1}_{new}$ and (2) $l^n_{new} \sqsubseteq^{n+1} l^{n+1}_{new}$. There two cases here:

Case: (3a) $l^{n+1}_{new} \sqsubseteq^{n+1} k^{n+1}$

- $l^{n+1}_{new} \sqsubseteq^{n+1} K^{n+1}$, By transitivity from (3a) and (1)
- $l^{n+1}_{new} \sqsubseteq^{n+1} K^{n+1}$, By transitivity from (3a) and (2)

Case: (3b) $l^{n+1}_{new} \not\sqsubseteq^{n+1} k^{n+1}]$

- $l^{n+1}_{new} \not\sqsubseteq^{n+1} K^{n+1}$, By Lemma 6 from (3b) and (1)
- $l^{n+1}_{new} \not\sqsubseteq^{n+1} K^{n+1}$, By Lemma 6 from (3b) and (2)

□

Now, we are ready for the main theorem.

Theorem 9 (Versioned-Labels Noninterference)**.** *The model $\mathcal{M}_d(-)$ satisfies versioned-labels noninterference.*

Proof. (1) follows directly from Lemma 7. The rest of the proof will follow the proof of Theorem 4, except for the case of executing **DECLASSIFY**, which was not defined. The proof for **DECLASSIFY** will closely follow the proof for the **RAISE** case with the following considerations. If the issuing app instance is low ,i.e., attacker-accessible, then this is similar to a raise from low to low, i.e., both are a change from a low label to another low label. If, on the other hand, the issuing app instance is high, i.e., not attacker-accessible or $l_{old} \not\sqsubseteq^n k^n$, we have three subcases:

1. If the declassification is blocked, there is no need for simulation, the projected system can simply take no steps.
2. If the declassification is to a high label, the projected system can simply take no steps as the whole operation will be projected out.
3. If the declassification is to a low label, i.e., $l^m_{new} \sqsubseteq^n k^n$, then, from Lemma 8, it must be the case that the old label is low too, i.e., $l^m_{old} \sqsubseteq^n k^n$. But this contradicts our assumption, and thus this subcase is not valid.

□

4.4.6 Tracking the call chains

Another way to make the definition more precise is to observe that not all apps with the declassified label communicate with each other. Some apps do not affect or send information, directly or indirectly, to the declassified app. Thus, we do not need to declassify the information from those apps. Instead we can track all the call chains throughout the execution of the system, up to the point of declassification, and only declassify the information from those apps within call chains that end with the declassifying app.

However, we think that this enhancement is not needed. IFC requires that the apps are isolated by the system, and that the only communication channel between apps is monitored, i.e. the inter-app calling API. In android, isolation is provided by the underlying Linux kernel. We conjecture that the isolation property along with a noninterference property is enough to guarantee that only apps in the call chain could have contributed information to the declassifying app.

4.5 Related Work

In Sec. 2.4 we discussed works on Android security and IFC. In this section we will focus works related to formal security definitions for IFC systems.

Pioneered by Denning [5] as a principled form of controlling the flow of information among processes, (lattice-based) IFC has been widely studied in the information security field. IFC has been applied to sequential and parallel programs [6, 76]. While these earlier systems were understood to be secure there was no formal definition of their security. Formal definitions of security appeared later, and generally fall into two categories: noninterfernce and knowledge-based (or epistemic).

4.5.1 Noninterference Definitions of IFC Security

Goguen and Meseguer introduced the notion of noninterference to define the guarantees of an IFC scheme [7]. Several enhanced definitions, which account for declassification operations, have been proposed by researchers since then. One of the first such definitions is *intransitive noninterference*, where information are only allowed to flow from one security level to another if it's explicitly allowed by the policy [12]. Thus, a declassification operation D, say from A to B, can be described by allowing flows from A to D and from D to B. Chong and Myers developed a generalization of noninterference that accommodates declassification and erasing of information [15].

Because Flume has no floating labels, a stronger noninterference can be proved for it than can be proved for our system: Flume's definition of noninterference is based on a stable failure model, which is a simulation-based definition. Our definition is trace-based, and does not capture information leaks due to a high process stalling.

A rich body of work has focused on noninterference in process calculi [68, 69]. Recently, researchers have re-examined definitions of noninterference for reactive systems [77, 78]. In these systems, each component waits in a loop to process input and produce one or more outputs (inputs to other components). These works propose new definitions of noninterference based on the (possibly infinite) streams produced by the system. Our definition of noninterference is weaker, since we only consider finite prefixes of traces. These reactive models are similar to ours, but do not consider shared state between components, and assume the inputs and outputs are the only way to communicate, which is not the case for Android. Further, to model the component-based Android architecture more faithfully, we extend pi-calculus with a label context construct, which also enables formal analysis of our enforcement mechanism in the presence of Android components' ability to change their labels at run time. To our knowledge, such dynamic behavior has rarely been dealt with in the context of process calculus.

4.5.2 Knowledge-Based Definitions of IFC Security

Beside noninterference, researchers proposed *knowledge-based* properties to define the security of IFC systems [14, 79, 80, 81, 82]. These proposals construct a model of the attacker's knowledge and define the security property based on the change of this knowledge in response to enforcement events. For example, a declassification event can result in an increase of the attacker knowledge. The first such definition was introduced by Askarov and Sabelfeld for their work on *gradual release*. They modeled the attacker knowledge based on what the attacker can observe of the initial memory contents [14]. There are two main difficulties in applying a similar condition to our model. First, we need a way to represent the attacker knowledge. One way to do this is to consider the initial memory of all instances in the system. However, this does not account for user inputs or the inherit nondeterminism in our model, both of which can produce values (or knowledge) not entirely derived from the initial system state. Askarov and Sabelfeld also introduced an extension to gradual release that models user input channels as streams of values [83]. In addition, recent work by McCall et al. proposes another solution for representing user inputs in a knowledge-based definition [84]. Second, we still need a way to further restrict knowledge increase after a raise operation that otherwise would have been permissible. While these difficulties dissuaded us from further pursing this direction, we do think that a knowledge-based security definition could be a viable alternative to our noninterference approach. Indeed, if we assume that the attacker knowledge is defined by the set of app instances that the attacker could have received information from, then we would be able to define a security condition similar to gradual release [14]. In our case, the attacker knowledge can only increase upon a declassification event (which corresponds to a new label version), where we need to add the declassified instance and any instance that it have received calls from, directly or indirectly, to the attacker's knowledge set. We believe it should be possible to show that the versioned-labels noninterference implies this definition of security. Knowledge-based definitions are more intuitive in terms of understanding their implications on the security of the system because they relate directly to what an attacker can know. On the other hand, noninterference-based definitions of security are usually easier to prove for a system model because they are defined directly in terms of the model semantics.

Another knowledge-based definition was introduced in the work of Banerjee et al. [16], which we discuss more in the context of controlled declassification in Sec. 5.6. The security condition in their work, called *conditioned gradual release*, is based on gradual release but adapted to include conditional declassification. A more recent work by Askarov and Chong introduces a knowledge-based security condition for a language-based model that allows dynamic update of policies [82]. They consider both static and run-time enforcement of their policies. However, similar to gradual release, they only consider initial inputs for modeling the attacker's knowledge.

Flow Locks, an IFC policy specification language introduced by Broberg and Sands, was initially developed with a bisimulation based security property [85], but a knowledge-based formulation was developed for it in a later work [80]. Paralocks is a successor language of Flow Locks that adds RBAC (Role-based access control) inspired features [81]. In both Flow Locks and Paralocks, information flow policies are specified based on *actors*. An actor can be, for example, a specific security level in a conventional IFC policy or a principal in an RBAC-like policy. The information flow to actors is guarded by *locks*, which can be opened and closed. The attacker knowledge in both works is specified as the set of all possible initial memories given the attacker-observable outputs. As specifically stated in the Flow Locks paper, these two approaches are strictly static and cannot handle systems with run-time enforcement, like ours, where actual permissions (or labels) are only known at run time [85]. However, for the sake of comparison, let us assume that there is version of our model where all labels and inputs, and the execution schedule, are deterministic. We should be able to encode this restricted model using a language like the one in Flow Locks and Paralocks. In such an encoding, declassification would be implemented as one or more open-lock operations and a raise operation will be based on one more close-lock operations. However, even then, we suspect that the security condition, *flow lock security*, used to define the security of the Flow Lock language, would not be sufficient to imply our versioned-label noninterference definition. The reason is that the apps in our model, or even instances of the same app, can open or close the same lock at the same time. We could have all the code between the declassify and raise operations be atomic, but such a solution could be expensive both in term of performance and the complexity of modeling and verification (given the added synchronization constructs).

Chapter 5

Controlled Declassification

This chapter explores our approach for controlled declassification. The goal is to increase the expressiveness of our security policies to provide for more useful scenarios while retaining our proved security properties. This goal supports our thesis statement that IFC mechanisms with controlled declassification can provide more expressive and provable security guarantees for Android. In the previous chapters, the declassification requests are always granted as long as the app has the required capability. However, we want to enable more expressive policies by generalizing this check to be more than a simple lookup. Ideally, we want to be able to implement traditional access control mechanisms to control declassification. In order to support such implementations, we need the declassification decision to not only be based the app's declassification capabilities, but also on instance-specific state, app-specific state, global state, and app-specific policy specification. Researchers have proposed many approaches to control declassification [11, 13, 16, 17]. The closest proposal to meet our requirements is *stateful declassification* by Vanhoef et al. [17]. Stateful declassification controls declassification via an *information release* function that takes a state along with a high input value and outputs the next state and the value, if any, to be released to a low observer. However, stateful declassification does not explicitly support global state, app-specific state, or app-specific policies that are separate from the information release function implementation. Global state, which is accessible to both high and low apps, needs careful treatment to prevent it from being used, maliciously or accidentally, to violate the IFC policy. The first example in Sec. 5.4 illustrates the need for either global or local state based on the required policy.

We will start this chapter with some motivating examples in Sec. 5.1. Sec. 5.2 describes our approach in more details. Then we augment our model to implement controlled declassification in Sec. 5.3. After that, we give some example instantiations of the controlled declassification model in Sec. 5.4. In Sec. 5.5, we discuss an example of security properties that take advantage of controlled declassification and present a general proof for similar properties. We conclude this chapter with a discussion of related work in Sec. 5.6.

5.1 Motivating Scenarios

The following examples are scenarios where regular information flow control, with uncontrolled declassification, is not sufficient or adequate to implement the desired policy. In each scenario, the desired policy is informally stated, and then the possibility of enforcement using regular IFC systems is discussed. The purpose of listing the scenarios below is to show that there are policies that cannot be implemented easily, if at all, with regular IFC schemes. We also want to show that some proposed extensions to IFC declassification are either too specific, like robust declassification [86] and delimited release ([75]), or too general, like trusted declassification [87] (see Sec. 5.6).

Scenario 1: NSA analyst

> **Policy:** Files labeled as "classified" can only be declassified by an analyst with the role "senior"

This is a classic example of a role-based access control (RBAC) policy. There is no straightforward way to implement such policies as regular IFC (with declassification) policies. One complex way is to add labels for each role and only assign a role label to an object if the object is accessible by the role. However, associating roles with principals or manage role hierarchies can be a challenge. We would need separate mechanisms to manage these relations. Also, there is no obvious way to consider "principals" that are not apps or processes. A language like Jif [1], with explicit principals and an "acts for" mechanism, might help with this specific issue.

On the other hand, implementing such policy using trusted declassifiers would not be

[1] `http://www.cs.cornell.edu/jif/`

different from using a stand alone RBAC implementation. This similarity is because we will need a whole separate infrastructure for the roles hierarchy (and maybe permissions and their associations to roles) and for policy specifications, which cannot be embedded in the IFC labels without considerable complexity.

RBAC has been formalized and studied extensively [88, 89, 90]. Thus, instead of trying to impractically specify the policy using IFC labels, we can use regular RBAC policy specifications to implement the RBAC parts of the policy. These RBAC parts of the policy will be enforced when declassifying the relevant labels ("the classified" label in this case). In addition, we can use a logic-based formalization, such as [91], to more easily reason about the properties that can be enforced using RBAC.

Scenario 2

> **Policy:** Certain contents are secret and can only be declassified if they are declassified by the user and a specified application. The user approval (i.e., declassification) and the declassification by the application need to be within a specified short period of each other.

This policy could apply to scenarios where decisions are based on changing or otherwise time-sensitive inputs. Consider this scenario: Pictures taken using the eye-glasses mounted camera, which is connected to the phone, are to remain in the phone. A social media application can automatically select some of these pictures as candidates for posting online. The application will only declassify these pictures after checking that they don't infringe on other people's privacy using an AI algorithm that is frequently updated. In addition, the user must approve the posting of any of these pictures online. Because of the frequent updates to the AI algorithm, the checked pictures need to be rechecked if they are not approved by the user within 10 minutes.

A regular IFC scheme cannot be used to implement such a policy. However, one could tweak the labels to have the ability to attach timestamps to them. Both the declassification mechanisms and the policy specification language, will also need to be modified so that certain labels can only be declassified if their timestamp is recent enough. This way we can have two secrecy tags for each user, S1, D1[t], S2, D2[t], such that the first user can only declassify S1 to D1[t] and only declassify D2[t] if recent enough. A similar restriction applies to the second user. Moreover, we need to have the ability to define different IFC policies for different users of the system.

We think that a better approach is to integrate an access control scheme with timing constraints support (such as [92, 93, 94]). There will only be one label S (secret), and the IFC will be user-neutral. When an app asks for S to be declassified, an authorization is generated on behalf of the current user. The declassification will fail unless the other user's authorization is also present, and it has been generated within the specified time window.

Scenario 3

> **Policy:** A certain set of documents are to remain secret (i.e., within the phone). However, any of these documents can be declassified if the user declassifies it through application A, followed by declassifying it using application B (in that order).

For example, this scenario manifests in a setting where highly sensitive documents can only be declassified after the document has been sanitized by an application (A). The sanitized document can then be reviewed by the user, using another application (B), and it will only be declassified if the user approves.

