Practical DIFC Enforcement on Android

Adwait Nadkarni, Benjamin Andow, William Enck
{*anadkarni,beandow,whenck*}*@ncsu.edu*
North Carolina State University

Somesh Jha
jha@cs.wisc.edu
University of Wisconsin-Madison

Abstract

Smartphone users often use private and enterprise data with untrusted third party applications. The fundamental lack of secrecy guarantees in smartphone OSes, such as Android, exposes this data to the risk of unauthorized exfiltration. A natural solution is the integration of secrecy guarantees into the OS. In this paper, we describe the challenges for decentralized information flow control (DIFC) enforcement on Android. We propose context-sensitive DIFC enforcement via *lazy polyinstantiation* and practical and secure network export through *domain declassification*. Our DIFC system, *Weir*, is backwards compatible by design, and incurs less than 4 ms overhead for component startup. With *Weir*, we demonstrate practical and secure DIFC enforcement on Android.

1 Introduction

Application-based modern operating systems, such as Android, thrive on their rich application ecosystems. Applications integrate with each other to perform complex user tasks, providing a seamless user experience. To work together, applications share user data with one another. Such sharing exposes the user's private and enterprise information to the risk of exfiltration from the device. For example, an email attachment opened in a third party document editor (e.g., *WPS Office*) could be exported if the editor was malicious or compromised.

Android's permission framework is used to protect application data. However, permissions are only enforced at the first point of access. Data once copied into the memory of an untrusted application can be exported. This problem is generic in OSes that provide only data protection, but not data secrecy, and can be solved by integrating information flow secrecy guarantees.

Classic information flow control (IFC) [8] only captures well-known data objects through a centralized policy. On Android, data is often application-specific (e.g., email attachments, notes). Therefore, Android requires decentralized IFC (DIFC) [21, 26, 44, 49], which allows data owners (i.e., applications) to specify the policy for their own data objects.

Although DIFC systems have been proposed for Android [19,28,46], existing enforcement semantics cannot achieve both security and practicality. For instance, an Android application's components are instantiated in the same process by default, even when executing separate user tasks. Since the various secrecy contexts from the tasks share state in process memory, DIFC enforcement on the process is hard, as the combined restrictions from all secrecy contexts would make individual components unusable. Prior approaches solve this problem by eliminating Android's default behavior of application multitasking, and in ways detrimental to backwards compatibility, i.e., 1) killing processes per new call, which could result in dangling state, or 2) blocking until the application voluntarily exits, which could lead to deadlocks.

Similarly, different secrecy contexts may share state on storage through common application files (e.g., application settings). Proposals to separate this shared state on storage (e.g., Maxoid [46]) either deny access to application resources or require applications to be modified. To summarize, prior DIFC proposals for Android cannot separate shared state in memory and on storage while maintaining security and backwards compatibility.

In this paper, we present *Weir*,[1] a practical DIFC system for Android. *Weir* allows data owner applications to set secrecy policies and control the export of their data to the network. Apart from the data owners, and applications that want to explicitly use *Weir* to change their labels, all other applications can execute unmodified. *Weir* solves the problem of shared state by separating memory and storage for different secrecy contexts through *polyinstantiation*. That is, *Weir* creates and manages instances of the application, its components, and its stor-

[1] Weir: A small dam that alters the flow of a river.

age for each secrecy context that the application is called from, providing availability along with context-sensitive separation. Our model is transparent to applications; i.e., applications that do not use *Weir* may execute oblivious to *Weir's* enforcement of secrecy contexts.

We term our approach as *"lazy" polyinstantiation*, as *Weir* creates a new instance of a resource only if needed, i.e., if there is no existing instance whose secrecy context matches the caller's. Additionally, *Weir* provides the novel primitive of *domain declassification* for practical and secure declassification in Android's network-driven environment. Our approach allows data owners to articulate trust in the receiver of data (i.e., trusted network domain). This paper makes the following contributions:

- *We identify the challenges of integrating DIFC* into Android. Using these challenges, we then derive the goals for designing DIFC enforcement for Android.
- *We introduce the mechanism of "lazy" polyinstantiation* for context-sensitive separation of the shared state. Further, we provide the primitive of *Domain Declassification* for practical declassification in Android's network-driven environment.
- *We design and implement Weir* on Android. *Weir* incurs less than 4ms overhead for starting components. *Weir's* design ensures backwards compatibility. We demonstrate *Weir's* utility with a case study using the *K-9 Mail* application.