Such a policy can be implemented with multiple labels, say S-PS-NS (secret, partial-secret, non-secret), so that the first user is only allowed to declassify from H to M, and the second user is allowed to declassify from M to L. This might not work in systems where it's not possible to specify such declassification abilities—most systems cannot, including Jif/JFlow, Flume, and ours [8, 18, 95]. We can implement such policy with trusted declassification ([87]) by only applying the declassification of the second user if the first user already declassified its label (some extra bookkeeping will be required though to keep of track who declassified which label). This logic and extra bookkeeping will be part of the policy, which will make policy verification more difficult.

The same approach, using access control schemes with time-constraint support, mentioned in Scenario 2 can be used here. This approach dictates that the declassification will fail unless the current user is user B and user A's authorization has been previously presented for the same file.

Scenario 4

> **Policy:** Certain modifications must be endorsed by at least n applications from the a set of m, where $n < m$, for them to apply.

An example of this is the approval of nontrivial payments in a corporate setting, where the approval of two executives is required (the so called "four eyes principle" or "two-man rule"). A similar requirement exists for nonprofit organizations who mandates two signatures on every check (e-signatures in our case, through a mobile app).[2] Another example includes situations where we don't want to trust only one viewer to accurately display a sensitive document to the user. Consequently, we require the document to be declassified using at least two separate viewers (e.g., from different vendors) from a list of available viewers before releasing online.

Similar to scenario 3, we might be able to craft special label lattices and give each application

[2] `https://www.councilofnonprofits.org/tools-resources/internal-controls-nonprofits`

the ability to only declassify certain labels to satisfy the policy. However, this not supported in current IFC schemes, and extending them as such will potentially increase the complexity of enforcement, policy specification, and verification. In addition, such policy specification can be unreadable, unless it's abstracted by higher level specification.

We believe a more suitable approach is to utilize, along with IFC, access control schemes that support *threshold-functions*, such as SecPAL and DKAL [96], to easily describe such constraints. Access control schemes that build on *threshold cryptography*, such as the work by Alsulaiman et al. [97] can also be used here.

Scenario 5

> **Policy:** An app developer wants only a certain set of pre-approved applications to access the information the app generates, and only a certain number of accesses. Each app has a different cap.

An example could be any social media app, like Facebook or Twitter, which already have similar restrictions on their online APIs [3] [4].

This policy cannot be enforced immediately by regular IFC systems. It could, however, be enforced through some tweaks to the implementation of the reference monitor to count the number of accesses and additions to the policy specification language. Similarly, trusted declassification [87] can be used to implement such policies, but would need some additional constructs included to identify sending (or receiving) apps and keep track of the number of accesses; and all that will be part of the policy. Instead, we can use access control schemes that limit the number of times credentials can be used (e.g., [98]).

[3] `dev.twitter.com/rest/public/rate-limiting`

[4] `developers.facebook.com/docs/graph-api/advanced/rate-limiting`

5.2 Approach

We propose the following approach to extend IFC. A declassification operation would only be successful if it's allowed by the IFC policy *and* an access control policy; otherwise it's disallowed.[5] The access control policy can be either 1) a global policy specified by the system administrator, 2) a distributed policy where each declassification capability will include an app-specific piece of policy specified by the app developer or the system administrator, or 3) a combination of both approaches. The effective policy for the whole system will be the combination of the IFC policy, represented by the security labels, and the access control policy.

We insert an additional check into all declassification operations. This check is implemented by a function that implement the access control scheme. This function inputs will be the access-control policy, the current security lattice, the current and final labels, and some *state*. The output will be a boolean decision and the updated state. There could different access control schemes, for different labels. However, we will restrict our discussion to one access control scheme in this work.

The state to which we give the check function access can be global, app-specific, instant-specific, or a combination of the above. For a general framework, we include all three kinds of states. For both global and app-specific state, we need to be careful of potential violations of IFC policy. Therefore, access to these state for each declassification request must be based on the requesting instance's dynamic label. There are two possible solutions for this issue. We could assign the specific label on these states. For example, the global state can be assigned an empty secrecy and integrity label, i.e., $(\{\},\{\})$, and app-specific state could be assigned the app's static label. Another approach that works better for the global state would be to partition it into multiple states, one for each possible label.[6] Then, whenever a declassification request is made, the access control function will only have write access to the partition with the exact label as the

[5]One could imagine an implementation where the declassification occurs if it's allowed by the access control policy *or* the IFC policy. However, we will not consider such implementation, as we suspect it will weaken the security properties that can be proved.

[6]An efficient implementation would allocate these partitions on demand; however, we will keep it simple here.

requesting instance and read access to the partitions with same label or any label below it in the lattice. We could allow write access to partitions with labels that are above the instance's in the lattice, but we doubt that it will be useful without read access.

Many extensions to regular IFC have been proposed [13, 16, 17, 75, 86, 87]. These proposals extend the expressiveness of the enforceable policies. The policies in many of the scenarios discussed above can be implemented by one or more of these extensions. However, some of these extensions are usually limited to a specific set of scenarios. Some closely related works do provide general conditions for declassification [16, 17], but lack explicit support for both global and local state as we do. In addition, we believe that the specification languages, if specified, of such extensions are not as practical as those for regular access control schemes. On the other hand, we explicitly separate policy and enforcement of the declassification conditions to support regular access control schemes.

For example, the policies in a couple of the scenarios above can be enforced using *quantitative information flow control* [99]. However, quantitative IFC can be expensive, performance-wise, to implement (see [100]). Another extension is the notion of trusted declassification discussed before [87]. While potentially arbitrary logic can be included in those trusted declassifiers, this approach is too flexible to aid in verification and will increase the size of the policy, which is almost always not desirable (see Sec. 5.6 for more details).

We believe that our approach can offer a good compromise between the extremely flexible trusted declassification approach and the limited schemes, such as *delimited release* [75], that provide useful and provable security properties for specific scenarios. However we do not claim that our approach is fundamentally better than the other IFC extensions, but instead it is another alternative to be considered.

5.3 A Model with Controlled Declassification $\mathcal{M}_{cd}(\mathcal{L}, f_{\mathrm{AC}}, \Sigma_{pol})$

In this section we extend our model to include the controlled declassification checks and state. First we redefine parts of the syntax and semantics in Secs. 5.3.1 and 5.3.2, respectively. Then, in Sec. 5.3.3, we will amend the versioned-labels noninterference proof to accommodate these changes.

Note that the model is parameterized by the language for the declassification policy labels Σ_{pol} and the access control function f_{AC}, which we will use to define the syntax and semantics, respectively, of the model.

5.3.1 Syntax

For the syntax changes, we need to add the states and the declassification policy labels. We use a key-value store, i.e., dictionary, data structure for the instance and app states. As for the global state, we need a data structure that can be indexed with labels, where each entry is a *Key-value store*. Figure 5.1 summarizes the notation and valid operations for both data structures. The required syntax changes are listed in Fig. 5.2. For generality, we do not specify the syntax for the declassification policy labels.

5.3.2 Semantics

The system's operational semantic are amended by adding the access control checks and making sure that the state stores are updated, whether the check succeeds or fails. We modify the system semantics, specifically those for declassification, in Figs. 5.3 and 5.4. The rules in Fig. 5.4 are updated to make sure that the global store changes are propagated.

The most important addition to the semantics is the access control function f_{AC} that is used to control declassification (Fig. 5.3). It takes as inputs the state stores, the declassification capability

Key-Value Store

$\square$	The empty key-value store
$\text{addKV}(st, k, v)$	Add a the key-value pair (k, v) to st
$\text{updKV}(st, k, v)$	Update the entry indexed by k in st with the value v
$\text{rmKV}(st, k)$	Remove the entry indexed by k from st
$\text{getKV}(st, k)$	Retrieves the value indexed by k in st

Global Store

$\text{filterGS}(gst, l)$	Remove all entries in st with label l' where $l' \not\sqsubseteq l$
$\text{updGS}(gst, l, st)$	Update the entry indexed by l in gst with the store st
	A new entry will be created if necessary
$\text{getGS}(gst, l)$	Retrieves the key-value store indexed by l in gst
	Returns $\square$ if the entry does not exist

Figure 5.1: Summary of the notation and operations for key-value and global stores. All store modifying operations will return the modified copy of the store.

$$
\begin{array}{lcl}
\mathit{Policy} & ::= & \langle \textit{any policy p, s.t., } p \in \Sigma_{\mathit{pol}} \rangle \\
\mathit{DeclCap} & ::= & \texttt{list}(\mathit{label}, \mathit{label}, \mathit{Policy}) \\
\mathit{KVS} & ::= & \langle \textit{Key-value store data structure} \rangle \\
\mathit{DynamicLabelList} & ::= & \texttt{list}(\mathit{IID}, \mathit{label}, \mathit{KVS}) \\
\mathit{AppLabelMap} & ::= & \text{SingleInstanceUnlaunched } \mathit{label}\ \mathit{IID}\ \mathit{DeclCap}\ \mathit{KVS} \\
 & | & \text{SingleInstanceLaunched } \mathit{label}\ \mathit{label}\ \mathit{IID}\ \mathit{DeclCap}\ \mathit{KVS}\ \mathit{KVS} \\
 & | & \text{MultiInstance } \mathit{label}\ \mathit{DynamicLabelList}\ \mathit{DeclCap}\ \mathit{KVS} \mid \text{NA} \\
\mathit{GKVS} & ::= & \langle \textit{Global store data structure} \rangle \\
\mathit{Sys} & ::= & (\mathit{Apps}, \mathit{IS}, \mathit{LM}, \mathit{PQ}, \mathit{GKVS})
\end{array}
$$

Figure 5.2: Amending system syntax to enable controlled declassification.

that includes the local policy, the global policy, the system model security lattice, the current label, and the new label. The lattice is part of the model, but making it an explicit argument to the access control function will help with the proof in Sec. 5.5.1. The output of the function is the updated stores and a boolean value to indicate if the declassification should be allowed or not. Notice that the state stores are updated even if the declassification was denied, which is fine because these stores are accessible from the current instance label anyway. The instance store, st_{i1}, is passed as is to the access control function, and the updated store, st_{i2} is saved to the label map. The app store, st_a, is passed to the access control function only if the app's label is below the instance's, and updated only if the labels agree. The access control function will only get the entries in the global store, gst_1, with labels that are at or below the instance's. The access control function output, st, replaces the global entry with the same label as the instance's. Also, the operator $\in^*$ in these rules works like $\in$, except that it will ignore the policy label, i.e., $(l_1, l_2) \in^* \delta \iff \exists pol \cdot (l_1, l_2, pol) \in \delta$. Note that the reset of the operational semantics will need trivial modification to account for those changes. These modifications are mainly to account for the extra fields in the apps' label map entries, which will be just ignored.

5.3.3 The Versioned-Labels Noninterference Proof

We need to ensure that the modifications made to the system model to add controlled declassification did not compromise the security guarantees. We do this by adapting the noninterference theorem to the new model. We first need to modify the projection function to account for the global store. The rest of the projection function definition from Figs. 4.30 and 4.31 stays the same, ignoring the addition of local stores. The projection function simply removes all entries with high labels from the global store (see Fig. 5.5). This works because the global store returns an empty store for nonexistent labels and allows update operations to create entries if necessary.

Theorem 10 (Versioned-Labels Noninterference)**.** *The model $\mathcal{M}_{cd}(-, f_{\text{AC}}, \Sigma_{pol})$ satisfies*

$$\frac{\begin{array}{c} aid@lm_1\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl\ \delta\ st_{a1}\ st_{i1}) \\ l \not\sqsubseteq dl \qquad (dl,l) \in^* \delta \qquad st_{a,in} = \texttt{if}\ sl \sqsubseteq dl\ \texttt{then}\ st_{a1}\ \texttt{else}\ \Box \\ f_{\text{AC}}(\text{filterGS}(gst_1, dl), st_{a,in}, st_{i1}, \delta, \mathcal{L}, dl, l) = (st, st_{a,out}, st_{i2}, \texttt{true}) \\ st_{a2} = \texttt{if}\ sl = dl\ \texttt{then}\ st_{a,out}\ \texttt{else}\ st_{a1} \\ \texttt{Update}\ aid@lm_1\ \texttt{with}\ (\text{SingleInstanceLaunched}\ iid\ sl\ l\ \delta\ st_{a2}\ st_{i2}) \mapsto lm_2 \\ gst_2 = \text{updGS}(gst_1, dl, st) \end{array}}{(lm_1, gst_1) \underset{\textbf{DECLASSIFY}}{\overset{(dl,l)}{\rhd}} \langle aid, iid, l\rangle\ (lm_2, gst_2)}$$

$$\frac{\begin{array}{c} aid@lm_1\ \texttt{HasAppLabel}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl\ \delta\ st_{a1}\ st_{i1}) \\ st_{a,in} = \texttt{if}\ sl \sqsubseteq dl\ \texttt{then}\ st_{a1}\ \texttt{else}\ \Box \\ f_{\text{AC}}(\text{filterGS}(gst_1, dl), st_{a,in}, st_{i1}, \delta, \mathcal{L}, dl, l) = (st, st_{a,out}, st_{i2}, b) \\ l \sqsubseteq dl \vee (dl,l) \notin^* \delta \vee b \qquad st_{a2} = \texttt{if}\ sl = dl\ \texttt{then}\ st_{a,out}\ \texttt{else}\ st_{a1} \\ \texttt{Update}\ aid@lm_1\ \texttt{with}\ (\text{SingleInstanceLaunched}\ iid\ sl\ dl\ \delta\ st_{a2}\ st_{i2}) \mapsto lm_2 \\ gst_2 = \text{updGS}(gst_1, dl, st) \end{array}}{(lm_1, gst_1) \underset{\textbf{DECLASSIFY}}{\overset{\tau}{\rhd}} \langle aid, iid, l\rangle\ (lm_2, gst_2)}$$