While *Weir* presents a mechanism that is independent of the actual policy syntax, our implementation uses the policy syntax of the Flume DIFC model [21]. *Weir* extends Flume by allowing implicit label propagation, i.e., *floating labels*, for backwards compatibility with unmodified applications. Since floating labels are *by themselves* susceptible to high bandwidth information leaks [8], we show how *Weir's* use of floating labels is inherently resistant to such leaks. Note that while language-level IFC models [40–42] often incorporate checks that prevent implicit flows due to floating labels, our solution addresses the challenges faced by OS-level floating label DIFC systems [19, 44]. Finally, we note that *Weir* provides practical DIFC enforcement semantics for Android, and the usability aspect of DIFC policy and enforcement will be explored in future work.

In the remainder of this paper, we motivate the problem (Section 2), and describe the challenges in integrating DIFC on Android (Section 3). We then describe the design (Section 4), implementation (Section 5) and security (Section 6) of *Weir*, followed by the evaluation (Section 7), and a case study (Section 8). We then discuss the limitations (Section 9), related work (Section 10) and conclude (Section 11).

2 Motivation and Background

We now motivate the need for data secrecy on Android. This is followed by background on DIFC and Android.

2.1 Motivating Example and Threat Model

Consider Alice, an enterprise user in a BYOD (bring your own device) context. Alice receives an email in the enterprise *OfficeEmail* application with an attached report. She edits the report in a document editor, *WPS Office*, and saves a copy on the SD card, accessible to all applications that have the `READ_EXTERNAL_STORAGE` permission. Later, Alice uses the *ES File Explorer* to browse for the report, edits it in *WPS Office*, and then shares it with *OfficeEmail* to reply to the initial email.

To perform their functions, untrusted third party data managers such as *ES File Explorer* require broad storage access. Even without direct access, user-initiated data sharing grants data editors like *WPS Office* access to confidential data. If *ES File Explorer* or *WPS Office* were malicious or compromised, they could export Alice's confidential data to an adversary's remote server.

Threat Model and Assumptions: We seek to enable legitimate use of third party applications to process secret user data, while preventing accidental and malicious data disclosure to the network. For this purpose, our solution, *Weir*, must mediate network access, and track flows of secret data 1) among applications and 2) to/from storage.

Weir's trusted computing base (TCB) consists of the Android OS (i.e., kernel and system services), and core network services (e.g., DNS). *Weir* assumes a non-rooted device, as an adversary with superuser privileges may compromise OS integrity. Further, we assume correct policy specification by the data owner applications, specifically regarding declassification. To prevent timing and covert channels based on shared hardware resources (e.g., a hardware cache), the only alternative is denying data access to secret data or the shared resource. *Weir* does not defend against such channels, which are notoriously hard to prevent in DIFC OSes in general.

2.2 Why Information Flow Control (IFC)?

Android uses its permission framework to protect user data. While permissions provide protection at the first point of access, the user or the data owner application (e.g., *OfficeEmail*) have no control over the flow of data once it is shared with another application (e.g., *WPS Office*). Unauthorized disclosure is an information flow problem that permissions are not designed to solve.

Information flow control (IFC) [8] can provide data secrecy and prevent unauthorized disclosure, through the definition and enforcement of the allowable data flows in the system. In an IFC system, subjects (e.g., processes)

and objects (e.g., files) are labeled with predefined security classes (e.g., top-secret, secret, confidential). The secrecy policy determines the data flow (i.e., ordering) between any two classes based on a partially ordered finite lattice. Labels may also be joined to form a higher label in the lattice. For secrecy, data can only flow up, i.e., to a higher security class [6], and violating flows require declassification by the policy administrator.

2.3 What is DIFC?

A centralized IFC policy can only describe the secrecy constraints for well-known data objects (e.g., location, IMEI). Decentralized IFC (DIFC) [26] extends the IFC lattice to include unknown subjects and objects, and is appropriate for protecting application-specific data, such as Alice's secret report received by *OfficeEmail*. We now describe some fundamental aspects of DIFC.

Label Definition: In a DIFC system, security principals create labels (i.e., security classes) for their own secret data. On Android, decentralized label definition would allow apps to control the flow of their data by creating and managing labels for their data. Note that while DIFC also provides integrity, our description is for data secrecy as it is the most relevant to the problem in Section 2.1.

Label Changes and *Floating* Labels: The finality of subject and object label assignment is called *tranquility* [6], a property of mandatory protection systems. Tranquility constraints have to be relaxed for DIFC policy. Subjects may then change (raise or lower) their labels "safely", i.e., with authorization from the data owners whose security classes are involved in the change.

Explicit label changes offer flexibility over immutable labels, but are not practical in environments where communication is user-directed and unpredictable a priori. *Floating* labels (e.g., in Asbestos [44]) make DIFC compatible with unmodified apps in such cases, by allowing seamless data flows through implicit label propagation. That is, the caller's and the callee's labels are joined, and the resultant label is set as the callee's label.

Declassification: The network is considered to be public, and any network export requires declassification by the data owner. Data owners may choose to explicitly declassify every request to export their data, or allow trusted third parties to declassify on their behalf. While the former is impractical when frequent declassification is required, the latter bloats the data owner's TCB.

System Integration: One of the first steps while integrating data secrecy into an existing OS is the selection of the subject for data flow tracking. Fine-grained dynamic taint tracking (e.g.,TaintDroid [13]) labels programming language objects to provide precision, but does not protect against implicit flows. OS-based DIFC approaches [21, 49] adopt the better mediated OS process granularity, but incur high false positives; i.e., functions sharing the process with unrelated functions that read secret data may be over-restricted. While secure process-level labeling is desired, practical DIFC enforcement must minimize its impact on functionality.

2.4 Android Background

The Android application model consists of four components, namely *activities* for the user interface (UI), *services* for background processing, *content providers* to provide a uniform interface to application data, and *broadcast receivers* to handle broadcast events.

Component Instantiation: Services and content providers run in the background, and have one active instance. Activities can have multiple instances, and the default "standard" launch behavior for activities is to start a new instance per call. Developers use Android's "android:launchMode" manifest attribute to manage activity instances as follows: *SingleTop* activities are resumed for new calls if they already exist at the top of the activity stack. *SingleTask* and *SingleInstance* activities are similar in that they are allocated an instance in a separate user task and every call to such an activity resumes the same instance; the only difference being that the latter can be the only activity in its task.

Inter-Component Communication: Inter-component communication on Android can be 1) indirect or 2) direct. Indirect communication is an asynchronous call from one component to another, through the *Activity Manager* service (e.g., startActivity, bindService). Direct communication involves a synchronous *Binder* remote procedure call (RPC) to the callee using the callee's "Binder object". While direct communication bypasses the Activity Manager, its setup involves one mediated indirect call to retrieve the callee's Binder object. For example, the first operation executed on a content provider (e.g., query, update) by a caller is routed through the Activity Manager, which retrieves the content provider's Binder object and loads it into the caller's memory. Future calls are routed directly to the content provider.

3 DIFC Challenges on Android

In this section, we discuss the four aspects of Android that make DIFC enforcement challenging. Further, we describe how previous Android DIFC systems fare with respect to the challenges, and state the design goals for practical DIFC enforcement on Android.

1. Multitasking on Android: Android's UI is organized into user tasks representing the user's high-level objectives. An application can be involved in multiple tasks

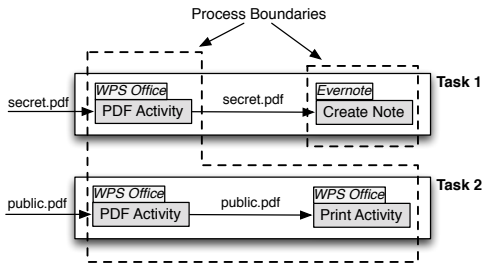


Figure 1: Shared state in memory: Instances of *WPS Office* performing different tasks in the same process.

by default. Further, a default activity can be instantiated multiple times in one or more concurrent tasks [3].

Figure 1 shows two tasks. In *Task 1*, the user opens a secret PDF (e.g., a contract) with *WPS Office*, which loads it in its *PDF Activity*, and shares it with the *Evernote* app. In *Task 2*, the user opens a non-confidential PDF (e.g., a published paper) in another instance of *WPS Office's PDF Activity*. Further, in *Task 2*, the user chooses to print the PDF, which is then sent to *WPS Office's* internal *Print Activity*. As seen in Figure 1, multiple activities of the *WPS Office* app as well as multiple instances of the *PDF Activity* run in the same process, and may share data in memory (e.g., via global variables).

As the two instances of the *PDF Activity* are instantiated with data of different secrecy requirements (i.e., secret.pdf and public.pdf), they run in different *secrecy contexts*. Enforcing the DIFC policy on the process due to the sensitive nature of *Task 1* would also unjustifiably restrict the non-sensitive *Task 2*. A naive solution of forcing every component to start in a separate process may break components; e.g., *Print Activity* may try to access a global variable initialized by the *PDF Activity*, and may crash if the *PDF Activity* is not in the same process. To summarize, component instances in various secrecy contexts often share state in process memory, making process-level enforcement challenging.

2. Background components: As described in Section 2.4, service and content provider components have only one active instance, which is shared among all of an application's instances, and may also communicate with other applications. As a result, various secrecy contexts may mix in a single background component instance.

If floating labels (described in Section 2.3) are applied, then the background component may accumulate the labels of all the secrecy contexts it communicates with, and then propagate its new label back to the components connected to it. This results in a *label explosion*, where the entire system acquires a large, restrictive label that cannot be declassified by any single security principal. Note that background components may run in the shared application process by default. Therefore, restarting a background component's process for each new call is infeasible, as it would crash the other components (e.g., a foreground activity) running in that process.