$$\frac{\begin{array}{c} aid@lm_1\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll_1\ \delta\ st_{a1}) \\ iid@dll_1\ \texttt{HasInstanceLabel}\ (dl, st_{i1}) \\ l \not\sqsubseteq dl \qquad (dl,l) \in^* \delta \qquad st_{a,in} = \texttt{if}\ sl \sqsubseteq dl\ \texttt{then}\ st_{a1}\ \texttt{else}\ \Box \\ f_{\text{AC}}(\text{filterGS}(gst_1, dl), st_{a,in}, st_{i1}, \delta, \mathcal{L}, dl, l) = (st, st_{a,out}, st_{i2}, \texttt{true}) \\ st_{a2} = \texttt{if}\ sl = dl\ \texttt{then}\ st_{a,out}\ \texttt{else}\ st_{a1} \qquad \texttt{Update}\ iid@dll_1\ \texttt{with}\ (l, st_{i2}) \mapsto dll_2 \\ \texttt{Update}\ aid@lm_1\ \texttt{with}\ (\text{MultiInstance}\ sl\ dll_2\ \delta\ st_{a2}) \mapsto lm_2 \\ gst_2 = \text{updGS}(gst_1, dl, st) \end{array}}{(lm_1, gst_1) \underset{\textbf{DECLASSIFY}}{\overset{(dl,l)}{\rhd}} \langle aid, iid, l\rangle\ (lm_2, gst_2)}$$

$$\frac{\begin{array}{c} aid@lm_1\ \texttt{HasAppLabel}\ (\text{MultiInstance}\ sl\ dll_1\ \delta\ st_{a1}) \\ iid@dll_1\ \texttt{HasInstanceLabel}\ (dl, st_{i1}) \qquad st_{a,in} = \texttt{if}\ sl \sqsubseteq dl\ \texttt{then}\ st_{a1}\ \texttt{else}\ \Box \\ f_{\text{AC}}(\text{filterGS}(gst_1, dl), st_{a,in}, st_{i1}, \delta, \mathcal{L}, dl, l) = (st, st_{a,out}, st_{i2}, b) \qquad l \sqsubseteq dl \vee (dl,l) \notin^* \delta \vee b \\ st_{a2} = \texttt{if}\ sl = dl\ \texttt{then}\ st_{a,out}\ \texttt{else}\ st_{a1} \qquad \texttt{Update}\ iid@dll_1\ \texttt{with}\ (dl, st_{i2}) \mapsto dll_2 \\ \texttt{Update}\ aid@lm_1\ \texttt{with}\ (\text{MultiInstance}\ sl\ dll_2\ \delta\ st_{a2}) \mapsto lm_2 \\ gst_2 = \text{updGS}(gst_1, dl, st) \end{array}}{(lm_1, gst_1) \underset{\textbf{DECLASSIFY}}{\overset{\tau}{\rhd}} \langle aid, iid, l\rangle\ (lm_2, gst_2)}$$

Figure 5.3: Amending the operational semantics for **DECLASSIFY** to add access control checks.

$$\frac{(lm_1, is_1, pq_1, gst_1)/apps \overset{\texttt{DECL}(l_{old}, l_{new})}{\rhd} (lm_2, is_2, pq_2, gst_2)}{(apps, lm_1, is_1, pq_1, gst_1) \overset{\texttt{DECL}(l_{old}, l_{new})}{\Longrightarrow} (apps, lm_2, is_2, pq_2, gst_2)}$$

$$\frac{(aid, iid)@is_1 \succ_{is} \langle \textbf{DECLASSIFY}\ l, is_2, ra\rangle \qquad (lm_1, gst_1) \underset{\textbf{DECLASSIFY}}{\overset{(dl,l)}{\rhd}} \langle aid, iid, l\rangle\ (lm_2, gst_2)}{(lm_1, is_1, pq, gst_1)/apps \overset{\texttt{DECL}(dl,l)}{\rhd} (lm_2, is_2, pq, gst_2)}$$

$$\frac{(aid, iid)@is_1 \succ_{is} \langle \textbf{DECLASSIFY}\ l, is_2, ra\rangle \qquad (lm_1, gst_1) \underset{\textbf{DECLASSIFY}}{\overset{\tau}{\rhd}} \langle aid, iid, l\rangle\ (lm_2, gst_2)}{(lm_1, is_1, pq, gst_1)/apps \overset{\tau}{\rhd} (lm_2, is_2, pq, gst_2)}$$

Figure 5.4: Amending system semantics to add controlled declassification state stores.

$$\begin{array}{ll}
\bowtie^k_{gst}(gst) & = \text{filterGS}(gst, k) \\
\hline\hline
\bowtie^k_{sys}(apps, lm, is, pq, gst) & = (apps, \bowtie^k_{lm}(lm), \bowtie^k_{is}\langle lm\rangle(lm, is), \bowtie^k_{pq}\langle lm, lm\rangle(pq), \bowtie^k_{gst}(gst))
\end{array}$$

Figure 5.5: The projection function modified to account for global stores.

versioned-labels noninterference.

Proof. In order for the versioned-labels noninterference proof to stay valid, we only need to verify that all the declassification steps can be simulated by the projected system. Other system steps won't be affected. From the proof of Theorem 9, we know that in the versioned system simulation, all successful declassification operations will either declassify from high to high or low to low. In the first case, the projected system does not need to take any steps, as all the changes are projected out. In the second case, we know that the low parts of the global store will be the same between the projected and original system. The same applies to the app's local store if the static label was low. If it was high, the store won't be used because the dynamic label, which is low, cannot be higher or equal to the static label. Therefore, all the inputs and outputs to the access control functions will be the same. Thus, the projected system will be able to simulate the same operation.

As for blocked declassification operations, the requesting instance can be either low or high. The same reasoning from the unblocked case above applies here too. □

5.4 Examples

In this section we introduce some example instantiations of the model $\mathcal{M}_{cd}(\mathcal{L}, f_{\text{AC}}, \Sigma_{pol})$.

5.4.1 Declassification Count

We start with a simple example policy that requires each instance to use certain declassification capabilities only a specific number of times. Here, the policy syntax for the declassification labels is defined as an integer, i.e., $\Sigma_{pol} ::= N$. The access control function will first check if there is a counter, say $count$, in the instance's store. If $count$ does not exist, the function will add one with an initial value of zero. Then it will only allow the declassification if $count \leq limit$, where limit is the integer from the policy. It will finally increment $count$ by one. If we need to enforce the policy for different declassification capabilities, we would use a uniquely named counter for each capability. This scheme can be used to implement Scenario 5 from Sec. 5.1.

If we require the limit to be global instead of a per app instance, then we need to use the global store. For a declassification capability from l_{old} to l_{new}, we would store the counter in the entry of l_{old} in the global store. This will work because regardless of the ultimate decision, the counter will always be accessible when declassifying from l_{old}. Such policy cannot be implemented, in a provably secure way, in systems which do not support declassification conditions based on a global state.

5.4.2 Declassification Chains

This policy requires an instance to only be able to declassify its label along a certain chain (i.e., sequence) of labels in the lattice. For example, if the chain is l_1, l_2, l_3, then the app instance will need to declassify its label from l_1 to l_2 before it's able to declassify from l_2 to l_3. The policy syntax is a list of labels from the lattice, i.e., $\Sigma_{pol} ::= \texttt{list}\, label$. The access control function will maintain a list, or history, of approved declassification in the instance store. For each declassification request, it will check whether the history matches the policy chain.

This scheme can be used to implement Scenario 3 from Sec. 5.1 by giving the individual declassification capabilities to the respective apps, and specifying a chain of length three as the declassification label policy for the app that must declassify last.

5.5 Lattice-Based Properties

In this section we focus on a special type of properties for IFC systems that are based on the versioned lattice. The goal is to give an example of properties that can be provably enforced using our controlled declassification system.

The versioned lattice, in a way, encodes the effects of declassification on the original lattice. Thus, we can use it to describe certain properties about the overall system behavior with regard to declassification. For example, by looking at the versioned lattice we can determine the number of declassification operations between specific labels. These properties can be captured with a predicate that takes a versioned lattice as an input, i.e., $\mathcal{P}(\mathcal{L}^n)$. For example, the predicate $\mathcal{P}((L^n, \sqsubseteq^n)) = \exists m_1, m_2 \cdot l_1^{m_1} \sqsubseteq^n l_2^{m_2}$ captures the condition of whether the declassification operations in this specific execution allowed information flows from label l_1 to l_2, which otherwise could not have happened.

It is important to note that these lattice-based properties do not provide any guarantees, unless the system also satisfies noninterference. Otherwise, the policy that is specified by the lattice is not provably enforced, and thus we cannot conclude anything based on the lattice. An example of this lack of guarantee is if a lattice-based property says that there was no declassification from l_1 to l_2, where $l_1 \not\sqsubseteq l_2$. One would expect that any app with l_1 will not be able to send information to an app with l_2. However, this is not guaranteed if noninterference does not hold for the system.

5.5.1 General Proof Strategy for Lattice-Based Properties

In order to prove that a system instance, with a specific access control scheme, satisfies a lattice-based predicate we need to show that for all possible system executions the predicate will be true given the resulting versioned lattice. We define this notion concretely in Definition 5 below. Note that t could be empty or contain no declassification, which means the predicate must be valid for the lattice of the initial instance S, i.e., $\mathcal{P}(\mathcal{L}^0)$.

Definition 5. *We say that a system instance $S \in \mathcal{M}_{cd}(\mathcal{L}, f_{\text{AC}}, \Sigma_{pol})$ satisfies lattice-based predicate $\mathcal{P}$ if, for any trace t, and system instances $S' \in \mathcal{M}_{cd}(\mathcal{L}, f_{\text{AC}}, \Sigma_{pol})$, if $S \overset{t}{\Longrightarrow}* \; S'$ then $\mathcal{P}(\omega(\mathcal{L}, t))$.*

In general, to prove that a system instance satisfies a property, it is sufficient to show that it is true for $\mathcal{L}^0$, and that if it is true for $\mathcal{L}^n$ then it will be true for $\mathcal{L}^{n+1}$ given the access control function of the system model. Theorem 11 codifies this inductive proof strategy for lattice-based properties. Note that the choice of the predicate Φ depends on the specific property and access control scheme.

Theorem 11 (Lattice-Based Properties)**.** *For any system model $\mathcal{M}_{cd}(\mathcal{L}, f_{\text{AC}}, \Sigma_{pol})$ and any initial system instance $S \in \mathcal{M}_{cd}(\mathcal{L}, f_{\text{AC}}, \Sigma_{pol})$, S satsifies $\mathcal{P}$ if we can show that*

- $\mathcal{P}(\mathcal{L}^0)$
- *for any app instance in S, $\Phi(gst, st_a, st_i)$ is true, where gst is the global store of S, st_a and st_i are the app and local stores of this instance, and Φ is a predicate we chose over those stores*
- *for any trace t, system instance $S' \in \mathcal{M}_{cd}(\mathcal{L}, f_{\text{AC}}, \Sigma_{pol})$, s.t., $S \overset{t}{\Longrightarrow}* \; S'$, and any labels l_{old}, l_{new}, s.t., $l_{new} \not\sqsubseteq l_{old}$, if*
  - $\mathcal{P}(\mathcal{L}^n)$*, and*
  - $\Phi(gst_1, st_{a,in}, st_{i1})$*, and*
  - $f_{\text{AC}}(\text{filterGS}(gst_1, l_{old}), st_{a,in}, st_{i1}, \delta, \mathcal{L}, l_{old}, l_{new}) = (st, st_{a,out}, st_{i2}, \texttt{true})$

then

- $\mathcal{P}(\mathcal{L}^{n+1})$, *and*
- $\Phi(\text{updGS}(gst_1, l_{old}, st), st_{a,out}, st_{i2})$

where $\mathcal{L}^n = \omega(\mathcal{L}, t)$ *and* $\mathcal{L}^{n+1} = \omega(\mathcal{L}, (\texttt{DECL}(l_{old}, l_{new})::t))$.

Proof. The proof is a straightforward induction proof once we note that only declassification operations change the state stores, and if successful, advance the lattice version. One key observation we need is that

$$f_{\text{AC}}(\text{filterGS}(gst_1, l_{old}), st_{a,in}, st_{i1}, \delta, \mathcal{L}, l_{old}, l_{new}) = (st, st_{a,out}, st_{i2}, \texttt{true})$$

if and only if

$$f_{\text{AC}}(\text{filterGS}(gst_1, l^n_{old}), st_{a,in}, st_{i1}, \delta, \mathcal{L}^{n+1}, l^n_{old}, l^n_{new}) = (st, st_{a,out}, st_{i2}, \texttt{true})$$

This is because $l_1^n \sqsubseteq^{n+1} l_2^n \iff l_1 \sqsubseteq l_2$. The same applies for denied declassifications. $\square$

Theorem 11 allows us to easily prove that a lattice-based predicate is satisfied by some system instance. In essence, we only need to show that the access control function will maintain the predicate after a successful declassification operation.

5.5.2 The Lattice-Based Chinese Wall Policy

Here we present our adaption of the Chinese Wall policy [101], slightly modified for our setting. Given two labels, the lattice-based Chinese wall policy states that there should never be an information flow between any two app instances with those two labels. This policy should hold regardless of the declassification operation in the system. There could be more than two labels; however, for simplicity, we will focus on the case of two labels. If two labels satisfy this policy, we say that they are *separated*.

In order for the policy to make sense, the lattice should not allow information flows between the two labels to begin with. Such labels are called *incomparable*. Two labels, l_1 and l_2, are incomparable if $l_1 \not\sqsubseteq l_2$ and $l_2 \not\sqsubseteq l_1$. In other words, information should not flow from l_1 to l_2 or from l_2 to l_1.

In order to facilitate our discussion, we need a way to describe what labels can send information to and from a specific label. We define the operators $\uparrow$ and $\downarrow$, which take a label and calculate the set of all labels that are below it and above it, respectively, in the lattice. We call these two sets the *up set* and *down set* of the label.

$$\begin{aligned} (L, \sqsubseteq) \uparrow l &= \{l' : l' \in L \wedge l \sqsubseteq l'\} \\ (L, \sqsubseteq) \downarrow l &= \{l' : l' \in L \wedge l' \sqsubseteq l\} \end{aligned}$$

Next, we formally define when two labels are separated. Definition 6 makes sure that regardless of the label version, there must not be an information flow to any version of the separated label. A system satisfies the lattice-based chinese wall policy for two labels if the labels are separated in all versioned lattices that can possibly result from the system execution.

Definition 6 (Separated Labels). *The two incomparable labels l_1 and l_2 are separated in a versioned lattice $\mathcal{L}^n$ if for any $m_1, m_2 \leq n$,*

- $\mathcal{L}^n \uparrow l_1^{m_1} \cap \mathcal{L}^n \downarrow l_2^{m_2} = \{\}$*, and*
- $\mathcal{L}^n \downarrow l_1^{m_1} \cap \mathcal{L}^n \uparrow l_2^{m_2} = \{\}$

There are multiple ways we can enforce this policy depending on how flexible we want to be. In the extreme, we can disallow all declassification requests. However, we would want to be more flexible. Label separation requires that the up and down sets of the labels do not intersect. This implies that, ideally, we should allow any declassification that does not change these sets or change them without making them intersect. The fact that these sets can dynamically grow, similar to how new users are dynamically assigned to a class in the original Chinese Wall model, is why we called this policy "lattice-based Chinese Wall". However, when we tried to implement this enforcement in our model, we discovered that it's not possible due to the restriction on global state access. We found that implementing such enforcement would result in IFC policy violation. An attacker who controls an app with the capability to declassify between two high labels can control the outcome of a declassification between two low labels. This scenario occurs when there is a path between one of the low labels and one of the high ones, such that allowing both declassifications leads to a violation of the separation policy. Below, we introduce a more conservative approach to enforcing label separation. The fact that we were unable to implement this enforcement mechanism, which violates the IFC policy, in our model, shows the importance of having proven secure models.

One way to implement label separation enforcement in our model is to allow all declassifications, unless they are declassifying to one of the two labels down set, i.e., if $l_{new} \in \mathcal{L} \downarrow l_i^m$ for any m. This strategy will ensure that new information flows to these labels are never introduced. This is true because (1) the down set for the label will never change in this case as we do not allow any new labels to be below it, and (2) the up set might acquire new labels, but never ones from the down sets. Thus, the intersection from the separation property will always be the empty set. One can visualize this enforcement as a "wall" surrounding the down sets and preventing any new flow or "arrow" added to the lattice from crossing inside. The up sets, which do not have such restrictions, are allowed to "grow". In fact, we could also enforce label separation by having the "wall" around the two up sets (allowing the down sets to grow) or around the down and up set of one of the labels (allowing the sets of the other label to grow).

We implement the enforcement above with the following scheme. We don't need the policy labels, thus Σ_{pol} can be set to any value since it will be ignored. The system will be initialized, possibly by some privileged app, such that the unversioned down sets of labels l_1 and l_2 are listed in the global store with the no secrets label, making them readable by any app. The access control function will only allow declassification requests if l_{new}, ignoring label version, is not in any of those down sets.

The proof strategy from Sec. 5.5.1 can be used here to prove that the label separation property will always hold. The invariant predicate Φ will not depend on the lattice version and will only require that the unversioned down sets are in the global store. Thus, Φ will always be true. The inductive step follows directly from the discussion above. That is the down sets will not change[7], and the up sets (which need to be recalculated after the declassification) will never include a label from the down sets. Therefore, the predicate from Definition 6 will always be true.

5.6 Related Work

Researchers have proposed several approaches to control declassification. Hicks et al. introduced the notion of *trusted declassification*, where declassification can only happen through a set of trusted functions that are part of the policy [87]. The problem with this approach is that policies are not declarative but specify the enforcement mechanism itself, which complicates both verification and policy management. The policy size will increase because one need to include not just declarative policy requirements, but the actual enforcement mechanism in the policy. For example, a declarative piece of policy could be "declassify once," while an enforcement mechanism would need to maintain some state and check the state each time a declassification is requested.

Another approach is *robust declassification*, which ensures that declassification operations cannot be controlled by the attacker [86, 102]. *Delimited release* esures that any variable that is later declassified is not controlled by the attacker [75]. Both of these works, while representing important security properties, do not allow for the specification of general conditions on declassification.

Banerjee et al. proposed a policy framework that allows the specification of logical conditions for declassification [16]. The conditions in their work can look at past events, which approximate

[7]New versions of the same label will be added to those down sets. However, we are ignoring label versions for the purpose of our proof.

state. Stateful declassification by Vanhoef et al. allows conditions in the form of *information release* functions that take a state variable as an input, which is similar to our instance state store, but they don't explicitly support global state with label-guarded paritions [17]. Our work and both of these works share the same goal of increasing the expressiveness of policies through conditional declassification and the use of some kind of state. In our work we focus on Android and explicitly allow conditions to be based on label-guarded global state, which enable even more expressive policies (see Sec. 5.4.1).

5.6.1 Logic-Based Access Control

Logic-based access control grants access based on formal proofs. These proofs are based on both a set of credentials and a policy written in formal language. A large number of logic-based access control schemes have been proposed in the last 20 years. The earliest work on designing a logic-based access control policy specification language was done by Abadi et al. [103]. Their work was the foundation for many of the later works in this area. SD3 was the first of a series of logic-based languages that are based on Datalog. Other notable works in this series are Binder [104], Cassandra [105], and SecPal [106]. Binder supported distributed policies and is more expressive than SD3 [104]. Cassandra was based on Datalog with constraints and it allowed for adjusting the expressiveness (and performance) by leaving the choice of the constraint domain to the implementation and by only requiring that the domain be *constraint compact* [105]. SecPal's syntax is closer to the natural language and admits more constraint domains by only requiring that the constraint be ground (i.e., no variables) at run-time [106]. DKAL builds on SecPal; however, instead of using Datalog, it uses *existential fixed-point logic* (EFPL) for its formalism, which supports even more expressive policies. DKAL also supports *distributed knowledge*: each assertion is qualified with the principal who knows (or is the recipient of) the fact in question [96]. While each newer scheme provided more expressiveness, none of these schemes, as far as we know, enabled expressing properties that are similar to IFC's noninterference in terms of controlling information throughout a closed system with multiple apps running together.

Chapter 6

Conclusion

We conclude this dissertation by summarizing the contributions and discussing potential future works.

6.1 Summary

We began this dissertation with the thesis that *IFC polices and enforcement mechanisms can provide expressive, practically useful, and provable security guarantees for Android.* To support this thesis we used an IFC approach to enhance the security the Android OS, as an example of modern application platforms.

We started by presenting our approach of using run-time enforcement of IFC policies in Android. In this approach we proposed a tag-based policy specification scheme, where each tag represents a certain confidential piece of information, such as GPS location data, or a certain permission to access a sensitive resource, such as Internet or SMS access. These tags are assigned to apps and app components, and their assignments form the security policy that we must enforce. A run-time monitor enforces the policy by intercepting all communications between apps, or app components, and only allowing the communication to proceed if it does not, potentially, lead to policy violation, such as leaking of confidential information or access to a sensitive resource. In addition, we support declassification capabilities and floating labels, which provide for expressive and flexible policies. We demonstrated, via example, that this approach can provide security for reasonable scenarios that are unsecure in the regular Android OS. Furthermore, in a parallel work, it has been show that implementing this scheme in Android is feasible and has no user discernible effects on the system performance [95].

The approach above relies on the fact that apps are isolated by the Android OS, and only allowed to communicate through system APIs. However, within an app, app components are not completely isolated and can communicate directly. Thus, component-level policy enforcement is not robust, and relies on developers choosing to strictly use Android APIs for component to component communication. To bridge this gap, we developed a static analysis too that can detect potential direct, i.e., unmonitored, communications between components. We then used this tool to study the feasibility of robust component-level enforcement for more than 2,500 apps from the Google Play Store. Our findings suggest that for the majority of existing apps, retrofitting component-level enforcement is unfeasible. However, the tool can still be used by developers to avoid direct communication in their apps, especially when using third-party code.

Next, and based on our believe that formal verification is essential to provide assurance in any security mechanism or protocol, we formally proved that our enforcement mechanism will correctly enforce the desired security policy. We first built a faithful π-calculus model of the Android OS and our enforcement mechanism and prove that it satisfies the noninterference property. The noninterference property guarantees, in the absence of declassification, that security policy will never be violated. Second, we developed a simplified, labeled-transition-system (LTS), model of our system and used the Coq proof assistant to prove that it satisfies noninterference. We then introduced a novel versioned-labels noninterference property definition that provide security guarantees even when declassification and raise operations are used, and proved that the simplified model satisfies this property.

Finally, we studied the addition of traditional access-control schemes to control the declassification operations within the IFC scheme. We present several useful scenarios whose desired policy can be practically enforced with access-controlled declassification, but not with basic IFC approach. We then augmented our simplified Android model to add access-control checks and policy specification for declassification. Our model explicitly provide global, app, and local states for implementing these access-control schemes, which provide for quite expressive policies. We also proved that the augmented model still satisfies the versioned-labels noninterference definition for any declassification access-control scheme. Finally, we present lattice-based properties as an example of a class of properties that relies on controlled declassification can be proved, along with noninterference, for our augmented model.

We disscus possible future work next.

6.2 Future Work

One of our immediate future works is to complete the model from Sec. 4.2 to include the missing features from the more complete π-calculus model. We also would like to have the whole versioned-labels noninterference proof in Coq. Another future work is to convert the Coq model into executable code and try to run it with some sample apps and policies that approximate what is typically in Android. This testing will increase the confidence in the accuracy of this Android model. Finishing these few pieces of work will increase the assurance in our models and proofs.

We also would like to apply our approach to other platforms, such as web browsers and the Internet-of-Things (IoT). Web browsers, while different from mobile OSs, share similar characteristics. For example, both platforms run code that is sourced from different vendors; though the level of isolation is somewhat weaker in web browsers. The IoT platform also has apps from different vendors, but these apps usually run widely different software stacks on separate devices, which makes policy specification and enforcement quite an interesting challenge.

In another direction, we would like to explore fine-grained policy enforcement at levels other than components. For example, it could be possible to enforce policies at the Java class level or at the level of Java packages. It should be interesting to have insight into the comparative advantages and disadvantages of each approach in terms of practicality, ease of policy enforcement, impact on performance, and complexity of verification.

Finally, we want to study the instantiation of the declassification policy in the model from Ch. 5 with different established access-control policy specification schemes. For example, we might want to instantiate the model with DKAL or SecPAL as the policy scheme for declassification [96, 106]. In particular, we would like to discover the limitations of our model in terms of allowing for the different access-control schemes. Maybe it need some extra features or tweaks to support some of the schemes. In addition, we are interested in what kind of properties that can be proved about the enforcement of such concrete policy schemes in our model. For example, can we prove that our enforcment satisfies *non-occlusion* for all schemes, some of the schemes, or none [107].

Bibliography

[1] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proceedings of the 18th Network and Distributed System Security Symposium*, ser. NDSS, 2011.

[2] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC, 2011.

[3] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proceedings of the 13th International Conference on Information Security*, ser. ISC, 2010.

[4] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC, 2011.

[5] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, 1976.

[6] G. R. Andrews and R. P. Reitman, "An axiomatic approach to information flow in programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1980.

[7] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the 1928 IEEE Symposium on Security and Privacy*, 1982.

[8] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP, 2007.

[9] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2010.

[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys, 2011.

[11] K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson, "Checking interaction-based declassification policies for Android using symbolic execution," in *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security*, 2015.

[12] J. Rushby, “Noninterference, transitivity, and channel-control security policies,” SRI International, Computer Science Laboratory, Tech. Rep. CSL-92-02, 1992.

[13] P. Li and S. Zdancewic, “Downgrading policies and relaxed noninterference,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL, 2005.

[14] A. Askarov and A. Sabelfeld, “Gradual release: Unifying declassification, encryption and key release policies,” in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, ser. S&P, 2007.

[15] S. Chong and A. C. Myers, “Language-based information erasure,” in *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, ser. CSFW, 2005.

[16] A. Banerjee, D. A. Naumann, and S. Rosenberg, “Expressive declassification policies and modular static enforcement,” in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. S&P, 2008.

[17] M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk, “Stateful declassification policies for event-driven programs,” in *Proceedings of the IEEE 27th Computer Security Foundations Symposium*, ser. CSF, 2014.

[18] A. C. Myers, “Jflow: Practical mostly-static information flow control,” in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL, 1999.

[19] D. Hedin and A. Sabelfeld, “Information-flow security for a core of JavaScript,” in *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, ser. CSF, 2012.

[20] D. Zhang, A. Askarov, and A. C. Myers, “Language-based control and mitigation of timing channels,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2012.

[21] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, “Android permissions: User attention, comprehension, and behavior,” in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS, 2012.

[22] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, 2003.

[23] A. Chudnov and D. A. Naumann, “Information flow monitor inlining,” in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, ser. CSF, 2010.

[24] A. Russo and A. Sabelfeld, “Dynamic vs. static flow-sensitive security analysis,” in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, ser. CSF, 2010.

[25] S. Moore and S. Chong, “Static analysis for efficient hybrid information-flow control,” in *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, ser. CSF, 2011.

[26] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, “Sharing mobile code securely with information flow control,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. S&P, 2012.

[27] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL, 2012.

[28] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2008.

[29] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP, 2009.

[30] W. Cheng, D. R. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov, "Abstractions for usable information flow control in Aeolus," in *Proceedings of the 2012 USENIX Annual Technical Conference*, ser. USENIX ATC, 2012.

[31] M. Krohn and E. Tromer, "Noninterference for a practical DIFC-based operating system," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, ser. S&P, 2009.

[32] W. Enck, M. Ongtang, and P. D. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security*, ser. CCS, 2009.

[33] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS, 2011.

[34] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," Department of Computer Science, University of Maryland, College Park, Tech. Rep. CS-TR-4991, 2009.

[35] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST, 2012.

[36] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS, 2010.

[37] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel, "Semantically rich application-centric security in Android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC, 2009.

[38] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for Android applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. SEC, 2012.

[39] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "ipShield: A framework for enforcing context-aware privacy," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2014.

[40] A. Chaudhuri, "Language-based security on Android," in *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, ser. PLAS, 2009.