3. Internal and External Application storage: Android provides each application with its own internal (i.e., private) storage shared amongst all of its runtime instances, irrespective of the secrecy context. For example, both the sensitive and non-sensitive instances of *WPS Office* may access the same user settings in the application's private directory. For availability from all secrecy contexts, storage access enforcement uses floating labels. The propagation of sensitive secrecy labels through shared application files (i.e., shared state on storage) may cause label explosion. The risk and impact of label explosion is higher on the external storage (i.e., the SD card) shared by all applications.

4. Internet-driven environment: Android applications are often connected to the Internet. In such an environment with frequent network export, explicit declassification by the data owner is inefficient. Delegation of the declassification privilege to allow export without the owner's intervention would bloat the application's TCB. Additionally, existing declassification mechanisms described in Section 2.3 make the policy decision based on the identity of the security principal performing the export. On Android, such mechanisms would limit the user to using a small subset of applications for data export (i.e., out of the 2 million applications on Google Play [39]), which would be detrimental to adoption of DIFC on Android.

3.1 Prior DIFC Proposals for Android

We discuss three prior DIFC proposals for Android, namely Aquifer [28], Jia et al. [19] and Maxoid [46], all of which are OS-level DIFC systems. Our objective is to understand the design choices made by these systems, with respect to the challenges described previously.

1. Aquifer: Our prior work, Aquifer [28], provides protection against accidental data disclosure, by tracking the flow of data through applications, and enforcing the declassification policy for network export.

For seamless data sharing between applications, Aquifer uses the floating labels described in Section 2.3. To limit label explosion, Aquifer does not label background components, and hence can only prevent accidental data disclosure. On the other hand, Aquifer labels storage, but does not claim to mitigate label explosion on storage. Further, to prevent different secrecy requirements for data in the memory of a single process, Aquifer disables Android's multi-tasking and restarts the process of the existing instance when the application is called from another secrecy context. Finally, Aquifer's declassification policy allows the data owner to explicitly specify the security principal that may export data,

or a condition on the call chain for implicit export.

2. Jia et al.: The DIFC system by Jia et al. [19] also uses floating labels to support general-purpose applications, but supports strict secrecy policies (i.e., relative to Aquifer) that may restrict data sharing among applications if needed.

Contrary to Aquifer, the system propagates labels to background components, providing stronger protection against malicious data exfiltration. At the same time, the system makes no claims of controlling label explosion via background components or storage. The system uses Flume's capabilities [21] for declassification. This work also acknowledges the challenge of multi-tasking along with DIFC enforcement, and disallows multi-tasking by blocking new calls to an application until all of its components voluntarily exit. Since Android components do not exit by themselves like conventional OS programs, such blocking could potentially lead to deadlocks.

3. Maxoid: Xu and Witchel [46] provide an alternate approach to file system labeling to prevent label explosion in Maxoid, by using file system polyinstantiation [22] to separate differently labeled data on disk.

Maxiod addresses new calls to existing labeled instances in a manner similar to Aquifer's; i.e., by restarting the instance. Additionally, Maxoid prevents access to background components from labeled instances, thereby preventing label explosion, although at the cost of backwards compatibility. On the other hand, Maxoid considers overt data flows through Binder IPC as declassification, unlike the system by Jia et al. that mediates such communication. Finally, Maxoid modifies system content providers (e.g., Contacts) to use a SQL proxy, in order to extend its label separation into system content providers. As a result, Maxoid's storage separation is unavailable for use by content providers in unmodified third party applications.

Takeaways: Prior approaches demonstrate the possibility of DIFC on Android, and make convincing arguments in favor of using floating labels, mainly for backwards compatibility with Android's unpredictable data flows. At the same time, we observe that in prior systems it becomes necessary to relax either security or backwards compatibility in order to use floating labels on Android (e.g., with background components). Additionally, prior approaches recognize the need to separate different secrecy contexts in process memory, but the proposed solutions disable Android's default multi-tasking.[2] Finally, in systems that aim to address label explosion on storage, only separating the shared state on storage without addressing the shared state in memory may be insufficient to support unmodified applications.

[2]Killing existing processes or blocking can result in the killing of unrelated components sharing the process, or deadlocks, respectively.

3.2 Design Goals

Our objective is to design DIFC enforcement that provides security, and is backwards compatible with unmodified applications. Our design goals are as follows:

G1 *Separation of shared state in memory.* DIFC enforcement must ensure that data from different secrecy contexts is always separated in memory, preferably in the memory of different processes. Process-level enforcement can then be used to mediate flows between differently labeled data.

G2 *Separation of shared state on storage.* DIFC enforcement must ensure that data from different secrecy contexts is separate on persistent storage. For mediation by the OS, the separation must be at the level of OS objects (e.g., files, blocks).

G3 *Transparency*. A naive implementation of goals **G1** and **G2** would affect the availability of components and storage. Our system must be transparent, i.e., applications that do not use the DIFC system must be able to operate oblivious to the enforcement.

G4 *Secure and practical declassification for network export.* A DIFC system on Android should provide a declassification primitive that is both feasible (i.e., does not hinder the use of applications) and secure.

4 Weir

In this paper, we propose *Weir*, a practical and secure DIFC system for Android. *Weir's* design is guided by the security and backwards compatibility goals described in Section 3.2. We now briefly describe the specific properties expected from our design, followed by an overview of *Weir* and design details.

Design Properties: Taking a lesson from prior work in Section 3.1, our system must allow seamless data sharing between applications for backwards compatibility with Android's application model. Data flows must be tracked using implicit label propagation (i.e., floating labels), while mitigating the risk of label explosion. More specifically, our system must not deny data access, unless an application explicitly changes its label and fails a label check. Since our goal is to prevent unauthorized data export, network access may be denied if an application tries to export sensitive data to the network in violation of the declassification requirements of the data owner. Finally, our system must mediate all overt data flows, but covert channels existing in Android are not the targets of our system (discussed further in Section 9).

4.1 Overview

In *Weir*, applications define the policy for their data by creating their own *security classes*. *Weir* labels files (as

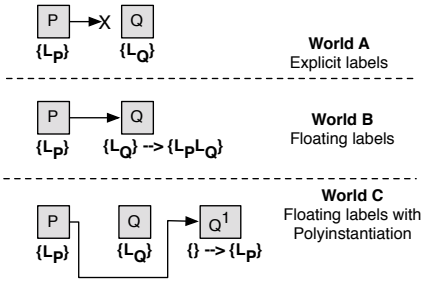


Figure 2: Overview of floating labels w/ polyinstantiation relative to explicit and floating labels.

objects) and processes (as subjects), granting the kernel complete mediation over all data flows among subjects and objects. As *Weir's* contributions are in its policy-agnostic mechanism, we use the generic terminology from Section 2.3 in policy-related discussions. Section 5.1 describes our implementation's policy model.

Weir uses floating labels (described in Section 2.3), as explicit labels are hard to assign a priori in Android, where data flows are often user-directed and unpredictable. However, naive (i.e., context-insensitive) floating label propagation can cause certain components to acquire more labels due to involvement in multiple secrecy contexts, and eventually become unusable. We propose polyinstantiation to make floating labels context sensitive, and hence separate the shared state from different secrecy contexts in memory (**G1**) and on storage (**G2**). Our approach is in principal similar to context sensitive inter-procedural analysis that adds precision by considering the calling context when analyzing the target of a function call (e.g., summary functions and call strings [36], k-CFA [37], and CFL-reachability [31]). To our knowledge, context sensitivity has not been explored in the scenario investigated in this paper. Further, the approach of secure multi-execution [11] also uses multiple executions of the program, but is fundamentally different in many aspects, as we describe in Section 10.

We describe polyinstantiation relative to explicit and floating labels with the example scenario in Figure 2, where an instance of component P with label $\{L_P\}$ tries to send a message to an instance of component Q with a label $\{L_Q\}$, and where $\{L_P\} \neq \{L_Q\}$. In World A where only explicit labels are allowed, the message would be denied as Q would not be able to explicitly change its label to $\{L_P\}$ without a priori knowledge of $P's$ intention to send a message. In World B with floating labels, the flow would be automatically allowed, with $Q's$ new label implicitly set to a join of the two labels. While World B allows seamless communication, it does not prevent the two secrecy contexts (i.e., $\{L_P\}$ and $\{L_Q\}$) from mixing, leading to the challenges we explored in Section 3. In World C, we use polyinstantiation along with floating labels, and a new instance of Q denoted as Q^1 is created in the caller's context (i.e. with the caller's label $\{L_P\}$), separate from the original instance of Q with label $\{L_Q\}$. Thus, our approach allows the call to take place, without the mixing of secrecy contexts. The "lazy" aspect of our approach (not represented in the figure) is that we would reuse a previously created instance of Q, denoted Q^{past}, if its label matched the caller's label (i.e., $\{L_P\}$). Additionally, while the new instance has an empty label (i.e., $\{\}$) as the base (compile-time) label in our prototype, our model can be adapted to support a different base label.

Weir uses lazy polyinstantiation for all indirect inter-component calls (e.g., starting an activity, querying a content provider) (described in Section 2.4). *Weir* polyinstantiates processes, Android components and the file system, creating new instances of each for different secrecy contexts. Floating labels allow legacy apps to integrate into *Weir* without modification for making or receiving calls, while polyinstantiation adds context sensitivity. *Weir's* use of floating labels supports process-level labeling along with application multi-tasking (**G3**), a more practical solution than the alternatives of killing existing instances [28, 46] or indefinite blocking [19].