[41] 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 *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS, 2011.

[42] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC, 2011.

[43] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *19th Annual Network and Distributed System Security Symposium*, ser. NDSS, 2012.

[44] P. Loscocco and S. Smalley, "Meeting critical security objectives with security-enhanced linux," in *Proceedings of the 2001 Ottawa Linux symposium*, 2001.

[45] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC, 2013.

[46] Y. Xu and E. Witchel, "Maxoid: Transparently confining mobile applications with custom views of state," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys, 2015.

[47] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC enforcement on Android," in *Proceedings of 25th USENIX Security Symposium*, ser. SEC, 2016.

[48] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A formal model to analyze the permission authorization and enforcement in the Android framework," in *Proceedings of the Second IEEE International Conference on Social Computing / the Second IEEE International Conference on Privacy, Security, Risk and Trust*, ser. SocialCom/PASSAT, 2010.

[49] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A small but non-negligible flaw in the Android permission scheme," in *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks*, ser. POLICY, 2010.

[50] E. Fragkaki, L. Bauer, and L. Jia, "Modeling and enhancing Android's permission system," in *Computer Security—ESORICS 2012: 17th European Symposium on Research in Computer Security*, 2012.

[51] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, W. Enck, and P. Traynor, "*droid: Assessment and evaluation of Android application analysis tools," *ACM Computing Surveys*, 2016.

[52] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX Conference*

on Security, ser. SEC, 2013.

[53] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to Android," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2012.

[54] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS, 2012.

[55] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC, 2012.

[56] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS, 2012.

[57] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE, 2014.

[58] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2013.

[59] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in DroidSafe," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, ser. NDSS, 2015.

[60] S. Calzavara, I. Grishchenko, and M. Maffei, "HornDroid: Practical and sound static analysis of Android applications by SMT solving," in *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, ser. Euro S&P, 2016.

[61] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2014.

[62] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in Android applications," in *MoST 2012: Mobile Security Technologies*, H. Chen, L. Koved, and D. S. Wallach, Eds., 2012.

[63] S. Liang, M. Might, and D. V. Horn, "Anadroid: Malware analysis of Android with user-supplied predicates," *Electronic Notes in Theoretical Computer Science*, 2015.

[64] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2014.

[65] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, 2009.

[66] M. Avalle, A. Pironti, and R. Sisto, “Formal verification of security protocol implementations: a survey,” *Formal Aspects of Computing*, 2014.

[67] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal verification of an os kernel,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP, 2009.

[68] R. Focardi and R. Gorrieri, “A classification of security properties for process algebras,” *Journal of Computer Security*, 1994.

[69] P. Y. A. Ryan and S. A. Schneider, “Process algebra and non-interference,” *Journal of Computer Security*, 2001.

[70] J. Aljuraidan, E. Fragkaki, L. Bauer, L. Jia, K. Fukushima, S. Kiyomoto, and Y. Miyake, “Run-time enforcement of information-flow properties on Android,” Carnegie Mellon University, Tech. Rep. CMU-CyLab-12-015, 2012.

[71] S. Moore, A. Askarov, and S. Chong, “Precise enforcement of progress-sensitive security,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS, 2012.

[72] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey, *Programming Language Foundations*, ser. Software Foundations series, volume 2. Electronic textbook, May 2018, version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[73] R. Milner and D. Sangiorgi, “Barbed bisimulation,” in *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, ser. ICALP, 1992.

[74] A. W. Roscoe and M. H. Goldsmith, “What is intransitive noninterference?” in *Proceedings of the 12th IEEE Workshop on Computer Security Foundations*, ser. CSFW, 1999.

[75] A. Sabelfeld and A. C. Myers, “A model for delimited information release,” in *Software Security-Theories and Systems*. Springer, 2004.

[76] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow,” *Communications of the ACM*, 1977.

[77] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, “Reactive noninterference,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS, 2009.

[78] W. Rafnsson and A. Sabelfeld, “Limiting information leakage in event-based communication,” in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, ser. PLAS, 2011.

[79] N. Broberg, B. van Delft, and D. Sands, “The anatomy and facets of dynamic policies,” *The Computing Research Repository*, 2015.

[80] N. Broberg and D. Sands, “Flow-sensitive semantics for dynamic information flow

policies," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS, 2009.

[81] ——, "Paralocks: Role-based information flow control and beyond," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL, 2010.

[82] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *Proceedings of the IEEE 25th Computer Security Foundations Symposium*, ser. CSF, 2012.

[83] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, ser. CSF, 2009.

[84] M. McCall, H. Zhang, and L. Jia, "Knowledge-based security of dynamic secrets for reactive programs," in *Proceedings of the IEEE 31st Computer Security Foundations Symposium*, ser. CSF, 2018.

[85] N. Broberg and D. Sands, "Flow locks: Towards a core calculus for dynamic flow policies," in *Programming Languages and Systems*, P. Sestoft, Ed.

[86] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification and qualified robustness," *Journal of Computer Security*, 2006.

[87] B. Hicks, D. King, P. McDaniel, and M. Hicks, "Trusted declassification: High-level policy for a security-typed language," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, ser. PLAS, 2006.

[88] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, 1996.

[89] M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr, "A first step towards formal verification of security policy properties for RBAC," in *Proceedings of the Quality Software, Fourth International Conference*, ser. QSIC, 2004.

[90] T. Mossakowski, M. Drouineaud, and K. Sohr, "A temporal-logic extension of role-based access control covering dynamic separation of duties," in *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic*, 2003.

[91] T. Kosiyatrakul, S. Older, and S.-K. Chin, "A modal logic for role-based access control," in *Proceedings of the Third International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, ser. MMM-ACNS, 2005.

[92] L. Bauer, M. A. Schneider, and E. W. Felten, "A general and flexible access-control system for the Web," in *Proceedings of the 11th USENIX Security Symposium*. San Francisco, CA: USENIX, Aug. 2002, pp. 93–108. [Online]. Available: http://www.ece.cmu.edu/~lbauer/papers/webauth-sec02.pdf

[93] L. Bauer, "Access control for the Web via proof-carrying authorization," Ph.D. dissertation, Princeton University, Nov. 2003. [Online]. Available: http://www.ece.cmu.edu/~lbauer/papers/thesis.pdf

[94] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati, "An access control model supporting periodicity constraints and temporal reasoning," *ACM Transactions on Database Systems*, 1998.

[95] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-time enforcement of information-flow properties on Android (extended abstract)," in *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*, 2013.

[96] Y. Gurevich and I. Neeman, "DKAL: Distributed-knowledge authorization language," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, ser. CSF, 2008.

[97] F. A. Alsulaiman, A. Miège, and A. El Saddik, "Threshold-based collaborative access control (t-cac)," in *Proceedings of the 2007 International Symposium on Collaborative Technologies and Systems*, ser. CTS, 2007.

[98] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter, "Consumable credentials in logic-based access-control systems," in *Proceedings of the 2007 Network and Distributed System Security Symposium*, ser. NDSS, 2007.

[99] G. Smith, "On the foundations of quantitative information flow," in *Foundations of software science and computational structures*. Springer, 2009.

[100] H. Yasuoka and T. Terauchi, "Quantitative information flow - verification hardness and possibilities," in *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, ser. CSF, 2010.

[101] D. F. C. Brewer and M. J. Nash, "The Chinese Wall security policy," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[102] S. Zdancewic and A. C. Myers, "Robust declassification," in *In Proceedings of the 14th IEEE Computer Security Foundations Workshop*, ser. CSFW, 2001.

[103] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1993.

[104] J. DeTreville, "Binder, a logic-based security language," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, ser. S&P, 2002.

[105] M. Y. Becker and P. Sewell, "Cassandra: distributed access control policies with tunable expressiveness," in *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, ser. POLICY, 2004.

[106] M. Y. Becker, C. Fournet, and A. D. Gordon, "Secpal: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, 2010.

[107] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Comput. Secur.*, vol. 17, no. 5, pp. 517–548, 2009.

Appendices

A Coq Model

lattice.v

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Require Import Coq.Program.Equality.
Require Import Coq.omega.Omega.

Module SecurityLattice.

Import ListNotations.

(* The set of labels for the security lattice *)
Inductive label : Set :=
  | H : label
  | L : label.


(*  The partial order relation between labels *)
Definition lte_bool (l1:label) (l2:label) : bool :=  match (l1,l2) with  | (H,L) =>
    false  | (_,_) => true end.
Definition lte (l1:label) (l2:label): Prop := lte_bool l1 l2 = true.
Definition nlte (l1:label) (l2:label): Prop := lte_bool l1 l2 = false.


End SecurityLattice.
```

labelmap.v

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Require Import Coq.Program.Equality.
Require Import Coq.omega.Omega.
Require Export lattice.

Module LabelMap.


Import ListNotations.
Import SecurityLattice.

(* App id *)
Definition aid : Type := nat.

Definition beq_aid := beq_nat.

(* Instance id *)
Inductive iid : Type :=
  | Iid : nat -> iid.

Definition beq_iid ii1 ii2 :=
  match ii1,ii2 with
    | Iid n1, Iid n2 => beq_nat n1 n2
  end.
```

```

Inductive app_label_map : Type :=
 | Single_Instance_App_Unlaunched (sl :label) (prev_ii : iid)
 | Single_Instance_App_Launched (sl : label) (dl : label) (ii: iid)
 | Multi_Instance_App (sl : label) (dl_list : list (iid * label))
 | NA.

Definition default_alm := Single_Instance_App_Unlaunched TOP (Iid 0).

(* The Label Map maps apps and instances into labels *)
Definition label_map : Type := list app_label_map.


(** Utility Functions *)
Fixpoint update_label_map_app (lm : label_map) (ai:aid) (rep_alm : app_label_map):
    label_map :=
  match lm with
  | nil => nil
  | alm :: tl => match ai with
                   | O => rep_alm :: tl
                   | S n => alm :: (update_label_map_app tl n rep_alm)
                   end
  end.


Fixpoint next_iid_nat (dll : list (iid * label)) : nat :=
match dll with
  | nil => O
  | (Iid n,_) :: tl => S (max n (next_iid_nat tl))
end.
Definition next_iid (dll : list (iid * label)) : iid := Iid (next_iid_nat dll).
Definition increment_iid (ii : iid) : iid := match ii with | Iid n => Iid (n+1) end.

Definition get_app_label_map (lm: label_map) (ai: aid) : option app_label_map :=
  nth_error lm ai.

Definition get_inst_label (dll : list (iid * label)) (ii:iid) : option label :=
option_map snd (find (fun a => beq_iid (fst a) ii) dll).

Definition get_label (alm: app_label_map) (ii: iid) : option label :=
  match alm with
  | Single_Instance_App_Launched sl dl ii2 => if beq_iid ii ii2 then Some dl else
    None
  | Multi_Instance_App sl dll => get_inst_label  dll ii
  | _ => None
  end.

(** Label Map Operations**)

Inductive has_app_label_map :
  label_map ->
  aid ->
  app_label_map -> Prop :=
  | halm_next lm ai alm alm2:
     has_app_label_map lm ai alm ->
     has_app_label_map ( alm2 :: lm ) (S ai) alm
  | halm_get lm alm:
    has_app_label_map ( alm :: lm ) O alm.

Inductive has_inst_label:
  list (iid * label) ->
  iid -> option label -> Prop :=
  | hil_next dll ii l ii2 l2:
    beq_iid ii ii2 = false ->
    has_inst_label dll ii l ->
    has_inst_label  ((ii2,l2) :: dll) ii l
  | hil_nil ii: has_inst_label nil ii None
  | hil_get dll ii l:
      has_inst_label ( (ii,l) :: dll) ii (Some l).

Inductive has_label:
  label_map ->
  aid -> iid ->
  option (label * label) -> Prop :=
  | hl_sl lm ai ii sl dl:
    has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
    has_label lm ai ii (Some (sl,dl))
  | hl_sl_none lm ai ii ii' sl dl:
    has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii') ->
    beq_iid ii ii' = false ->
    has_label lm ai ii None
  | hl_m lm ai ii sl dll dl:
    has_app_label_map lm ai (Multi_Instance_App sl dll) ->
    has_inst_label dll ii (Some dl) ->
    has_label lm ai ii (Some (sl,dl))
  | hl_m_none lm ai ii sl dll:
    has_app_label_map lm ai (Multi_Instance_App sl dll) ->
    has_inst_label dll ii None ->
    has_label lm ai ii None.


Inductive update_inst_label:
  list (iid * label) ->
  iid -> label ->
  list (iid * label) -> Prop :=
  | uil_next dll ii l ii2 l2 dll2:
    beq_iid ii ii2 = false ->
    update_inst_label dll ii l dll2 ->
    update_inst_label  ((ii2,l2) :: dll) ii l ((ii2,l2) :: dll2)
  | uil_update dll ii l l_old:
      update_inst_label ( (ii,l_old) :: dll) ii l ( (ii,l) :: dll).

Inductive update_app_label_map :
  label_map ->
  aid ->
  app_label_map ->
  label_map -> Prop :=
  | ualm_next lm lm2 ai alm alm2:
     update_app_label_map lm ai alm lm2 ->
     update_app_label_map ( alm2 :: lm ) (S ai) alm ( alm2 :: lm2)
  | ualm_get lm alm alm_old:
     update_app_label_map ( alm_old :: lm ) O alm (alm :: lm).


```

```
End LabelMap.
```

core.v

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Require Import Coq.Program.Equality.
Require Import Coq.omega.Omega.

Require Export labelmap.
Require Export lattice.

Module CoreLanguage.

Import ListNotations.

Import LabelMap.
Import SecurityLattice.

(* id for variables*)
Inductive id : Type :=
  | Id : nat -> id
  | ARG.

Hint Constructors id.

Definition beq_id id1 id2 :=
  match id1,id2 with
    | Id n1, Id n2 => beq_nat n1 n2
    | ARG, ARG => true
    | _,_ => false
  end.

Inductive tp : Type :=
  | Tbool
  | Tnat
  | Tref (t : tp).

Definition loc := nat.