We now describe *Weir's* polyinstantiation of memory, followed by storage. We then discuss how *Weir* supports explicit label changes. Finally, we describe how *Weir's* domain declassification satisfies goal **G4**.

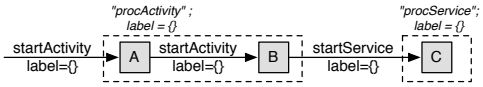
4.2 Polyinstantiation of Memory

To satisfy goal **G1**, *Weir* must ensure that no two component instances with mismatching labels execute in the same process. At the same time, *Weir* must make components available if the underlying Android enforcement (i.e., permission framework) allows. Therefore, our approach polyinstantiates both components and processes to make them available in multiple secrecy contexts.

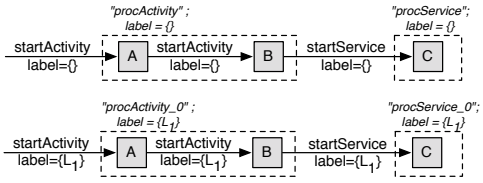
For backwards compatibility, our approach refrains from affecting developer configurations (e.g., by forcing the "multi-process" manifest attribute). Instead, *Weir* polyinstantiates components within the application's own context. Specifically, *Weir* upholds the process assignments made for components by the developer, through the "android:process" manifest attribute (i.e., the component's *processName*). That is, *Weir* ensures that components that were meant to run together (i.e., assigned the same *processName*), still run together . We now describe our approach, followed by an example.

Our approach: On every call, *Weir* retrieves the label of the caller (i.e., the *callerLabel*). *Weir* then checks if an instance of the desired component is running in a process whose label matches *callerLabel*. If one is found, the call is delivered to the matching instance. If not, *Weir* creates a new instance of the called component.

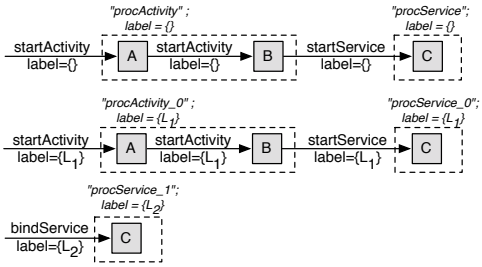
When the target component instance is assigned, *Weir*



(a) Step 1: A is called from an empty label.



(b) Step 2: A is called from a label = $\{L_1\}$.



(c) Step 3: C is called from a label = $\{L_2\}$.

Figure 3: *Weir's* lazy polyinstantiation of three app components; activities A and B, and a service C.

must find a process to execute it. If the process associated with this component (i.e., *processName*) has a different label, *Weir* cannot execute the new instance in it, and has two options: a) assign a polyinstantiated process that is associated with *processName* and has the label *callerLabel* or 2) create a new process associated with the *processName* with label *callerLabel*. As it is evident based on the first option, *Weir* keeps track of a *processName* and all its instances created for various secrecy labels. *Weir* can then reuse a previously instantiated process that is associated with the original *processName* and already has the required label (i.e., *lazy* polyinstantiation). Additionally, *Weir* can ensure adherence to the developer's process assignment; i.e., that component instances only execute in the process associated with their *processName*. If a matching process is not available, *Weir* creates a new process for *callerLabel*, and internally maintains its association with the original *processName*.

Example: Consider an app with three components, activities A and B, and a service C. The developer sets the *processName* for A and B to be "procActivity", whereas the *processName* for C is set to "procService". This means that A and B are expected to run in the same process, while C runs in a separate process. The app is programmed such that when A is started, it starts B, following which B starts C. Using Figure 3, we describe *Weir's* instantiation of A, B, and C and their processes.

In Step 1 (Figure 3a), A is first called by an unlabeled caller; i.e., the *callerLabel* is empty. A new instance of A is created, and a new process by the name "procActivity" is started for it. Then, A calls B. The label of $A's$ process is empty, so B is also instantiated with an empty label, in the matching process, i.e., "procActivity". B then calls C, which is instantiated in the new process "procService".

In the Step 2 (Figure 3b), A is called from a caller with $callerLabel = \{L_1\}$. *Weir* cannot deliver the call to the existing instance of A, as its process has a mismatching label (i.e., $callerLabel = \{L_1\} \neq \{\}$). Thus, *Weir* creates a new instance of A for this call. As there are no processes associated with "procActivity" and with the label $\{L_1\}$, *Weir* also allocates a new process "procActivity_0" to host this instance. Thus, for this call, a new instance of A is started in a new process "procActivity_0", whose label is set to $\{L_1\}$. When this instance of A calls B, the call is treated as a call to B with $callerLabel = \{L_1\}$, the caller being $A's$ new instance with label $\{L_1\}$. As *Weir* keeps records of all the processes created for polyinstantiation, it starts a new instance of B in the process that is associated with B's original process "procActivity", and has a matching label $\{L_1\}$, i.e., "procActivity_0". Reusing an existing process instance is an example of "lazy" polyinstantiation. When this instance of B starts C, *Weir* creates a new instance of C due to mismatching labels, in a new process "procService_0" with label $\{L_1\}$.