Inductive val : Type :=
  | v_bool (b:bool)
  | v_nat (n: nat)
  | v_loc (l: loc).


Inductive exp : Type :=
  | e_var (x: id) : exp
  | e_val (v:val) : exp
  | e_eq : exp->exp->exp
  | e_deref : exp -> exp.

Definition e_bool (b: bool) : exp := e_val (v_bool b).
Definition e_nat (n: nat) : exp := e_val (v_nat n).
Definition e_loc (l: loc) : exp := e_val (v_loc l).

Definition mem : Type := list val.
Definition empty_mem : mem := nil.

Fixpoint update_mem (m:mem) (l:loc) (v:val) : mem :=
  match m with
  | nil => nil
  | h :: t => match l with
      | O => v :: t
      | S n => h :: (update_mem t n v)
      end
  end.

Definition lookup_mem (m:mem) (l:loc) : val :=
  nth l m (v_bool false).

Inductive value : exp -> Prop :=
  | Value v : value (e_val v).


Inductive cmd : Type :=
  | Raise (l:label)
  | Decl (l:label)
  | Call (ai : aid) (e : exp) (expect_return: bool)
  | Return (e : exp)
  | Exit (e : exp).

Inductive stm : Type :=
  | s_seq (s1 s2 : stm)
  | s_if (e:exp) (s1 s2:stm)
  | s_null
  | s_assign (e1 e2: exp)
  | s_let  (x:id) (e: exp) (s:stm)
  | s_new (x:id) (e: exp) (s:stm)
  | s_cmd (c : cmd).

Fixpoint subst_exp  (x:id) (v:val) (e:exp) : exp :=
  match e with
  | e_var y => if beq_id x y then (e_val v) else e
  | e_eq e1 e2 => e_eq (subst_exp x v e1) (subst_exp x v e2)
  | e_val v => e
  | e_deref e => e_deref (subst_exp x v e)
  end.

Definition subst_cmd (x:id) (v:val) (c:cmd) : cmd :=
  match c with
  | Call ai e er => Call ai (subst_exp x v e) er
  | Return e => Return (subst_exp x v e)
  | Exit e => Exit (subst_exp x v e)
  | Raise _ => c
  | Decl _ => c
  end.


Fixpoint subst_stm (x:id) (v:val) (s:stm) : stm :=
  match s with
  | s_seq s1 s2 => s_seq (subst_stm x v s1) (subst_stm x v s2)
  | s_if e s1 s2 => s_if (subst_exp x v e) (subst_stm x v s1) (subst_stm x v s2)
  | s_null => s_null
  | s_assign e1 e2 => s_assign (subst_exp x v e1) (subst_exp x v e2)
  | s_let y e s =>
    if beq_id x y then s_let x (subst_exp x v e)  s else s_let y (subst_exp x v e) (
   subst_stm x v s)
  | s_new y e s =>
    if beq_id x y then s_new x (subst_exp x v e)  s else s_new y (subst_exp x v e) (
   subst_stm x v s)
  | s_cmd c => s_cmd (subst_cmd x v c)
  end.

Inductive step_exp : mem -> exp -> exp -> Prop :=
  | se_eq1 m b1 b2 :  step_exp m (e_eq (e_bool b1) (e_bool b2) ) (e_bool (eqb b1 b2))
  | se_eq2 m n1 n2 :  step_exp m (e_eq (e_nat n1) (e_nat n2) ) (e_bool (beq_nat n1 n2
   ))
  | se_eq3 m l1 l2 :  step_exp m (e_eq (e_loc l1) (e_loc l2) ) (e_bool (beq_nat l1 l2
   ))
  | se_eq4 m v e2 e2' : step_exp m e2 e2' ->  step_exp m (e_eq (e_val v) e2 ) (e_eq (
   e_val v) e2' )
  | se_eq5 m  e1 e1' e2 : step_exp m e1 e1' ->  step_exp m (e_eq e1 e2 ) (e_eq e1' e2
    )
  | se_deref1 m e e': step_exp m e e' -> step_exp m (e_deref e) (e_deref e')
  | se_deref2 m l v : lookup_mem m l = v -> step_exp m (e_deref (e_loc l)) (e_val v).

Inductive step_cmd : mem -> cmd -> cmd -> Prop :=
  | sc_call m e e' ai er: step_exp m e e' -> step_cmd m (Call ai e er) (Call ai e' er
   )
  | sc_return m e e': step_exp m e e' -> step_cmd m (Return e) (Return e')
  | sc_exit m e e': step_exp m e e' -> step_cmd m (Exit e) (Exit e').

Inductive step_stm : mem -> stm -> mem -> stm -> Prop :=
  | ss_seq1 m s: step_stm m (s_seq s_null s) m s
  | ss_seq2 m m' s1 s1' s2: step_stm m s1 m' s1' -> step_stm m (s_seq s1 s2) m' (
   s_seq s1' s2)
  | ss_if1 s_true s_false m :
      step_stm m (s_if (e_bool true) s_true  s_false) m s_true
  | ss_if2 s_true s_false m :
      step_stm m (s_if (e_bool false) s_true  s_false) m s_false
  | ss_if3 s_true s_false m e e'  :
      step_exp m e e' -> step_stm m (s_if e s_true s_false)  m (s_if e' s_true
   s_false)
  | ss_assign1 l m v: step_stm m (s_assign (e_loc l) (e_val v)) (update_mem m l v)
   s_null
  | ss_assign2 m e e1 e': step_exp m e e' -> step_stm m (s_assign e e1) m (s_assign e
   ' e1)
  | ss_assign3 m l e e': step_exp m e e' -> step_stm m (s_assign (e_loc l) e) m (
   s_assign (e_loc l) e')
  | ss_let1 m x v s : step_stm m (s_let x (e_val v) s) m (subst_stm x v s)
  | ss_let2 m x e e' s : step_exp m e e' -> step_stm m (s_let x e s) m (s_let x e' s)
  | ss_new1 m x v s:  step_stm m ( s_new x (e_val v) s) (m ++ [v]) (subst_stm x (
   v_loc (length m)) s)
  | ss_new2 m x e e' s : step_exp m e e' -> step_stm m (s_new x e s) m (s_new x e' s)
  | ss_cmd m cm cm' : step_cmd m cm cm' -> step_stm m (s_cmd cm) m (s_cmd cm').

Definition cntx := id ->  option tp.
Definition empty_cntx : cntx := fun _ => None.

Definition update_cntx : cntx -> id -> tp -> cntx := fun c x t =>
```

```
  fun y => if beq_id x y then Some t else c y.

Definition default_cntx : cntx := update_cntx empty_cntx ARG Tnat.


Definition mem_tp := list tp.
Definition lookup_mem_tp (mt: mem_tp) (l: loc) : tp :=
  nth l mt Tbool.

Inductive extends : mem_tp -> mem_tp -> Prop :=
  | extends_nil mt' : extends mt' nil
  | extends_cons v mt' mt : extends mt' mt -> extends (v::mt') (v::mt).

Inductive wt_exp : cntx -> mem_tp -> exp -> tp -> Prop :=
  | wte_var c mt x t:  (c x) = Some t -> wt_exp c mt (e_var x) t
  | wte_bool c mt b: wt_exp c mt (e_bool b) Tbool
  | wte_nat c mt n: wt_exp c mt (e_nat n) Tnat
  | wte_loc c mt l: l < length mt -> wt_exp c mt (e_loc l) (Tref (lookup_mem_tp mt l)
    )
  | wte_eq c mt e1 e2 t: wt_exp c mt e1 t -> wt_exp c mt e2 t -> wt_exp c mt (e_eq e1
     e2) Tbool
  | wte_deref c mt e t: wt_exp c mt e (Tref t) -> wt_exp c mt (e_deref e) t.

Inductive wt_cmd : cntx -> mem_tp -> cmd -> Prop :=
  | wtc_raise c mt l: wt_cmd c mt (Raise l)
  | wtc_decl c mt l: wt_cmd c mt (Decl l)
  | wtc_call c mt ai e er: wt_exp c mt e Tnat -> wt_cmd c mt (Call ai e er)
  | wtc_return c mt e: wt_exp c mt e Tnat -> wt_cmd c mt (Return e)
  | wtc_exit c mt e: wt_exp c mt e Tnat -> wt_cmd c mt (Exit e).

Hint Constructors wt_cmd.

Inductive wt_stm : cntx -> mem_tp -> stm -> Prop :=
  | wts_seq c mt s1 s2 : wt_stm c mt s1 -> wt_stm c mt s2 -> wt_stm c mt (s_seq s1 s2
    )
  | wts_if c mt e s1 s2 : wt_exp c mt e Tbool -> wt_stm c mt s1 -> wt_stm c mt s2 ->
    wt_stm c mt (s_if e s1 s2)
  | wts_assign c mt e1 e2 t: wt_exp c mt e1 (Tref t) -> wt_exp c mt e2 t -> wt_stm c
    mt (s_assign e1 e2)
  | wts_let c mt x e s t: wt_exp c mt e t -> wt_stm (update_cntx c x t) mt s ->
    wt_stm c mt (s_let x e s)
  | wts_new c mt x e s t: wt_exp c mt e t -> wt_stm (update_cntx c x (Tref t)) mt s
    -> wt_stm c mt (s_new x e s)
  | wts_null c mt : wt_stm c mt s_null
  | wts_cmd c mt cm: wt_cmd c mt cm -> wt_stm c mt (s_cmd cm).

Definition wt_mem (m:mem) (mt:mem_tp) : Prop :=
  length m = length mt /\
  (forall (l:loc), wt_exp empty_cntx mt (e_val (lookup_mem m l)) (lookup_mem_tp mt l)
    ).

Inductive free_in_exp : id -> exp -> Prop :=
  | fie_var x : free_in_exp x (e_var x)
  | fie_eq1 x e1 e2 : free_in_exp x e1 -> free_in_exp x (e_eq e1 e2)
  | fie_eq2 x e1 e2 : free_in_exp x e2 -> free_in_exp x (e_eq e1 e2)
  | fie_deref x e : free_in_exp x e -> free_in_exp x (e_deref e).

```

```
Inductive free_in_cmd : id -> cmd -> Prop :=
  | fic_call x e ai er: free_in_exp x e -> free_in_cmd x (Call ai e er)
  | fic_return x e: free_in_exp x e -> free_in_cmd x (Return e)
  | fic_exit x e: free_in_exp x e -> free_in_cmd x (Exit e).

Inductive free_in_stm : id -> stm -> Prop :=
  | fis_seq1 x s1 s2 : free_in_stm x s1 -> free_in_stm x (s_seq s1 s2)
  | fis_seq2 x s1 s2 : free_in_stm x s2 -> free_in_stm x (s_seq s1 s2)
  | fis_ifc x s1 s2 e : free_in_exp x e -> free_in_stm x (s_if e s1 s2)
  | fis_if1 x s1 s2 e : free_in_stm x s1 -> free_in_stm x (s_if e s1 s2)
  | fis_if2 x s1 s2 e : free_in_stm x s2 -> free_in_stm x (s_if e s1 s2)
  | fis_assign1 x e1 e2 : free_in_exp x e1 -> free_in_stm x (s_assign e1 e2)
  | fis_assign2 x e1 e2 : free_in_exp x e2 -> free_in_stm x (s_assign e1 e2)
  | fis_lete x y e s :  free_in_exp x e -> free_in_stm x (s_let y e s)
  | fis_lets x y e s : x <> y -> free_in_stm x s -> free_in_stm x (s_let y e s)
  | fis_newe x y e s :  free_in_exp x e -> free_in_stm x (s_new y e s)
  | fis_news x y e s : x <> y -> free_in_stm x s -> free_in_stm x (s_new y e s)
  | fis_cmd x cm: free_in_cmd x cm -> free_in_stm x (s_cmd cm).

End CoreLanguage.
```

system.v

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Require Import Coq.Program.Equality.

Require Export core.
Require Export labelmap.
Require Export lattice.

Module System.

Import ListNotations.

Import CoreLanguage.
Import LabelMap.
Import SecurityLattice.

Definition ret_addr : Type := option (aid * iid).

Definition get_ret_addr (expect_ret : bool) (ai: aid) (ii: iid) : ret_addr :=
  if expect_ret then Some (ai, ii) else None.


(* An app is a statement *)
Definition app : Type := stm.

(* An app_set is a list of apps *)
Definition app_set : Type := list app.

(* instance is an app instance that consist of a the instance id, a memory
   a (running) copy of an app, and a return address *)
Inductive instance : Type :=
  | Inst (id: iid) (m: mem) (a: app) (ret: ret_addr).