In Step 3 (Figure 3c), *bindService* is called on C with the label $callerLabel = \{L_2\}$. Since the caller's label $\{L_2\}$ mismatches with the two existing instances of C that are running with labels $\{\}$ and $\{L_1\}$, a new instance of C is created. As there are no processes associated with "procService" that have a label matching $\{L_2\}$, a new process "procService_1" is created to host the new instance. Note that all of these instances and processes exist simultaneously, as shown in the figures. If C is called again with the label $callerLabel = \{L_2\}$, *Weir* will not have to create a new instance, and the call will be delivered to the existing instance of C running in process "procService_1" with the matching label $\{L_2\}$ (i.e., "lazy" polyinstantiation).

Weir's approach maintains context-based separation in memory (**G1**), and also ensures that components configured to run in the same process still run together; i.e., our approach is transparent to the application, satisfying goal **G3**. For example, instances of A and B exist together, both in the labeled as well as the unlabeled contexts. *Weir* supports all Android components declared in the application manifest, i.e., activities, services, content providers and broadcast receivers. An exception is broadcast receivers registered at runtime, which are instantiated at registration in the secrecy context of the registering process, and hence not subject to further instantiation. Any future broadcasts to such receivers are treated as direct calls subject to strict DIFC label checks.

4.3 Polyinstantiation of Storage

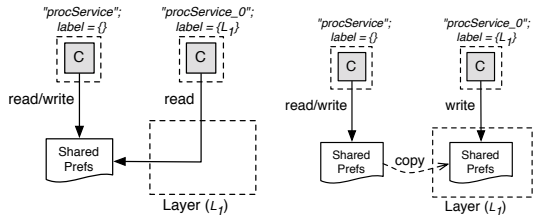
To prevent restrictive labeling of shared storage by processes running in sensitive contexts, *Weir* extends context-based separation to the storage as well (**G2**). *Weir* achieves this separation without denying access to instances in sensitive secrecy contexts (**G3**).

Our approach: *Weir* separates shared state in the internal and external storage using file-system polyinstantiation via a *layered* file system approach [29]. Our approach is similar to Solaris Containers [22], and more recently, Docker [24]. Context-sensitive storage separation has also been used previously in DIFC, for *known* persistent data objects. For example, in their DIFC system for the Chromium Web browser, Bauer et al. create context-specific copies of bookmarks to prevent a restrictive label from making bookmarks unusable [5].

In *Weir*, every secrecy context receives its own copy-on-write file system layer. Processes running in a particular secrecy context have the same view of the file system, which may be different from those running in other contexts. All file operations are performed on the context-specific layer attached to a process, which relays them to the underlying file system (i.e., the default layer). Unlabeled processes are assigned the default layer.

For simplicity, new layers are always created from the default layer, and never from existing labeled layers. That is, for any layer with label L, the copy-on-write always occurs from the default layer (with label $\{\}$), and not another lower layer (say label L_1), even if L_1 is lower than L (i.e., $L_1 \subseteq L$). An alternate design choice of using a non-default lower layer for copy-on-write could lead to conflicts due to incompatible copies of data at different, but similarly labeled lower layers. For example, two labels L_2 and L_3 might be at the same level below L in the DIFC lattice, but may have different copies of the same file. For resolving such conflicts, the system may have to either involve the user or the application. The backwards compatibility and usability effects of choosing a lower layer need further exploration, although it may be a more flexible option. Hence, our design simplifies the potential choice between contending layers by always choosing to copy from the default layer.

For efficiency, a layer only stores the changes made to the default layer by processes in the layer's secrecy context (i.e., copy-on-write). When a file present in the default layer is first written by a process attached to a non-default layer, the file is first copied to the non-default layer and then modified. Future accesses for the file from that context are directed to its own copy. When a process attached to a non-default layer tries to read a file that has never been modified in the calling process's layer, it reads the original file on the default layer. *Weir's* storage approach is an extension of its *lazy* polyinstantiation, i.e., new layers are created only when a process with a previously unknown secrecy context is initialized. Applications transparently access storage using any file system API, and *Weir* directs the accesses to the correct files.



(a) Labeled instance reads. (b) Labeled instance writes.

Figure 4: *Weir's* storage polyinstantiation using layers.