```

```
(* An app_inst_set is a list of running instances of a particular app *)
Definition app_inst_set : Type := list instance.

(* An inst_set is a list of instances running in parallel *)
Definition inst_set : Type := list app_inst_set.

Inductive pendingType : Type :=
| pCall (ra : ret_addr)
| pReturn ( callee_ii : iid).

Inductive pending : Type :=
| Pend (callee : aid) (n : nat) (caller_dl : label) (pt : pendingType).

Definition pqueue : Type := list pending.

Inductive sys : Type :=
  | Sys ( apps: app_set ) (lm: label_map) (insts: inst_set) (queue : pqueue).



Fixpoint not_a_command (s : stm) : Prop :=
  match s with
   | s_cmd (Call _ (e_val _) _ ) | s_cmd (Return (e_val _))
     | s_cmd (Exit (e_val _)) | s_cmd (Raise _) | s_cmd (Decl _) =>
      False
   | s_seq s1 s2 => not_a_command s1
   | _ => True
  end.

(* retrieve original statement/code of an app *)
Definition app_stm (apps: list app) (ai : aid): stm :=
  nth ai apps s_null.


(* An instance in an instance list takes a step *)
Inductive app_is_step :
  app_inst_set ->
  app_inst_set -> Prop :=
  | aiss_next inst ais1 ais2 : app_is_step ais1 ais2 -> app_is_step (inst :: ais1) (
    inst :: ais2)
  | aiss_step ii m1 m2 s1 s2 ret ais:
     not_a_command s1 -> step_stm m1 s1 m2 s2 -> app_is_step ((Inst ii m1 s1 ret) ::
    ais) ((Inst ii m2 s2 ret) :: ais).

Inductive is_step :
  inst_set ->
  inst_set -> Prop :=
  | iss_next h is1 is2 : is_step is1 is2 -> is_step (h :: is1) (h :: is2)
  | iss_step ais1 ais2 is:  app_is_step ais1 ais2 -> is_step (ais1 :: is) (ais2 :: is
    ).

Definition is_ready (c : cmd) : Prop :=
  match c with
  | Raise _ | Decl _ | Call _ (e_val _) _
  | Return (e_val _) | Exit (e_val _ ) => True
  | _ => False
  end.
```

```

(* Decide wether a stament represent an active command
   and compute the residual command left after executing*)
Inductive is_command:
  stm ->
  cmd -> stm -> Prop :=
  | iscom_c c :
      is_ready c -> is_command (s_cmd c) c s_null
  | iscom_seq s c residual s2:
      is_command s c residual -> is_command (s_seq s s2) c (s_seq residual s2).

(* An instance in an instance set dispatches a command c *)
Inductive app_is_cmd :
  iid ->
  cmd ->
  app_inst_set ->
  app_inst_set -> ret_addr -> Prop :=
  | aisc_next ii c inst ra ais1 ais2 :
      app_is_cmd ii c ais1 ais2 ra ->
      app_is_cmd ii c (inst :: ais1) (inst :: ais2) ra
  | aisc_call ii s callee e er residual m ra ais:
      is_command s (Call callee e er) residual ->
      app_is_cmd ii (Call callee e er) ((Inst ii m s ra) :: ais) ((Inst ii m residual
     ra) :: ais) ra
  | aisc_raise ii s l residual m ra ais:
      is_command s (Raise l) residual ->
      app_is_cmd ii (Raise l) ((Inst ii m s ra) :: ais) ((Inst ii m residual ra) ::
    ais) None
  | aisc_decl ii s l residual m ra ais:
      is_command s (Decl l) residual ->
      app_is_cmd ii (Decl l) ((Inst ii m s ra) :: ais) ((Inst ii m residual ra) ::
    ais) None
  | aisc_return ii s e residual m ra ais:
      is_command s (Return e) residual ->
      app_is_cmd ii (Return e) ((Inst ii m s ra) :: ais) ((Inst ii m s_null ra) ::
    ais) None
  | aisc_exit ii s e residual m ra ais:
      is_command s (Exit e) residual ->
      app_is_cmd ii (Exit e) ((Inst ii m s ra) :: ais) ((Inst ii nil s_null ra) ::
    ais) None.

Inductive is_cmd:
 aid -> iid -> cmd -> inst_set -> inst_set -> ret_addr -> Prop :=
 | isc_next ai ii c is1 is2 ra ais:
     is_cmd ai ii c is1 is2 ra ->
     is_cmd (S ai) ii c (ais :: is1) (ais :: is2) ra
 | isc_cmd ii c is ra ais1 ais2:
     app_is_cmd ii c ais1 ais2 ra ->
    is_cmd O ii c (ais1 :: is) (ais2 :: is) ra.

Inductive app_send_intent_existing:
  app_inst_set -> iid -> stm -> nat -> ret_addr -> app_inst_set -> Prop :=
 | asie_next ais1 ii s n ra ais2 inst:
    app_send_intent_existing ais1 ii s n ra ais2 -> app_send_intent_existing (inst::
    ais1) ii s n ra (inst::ais2)
 | asie_send ais ii s n old_ra ra m:
    app_send_intent_existing ((Inst ii m s_null old_ra) :: ais) ii s n ra
```

```
                                      ((Inst ii m (subst_stm ARG (v_nat n) s) ra) ::
    ais).

Inductive app_send_intent_new:
  app_inst_set -> iid -> stm -> nat -> ret_addr -> app_inst_set -> Prop :=
| asin_send ais ii s n ra:
   app_send_intent_new ais ii s n ra ((Inst ii empty_mem (subst_stm ARG (v_nat n) s)
    ra) :: (ais)).

Inductive update_app_label_map_and_inst_set:
  label_map -> inst_set ->
  aid -> app_label_map -> app_inst_set ->
  label_map -> inst_set -> Prop :=
  | upd_next lm1 is1 ai new_alm new_ais lm2 is2 alm ais:
      update_app_label_map_and_inst_set lm1 is1 ai new_alm new_ais lm2 is2 ->
      update_app_label_map_and_inst_set
        (alm :: lm1) (ais :: is1) (S ai) new_alm new_ais (alm :: lm2) (ais :: is2)
  | upd_update lm is old_alm new_alm old_ais new_ais:
      update_app_label_map_and_inst_set
        (old_alm :: lm) (old_ais :: is) O new_alm new_ais (new_alm :: lm) (new_ais ::
     is).

Inductive app_remove_instance:
  list (iid*label) -> app_inst_set -> iid -> list (iid*label) -> app_inst_set -> Prop
     :=
  | are_next dll1 dll2 ais1 ais2 ii inst_l inst:
(*  beq_iid ii i2 = false -> *)
    app_remove_instance dll1 ais1 ii dll2 ais2 ->
    app_remove_instance ( inst_l :: dll1) (inst :: ais1) ii ( inst_l :: dll2) (inst
    :: ais2)
| are_remove dll ais ii l m s ra:
    app_remove_instance ( (ii,l) :: dll) ((Inst ii m s ra) :: ais) ii dll ais.

Inductive Busy_ais : iid -> app_inst_set -> Prop :=
  | busya_busy ii m s ra ais:
    s <> s_null -> Busy_ais ii (Inst ii m s ra::ais)
  | busya_next ii ais inst:
    Busy_ais ii ais -> Busy_ais ii (inst::ais).

Inductive Busy : aid -> iid -> inst_set -> Prop :=
  | busy_busy ii is ais:
    Busy_ais ii ais -> Busy O ii (ais::is)
  | busy_next ai ii is ais:
    Busy ai ii is -> Busy (S ai) ii (ais::is).

(* Execute a Call command *)
Inductive execute_call :
  app_set ->
  label_map -> inst_set -> pqueue ->
  label ->
  aid -> nat -> ret_addr ->
  label_map -> inst_set -> pqueue -> Prop :=
  | ecall_none apps lm is pq caller_dl ai n ret:
      ai >= length apps \/ has_app_label_map lm ai NA ->
      execute_call apps lm is pq caller_dl ai n ret lm is pq
  | ecall_sl apps lm is pq caller_dl ai n ra lm2 is2 sl dl ii new_ais:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      lte caller_dl dl ->
      app_send_intent_existing (nth ai is []) ii (app_stm apps ai) n ra new_ais ->
      update_app_label_map_and_inst_set lm is ai
          (Single_Instance_App_Launched sl dl ii) new_ais lm2 is2 ->
      execute_call apps lm is pq caller_dl ai n ra lm2 is2 pq
  | ecall_sl_pending apps lm is pq caller_dl ai n ra sl dl ii:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      lte caller_dl dl ->
      Busy ai ii is ->
      execute_call apps lm is pq caller_dl ai n ra lm is (Pend ai n caller_dl (pCall
     ra) :: pq)
  | ecall_sl_block apps lm is pq caller_dl ai n ret sl dl ii:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      nlte caller_dl dl ->
      execute_call apps lm is pq caller_dl ai n ret lm is pq
  | ecall_su apps lm is pq caller_dl ai n ra lm2 is2 sl prev new_ais:
      has_app_label_map lm ai (Single_Instance_App_Unlaunched sl prev) ->
      lte caller_dl sl ->
      app_send_intent_new (nth ai is []) (increment_iid prev) (app_stm apps ai) n ra
     new_ais ->
      update_app_label_map_and_inst_set lm is ai
          (Single_Instance_App_Launched sl sl (increment_iid prev) ) new_ais lm2 is2
     ->
      execute_call apps lm is pq caller_dl ai n ra lm2 is2 pq
  | ecall_su_block apps lm is pq caller_dl ai n ret sl prev:
      has_app_label_map lm ai (Single_Instance_App_Unlaunched sl prev) ->
      nlte caller_dl sl ->
      execute_call apps lm is pq caller_dl ai n ret lm is pq
  | ecall_m apps lm is pq caller_dl ai n ra lm2 is2 sl dll ii new_ais:
      has_app_label_map lm ai (Multi_Instance_App sl dll) ->
      lte caller_dl sl ->
      next_iid dll = ii ->
      app_send_intent_new (nth ai is []) ii (app_stm apps ai) n ra new_ais ->
      update_app_label_map_and_inst_set lm is ai
          (Multi_Instance_App sl ( (ii,sl) :: dll) ) new_ais lm2 is2 ->
      execute_call apps lm is pq caller_dl ai n ra lm2 is2 pq
  | ecall_m_block apps lm is pq caller_dl ai n ret sl dll:
      has_app_label_map lm ai (Multi_Instance_App sl dll) ->
      nlte caller_dl sl ->
      execute_call apps lm is pq caller_dl ai n ret lm is pq.

(* Execute a return call *)
Inductive execute_return :
  app_set ->
  label_map -> inst_set -> pqueue ->
  label ->
  aid -> iid -> nat ->
  inst_set -> pqueue -> Prop :=
  | ereturn_sl apps lm is pq caller_dl ai ii n is2 sl dl new_ais:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      lte caller_dl dl ->
      app_send_intent_existing (nth ai is []) ii (app_stm apps ai) n None new_ais ->
      update_app_label_map_and_inst_set lm is ai
          (Single_Instance_App_Launched sl dl ii) new_ais lm is2 ->
      execute_return apps lm is pq caller_dl ai ii n is2 pq
  | ereturn_sl_pending apps lm is pq caller_dl ai ii n sl dl:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      lte caller_dl dl ->
      Busy ai ii is ->
      execute_return apps lm is pq caller_dl ai ii n is (Pend ai n caller_dl (pReturn
    ii) :: pq)
  | ereturn_sl_block apps lm is pq caller_dl ai ii n sl dl:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      nlte caller_dl dl ->
      execute_return apps lm is pq caller_dl ai ii n is pq
  | ereturn_m apps lm is pq caller_dl ai ii n is2 sl dll dl new_ais:
      has_app_label_map lm ai (Multi_Instance_App sl dll) ->
      has_inst_label dll ii (Some dl) ->
      lte caller_dl dl ->
      app_send_intent_existing (nth ai is []) ii (app_stm apps ai) n None new_ais ->
      update_app_label_map_and_inst_set lm is ai
          (Multi_Instance_App sl dll) new_ais lm is2 ->
      execute_return apps lm is pq caller_dl ai ii n is2 pq
  | ereturn_m_pending apps lm is pq caller_dl ai ii n sl dll dl:
      has_app_label_map lm ai (Multi_Instance_App sl dll) ->
      has_inst_label dll ii (Some dl) ->
      lte caller_dl dl ->
      Busy ai ii is ->
      execute_return apps lm is pq caller_dl ai ii n is (Pend ai n caller_dl (pReturn
  ii) :: pq)
  | ereturn_m_block apps lm is pq caller_dl ai ii n sl dll dl:
      has_app_label_map lm ai (Multi_Instance_App sl dll) ->
      has_inst_label dll ii (Some dl) ->
      nlte caller_dl dl ->
      execute_return apps lm is pq caller_dl ai ii n is pq.

(* Execute a Raise command *)
Inductive execute_raise :
  label_map ->
  aid -> iid ->
  label ->
  label_map -> Prop :=
  | eraise_s lm ai ii l lm2 sl dl sl2:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      lte_bool dl l = true ->
      (nlte l sl /\ sl2 = l \/ lte l sl /\ sl2 = sl) ->
      update_app_label_map lm ai (Single_Instance_App_Launched sl2 l ii) lm2 ->
      execute_raise lm ai ii l lm2
  | eraise_s_block lm ai ii l sl dl:
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      lte_bool dl l = false ->
      execute_raise lm ai ii l lm
  | eraise_m lm ai ii l lm2 sl dll dl dll2:
     has_app_label_map lm ai (Multi_Instance_App sl dll) ->
     has_inst_label dll ii (Some dl) ->
     lte_bool dl l = true ->
     update_inst_label dll ii l dll2 ->
     update_app_label_map lm ai (Multi_Instance_App sl dll2) lm2 ->
     execute_raise lm ai ii l lm2
  | eraise_m_block lm ai ii l sl dll dl:
     has_app_label_map lm ai (Multi_Instance_App sl dll) ->
     has_inst_label dll ii (Some dl) ->
     lte_bool dl l = false ->
     execute_raise lm ai ii l lm.
```

```

Inductive remove_ra_ais:
  app_inst_set -> aid -> iid -> app_inst_set -> Prop :=
  | rrais_nil ai ii: remove_ra_ais [] ai ii []
  | rrais_skip ai ii ai2 ii2 ais1 ais2 ii’ m s:
      remove_ra_ais ais1 ai ii ais2 ->
      (beq_aid ai2 ai = false \/ beq_iid ii2 ii = false) ->
      remove_ra_ais (Inst ii’ m s (Some (ai2,ii2)) :: ais1) ai ii
                         (Inst ii’ m s (Some (ai2,ii2)) :: ais2)
  | rrais_remove ai ii ais1 ais2 ii’ m s:
      remove_ra_ais ais1 ai ii ais2 ->
      remove_ra_ais (Inst ii’ m s (Some (ai,ii)) :: ais1) ai ii
                         (Inst ii’ m s None :: ais2)
  | rrais_none ai ii ais1 ais2 ii’ m s:
      remove_ra_ais ais1 ai ii ais2 ->
      remove_ra_ais (Inst ii’ m s None :: ais1) ai ii
                         (Inst ii’ m s None :: ais2).


Inductive remove_ra_is:
  inst_set -> aid -> iid -> inst_set -> Prop :=
  | rris_nil ai ii: remove_ra_is [] ai ii []
  | rris_con ai ii is1 is2 ais1 ais2:
      remove_ra_ais ais1 ai ii ais2 ->
      remove_ra_is is1 ai ii is2 ->
      remove_ra_is (ais1 :: is1) ai ii (ais2 :: is2).

Inductive remove_pending:
  pqueue -> aid -> iid -> pqueue -> Prop :=
  | rmp_nil ai ii: remove_pending [] ai ii []
  | rmp_call_skip_none ai ii pq1 pq2 ai’ n caller_dl:
      remove_pending pq1 ai ii pq2 ->
      remove_pending (Pend ai’ n caller_dl (pCall None) :: pq1) ai ii
                          (Pend ai’ n caller_dl (pCall None) :: pq2)
  | rmp_call_skip_some ai ii pq1 pq2 ai’ n caller_dl ai2 ii2:
      remove_pending pq1 ai ii pq2 ->
      (beq_aid ai2 ai = false \/ beq_iid ii2 ii = false) ->
      remove_pending (Pend ai’ n caller_dl (pCall (Some (ai2,ii2))) :: pq1) ai ii
                          (Pend ai’ n caller_dl (pCall (Some (ai2,ii2))) ::
    pq2)
  | rmp_call_remove ai ii pq1 pq2 ai’ n caller_dl:
      remove_pending pq1 ai ii pq2 ->
      remove_pending (Pend ai’ n caller_dl (pCall (Some (ai,ii))) :: pq1) ai ii
                          (Pend ai’ n caller_dl (pCall None) :: pq2)
  | rmp_return_skip ai ii pq1 pq2 ai2 n caller_dl ii2:
      remove_pending pq1 ai ii pq2 ->
      (beq_aid ai2 ai = false \/ beq_iid ii2 ii = false) ->
      remove_pending (Pend ai2 n caller_dl (pReturn ii2) :: pq1) ai ii
                          (Pend ai2 n caller_dl (pReturn ii2) :: pq2)
  | rmp_return_remove ai ii pq1 pq2 n caller_dl:
      remove_pending pq1 ai ii pq2 ->
      remove_pending (Pend ai n caller_dl (pReturn ii) :: pq1) ai ii pq2.


Inductive execute_exit:
  label_map -> inst_set -> pqueue ->
  aid -> iid ->
  label_map -> inst_set -> pqueue -> Prop :=
  | eexit_s lm lm' ai ii is is' is'' pq pq' sl dl :
      has_app_label_map lm ai (Single_Instance_App_Launched sl dl ii) ->
      remove_ra_is is ai ii is' ->
      update_app_label_map_and_inst_set lm is' ai
          (Single_Instance_App_Unlaunched sl ii) [] lm' is'' ->
      remove_pending pq ai ii pq' ->
      execute_exit lm is pq ai ii lm' is'' pq'
  | eexit_m lm lm' ai ii is is' is'' pq pq' sl dll dll' new_ais:
      has_app_label_map lm ai (Multi_Instance_App sl dll) ->
      remove_ra_is is ai ii is' ->
      app_remove_instance dll (nth ai is' []) ii dll' new_ais ->
      update_app_label_map_and_inst_set lm is' ai
          (Multi_Instance_App sl dll') new_ais lm' is'' ->
      remove_pending pq ai ii pq' ->
      execute_exit lm is pq ai ii lm' is'' pq'.


Definition out : Type := (nat * label).

(* Execute a comand *)
Inductive execute :
  app_set ->
  label_map -> inst_set -> pqueue ->
  option out ->
  label_map -> inst_set -> pqueue ->
  Prop :=
  | exec_call apps lm1 lm2 is1 is2 pq1 pq2 callee ai ii n expect_ret ra sl dl is':
      is_cmd ai ii (Call callee (e_nat n) expect_ret) is1 is' ra ->
      has_label lm1 ai ii (Some (sl,dl)) ->
      execute_call apps lm1 is' pq1 dl callee n ra lm2 is2 pq2 ->
      execute apps lm1 is1 pq1 (Some (n,dl)) lm2 is2 pq2
  | exec_raise apps lm1 is1 pq ai ii l lm2 is2 ra:
      is_cmd ai ii (Raise l) is1 is2  ra ->
      execute_raise lm1 ai ii l lm2 ->
      execute apps lm1 is1 pq None lm2 is2 pq
  | exec_return apps lm is1 is2 pq1 pq2 ai ii n callee callee_ii sl dl is':
      is_cmd ai ii (Return (e_nat n)) is1 is' (Some (callee,callee_ii)) ->
      has_label lm ai ii (Some (sl,dl)) ->
      execute_return apps lm is' pq1 dl callee callee_ii n is2 pq2 ->
      execute apps lm is1 pq1 (Some (n,dl)) lm is2 pq2
  | exec_return_no_ra apps lm is1 is2 pq ai ii n sl dl:
      is_cmd ai ii (Return (e_nat n)) is1 is2 None ->
      has_label lm ai ii (Some (sl,dl)) ->
      execute apps lm is1 pq (Some (n,dl)) lm is2 pq
  | exec_exit apps lm1 lm2 is1 is2 pq1 pq2 ai ii n callee callee_ii sl dl is' is'' pq
    ':
      is_cmd ai ii (Exit (e_nat n)) is1 is' (Some (callee,callee_ii)) ->
      beq_aid callee ai = false \/ beq_iid callee_ii ii = false ->
      has_label lm1 ai ii (Some (sl,dl)) ->
      execute_exit lm1 is' pq1 ai ii lm2 is'' pq' ->
      execute_return apps lm2 is'' pq' dl callee callee_ii n is2 pq2 ->
      execute apps lm1 is1 pq1 (Some (n,dl)) lm2 is2 pq2
  | exec_exit_no_return apps lm1 lm2 is1 is2 pq1 pq2 ai ii n sl dl is' ra:
      is_cmd ai ii (Exit (e_nat n)) is1 is' ra ->
      ra = (Some (ai,ii)) \/ ra = None ->
      has_label lm1 ai ii (Some (sl,dl)) ->
      execute_exit lm1 is' pq1 ai ii lm2 is2 pq2 ->
      execute apps lm1 is1 pq1 (Some (n,dl)) lm2 is2 pq2.


Inductive process_pending':
  app_set ->
  label_map -> inst_set -> pqueue -> pqueue ->
  label_map -> inst_set -> pqueue -> Prop :=
  | pp_next apps lm1 lm2 is1 is2 pq1 pq2 p pq':
    process_pending' apps lm1 is1 pq1 (pq' ++ [p]) lm2 is2 pq2 ->
    process_pending' apps lm1 is1 (p :: pq1) pq' lm2 is2 pq2
  | pp_call apps lm1 lm2 is1 is2 pq pq' caller_dl callee n ra:
    execute_call apps lm1 is1 (pq' ++ pq) caller_dl callee n ra lm2 is2 (pq' ++ pq)
    ->
    process_pending' apps lm1 is1 (Pend callee n caller_dl (pCall ra) :: pq) pq'
        lm2 is2 (pq' ++ pq)
  | pp_return apps lm is1 is2 pq pq' callee callee_ii n caller_dl:
    execute_return apps lm is1 (pq' ++ pq) caller_dl callee callee_ii n is2 (pq' ++
    pq) ->
    process_pending' apps lm is1 (Pend callee n caller_dl (pReturn callee_ii) :: pq)
    pq'
        lm is2 (pq' ++ pq).

Definition process_pending apps lm1 is1 pq1 lm2 is2 pq2 :=
  process_pending' apps lm1 is1 pq1 [] lm2 is2 pq2.

Inductive step_type : Type :=
 | tau
 | output (o : out).

(* The whole system takes a step if either an app instance takes an internal step or
    a command is executed *)
Inductive sys_step: sys -> sys -> step_type -> Prop :=
  | ss_tau_iss apps lm is1 is2 pq:
      is_step is1 is2 ->
      sys_step (Sys apps lm is1 pq) (Sys apps lm is2 pq) tau
  | ss_tau_cmd apps lm1 lm2 is1 is2 pq1 pq2:
      execute apps lm1 is1 pq1 None lm2 is2 pq2 ->
      sys_step (Sys apps lm1 is1 pq1) (Sys apps lm2 is2 pq2) tau
  | ss_tau_pend apps lm1 lm2 is1 is2 pq1 pq2 :
      process_pending apps lm1 is1 pq1 lm2 is2 pq2 ->
      sys_step (Sys apps lm1 is1 pq1) (Sys apps lm2 is2 pq2) tau
  | ss_output_cmd apps lm1 lm2 is1 is2 pq1 pq2 v l:
      execute apps lm1 is1 pq1 (Some (v,l)) lm2 is2 pq2 ->
      sys_step (Sys apps lm1 is1 pq1) (Sys apps lm2 is2 pq2) (output (v,l)).

Definition trace := list out.

End System.
```

projection.v

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Require Import Coq.Program.Equality.
Require Import Omega.
Require Import Coq.Program.Tactics.
Require Export Coq.Structures.Equalities.

Require Export labelmap.
Require Export lattice.
Require Export core.
Require Export system.
Require Export sys_preservation.

Module Projection.


Import ListNotations.

Import LabelMap.
Import SecurityLattice.
Import CoreLanguage.
Import System.
Import Preservation.


Fixpoint proj_dll (n: nat) (k : label) (dll : list (iid * label) ) : list (iid *
    label) :=
  match dll with
  | [] => []
  | (ii, l) :: dll => if lte_bool l k then
      (Iid n , l) :: proj_dll (S n) k dll else proj_dll n k dll
  end.

Fixpoint proj_ii_dll (n: nat) (k: label) (dll : list (iid * label)) (ii:iid) : option
     iid :=
match dll with
| [] => None
| (ii2,l) :: dll =>
  if beq_iid ii ii2 then
    if lte_bool l k then Some (Iid n) else None
  else
    if lte_bool l k then proj_ii_dll (S n) k dll ii else proj_ii_dll n k dll ii
end.

Definition proj_ii_alm (k: label) (alm : option app_label_map)  (ii:iid) : option iid
     :=
  match alm with
  | Some (Single_Instance_App_Launched sl dl ii2) =>
      if (beq_iid ii2 ii)  && (lte_bool dl k) then Some ii else None
  | Some (Multi_Instance_App _ dll) => proj_ii_dll 0 k dll ii
  | _ => None
  end.

Definition proj_ii_lm (k: label) (lm: label_map) (ai: aid) (ii:iid) : option (aid*iid
    ) :=
  match proj_ii_alm k (get_app_label_map lm ai) ii with
  | None => None
  | Some ii' => Some (ai,ii')
  end.

Definition proj_ra (k: label) (lm: label_map) (ra : ret_addr) : ret_addr :=
  match ra with
  | None => None
  | Some (ai,ii) => proj_ii_lm k lm ai ii
  end.

Fixpoint proj_ais_m (n: nat) (k: label) (lm: label_map) (dll : list (iid * label))
    ais : app_inst_set :=
  match dll, ais with
  | (_,l) :: dll, Inst _ m s ra :: ais =>
      if lte_bool l k then
        Inst (Iid n) m s (proj_ra k lm ra) :: proj_ais_m (S n) k lm dll ais
      else proj_ais_m n k lm dll ais
  | _ , _ => []
  end.


Definition proj_alm (k : label) (alm : app_label_map) : app_label_map:=
  match alm with
  | Multi_Instance_App sl dll => Multi_Instance_App sl (proj_dll 0 k dll)
  | Single_Instance_App_Unlaunched sl prev => if lte_bool sl k then
                          alm else NA
  | Single_Instance_App_Launched sl dl ii => if lte_bool dl k then
                          alm else NA
  | NA => NA
  end.


Definition proj_ra_inst (k : label) (lm : label_map) (inst : instance) : instance :=
  match inst with
  | Inst ii m s ra => Inst ii m s (proj_ra k lm ra) end.

Definition proj_ais (k : label) (lm : label_map) (alm : app_label_map) ais :
    app_inst_set:=
  match alm with
  | Single_Instance_App_Unlaunched sl prev => if lte_bool sl k then
                          map (proj_ra_inst k lm) ais else []
  | Single_Instance_App_Launched sl dl ii => if lte_bool dl k then
                          map (proj_ra_inst k lm) ais else []
  | Multi_Instance_App sl dll => (proj_ais_m 0 k lm dll ais)
  | NA => []
  end.


Definition proj_lm (k : label) (lm: label_map) : label_map :=
  map (proj_alm k) lm.

Definition proj_is (k: label) (LM: label_map) (lm: label_map) (is : inst_set) :
    inst_set :=
  map (fun a => proj_ais k LM (fst a) (snd a)) (combine lm is).

Definition proj_pend (k: label) (LM: label_map)
    (alm : option app_label_map) (pt: pendingType) : option pendingType :=
  match pt with
  | pCall ra => match alm with
    | Some (Single_Instance_App_Unlaunched l _)
    | Some (Single_Instance_App_Launched _ l _)
    | Some (Multi_Instance_App l _) =>
```

```
      if lte_bool l k then Some (pCall (proj_ra k LM ra) ) else None
    | _ => None
    end
  | pReturn ii => match alm with
    | Some (Single_Instance_App_Launched _ dl ii2) =>
      if beq_iid ii ii2 && lte_bool dl k then Some pt else None
    | Some (Multi_Instance_App _ dll) =>
      match proj_ii_dll 0 k dll ii with
      | Some ii' => Some (pReturn ii') | None => None end
    | _ => None
    end
  end.

Definition proj_pending (k: label) (LM: label_map) (lm: label_map) (p: pending) :
    option pending :=
  match p with
  | Pend callee n dl pt =>
    match proj_pend k LM (get_app_label_map lm callee) pt with
    | None => None
    | Some pt => Some (Pend callee n dl pt)
    end
  end.

Fixpoint proj_pq (k : label) (LM:label_map) (lm: label_map) (pq: pqueue) : pqueue :=
  match pq with
  | [] => []
  | p :: pq =>
    match proj_pending k LM lm p with
    | None => proj_pq k LM lm pq
    | Some p => p :: proj_pq k LM lm pq
    end
  end.


Definition proj (k : label) (s: sys) : sys :=
  match s with (Sys apps lm is pq) =>
      (Sys apps (proj_lm k lm) (proj_is k lm lm is) (proj_pq k lm lm pq))
  end.

Definition proj_trace (k : label) (t: trace) : trace :=
  filter (fun a => lte_bool (snd a) k) t.

End Projection.
```