Weir stores the files copied to layers in layer-specific copy-on-write directories. While creating such directories, *Weir* accounts for the security and availability requirements of applications and users. For application-specific internal storage, copy-on-write directories are created in an area that is accessible only to instances of the particular application. For public external storage, *Weir* creates common label-specific copy-on-write directories in an area accessible to all apps. This approach ensures that when an application is uninstalled, its data on external storage is still available to the user.

Example: Figure 4 shows two instances of the component *C*, one of which is running in the unlabeled secrecy context (i.e., label $\{\}$), while the other has a label $\{L_1\}$. *Weir* sets up a file system layer, i.e., *Layer* (L_1), to mediate all file accesses by the labeled instance. *Layer* (L_1) is only attached to processes with label $\{L_1\}$.

As seen in Figure 4a, initially, both the unlabeled and labeled instances of *C* read from the shared preferences file (i.e., *SharedPrefs*). That is, *Layer* (L_1) relays all the read requests by the labeled instance of *C* for unmodified *SharedPrefs* file to the default storage. Once the labeled instance of *C* attempts to write to the *SharedPrefs* file, *Weir* copies it to *Layer* (L_1). This copy is used for all future read or write accesses by instances with label (L_1).

Security of copy-on-write directories: *Weir* ensures the security of the layered directories through a combination of file labeling and Linux permissions. File labels are initialized when first written, to the writing process's label. *Weir* uses strict DIFC label checks for all successive file accesses. Further, *Weir* prevents the implicit flows due to the presence or number (i.e., count) of such copy-on-write directories. To address flows through the presence of specific copy-on-write directories, *Weir* uses random directory names known only to the system. To prevent flows that make use of the number of such directories, *Weir* creates the copy-on-write directories inside a parent directory owned by *Weir*. The Linux permissions of this parent directory are set to deny the read operation on it, and hence cannot be used to list or count subdirectories.

Together with the polyinstantiation in memory, *Weir's* approach enables transparent separation of different secrecy contexts without modifying legacy apps.

4.4 Label Changes and Binder checks

A component instance's label is implicitly set when it is instantiated. Similarly, a file's label is initialized when it is first written to. For all successive accesses (i.e., direct Binder IPC and file reads/writes respectively), *Weir* does not apply floating labels, but performs a strict DIFC label check (i.e., data may not flow to a "lower" label). Hence, any label changes after initialization can only be explicit.

An application aware of *Weir* may change the label of its instances by raising it (to read secret data), or lowering it (to declassify data), provided the change is legal with respect to the policy for the security classes involved in the change, as described in Section 2.3. For example, to read secret data labeled with label $\{L_1\}$, a component instance may raise its label to $\{L_1\}$, if it has authorization (e.g., a capability) from the owner of $\{L_1\}$ (see Section 5.1 for the policy syntax). We now describe the problem caused by explicit label changes.

Problem of explicit label change: A component instance may establish Binder connections with other instances through the Activity Manager, and then use direct Binder RPC. When an instance changes its label, its existing Binder RPC connections (established via indirect communication, see Section 2.4) may be affected. That is, its new label may be higher or lower relative to the instances it is connected to. Hence, it may not be able to send or receive data on existing connections due to the strict DIFC check on Binder transactions. An explicit label change may also make the component instance's context inconsistent with its attached storage layer. At the same time, explicit label changes are unavoidable in applications that use *Weir*.

Our solution: *Weir* provides applications with the *intent labeling* mechanism, i.e., components can label calls (i.e., intent messages), before they are sent to the Activity Manager service, ensuring that the target component is instantiated with the label set on the intent. In fact, a component may instantiate itself with the desired secrecy label by specifying itself as the intent's target. Intent labeling eliminates the need for explicit label changes.

Security of Intent Labeling: *Weir* does not blindly trust the label set on the intent, as applications may otherwise abuse the mechanism for unauthorized declassification. For example, a malicious component with the label $\{L_1\}$ may add secret data to an intent, and set an empty label (i.e., $\{\}$) on the intent before calling itself with it. To account for such malicious use cases, *Weir* checks if the calling application would be authorized to explicitly change its current label to the label on the intent, as per the policy (see Section 5.1). A call with a labeled intent may proceed only if the caller passes the check.

While we have not encountered use cases that cannot be expressed using intent labeling, our implementation allows explicit label changes, mainly for expert developers who may want to make temporary changes to their instance labels. Explicit label changes must be used with caution, as our design does not account for the problems due to label change after instantiation (e.g., dropped Binder calls), since labels do not float to existing Binder connections and files to prevent label explosion (explained in Section 3).

4.5 Domain Declassification

Problems with traditional network declassification are rooted in the decision to declare trust in the exporting subject, as discussed in Section 3. More precisely, in an internet-driven environment, it may be more practical for the data secrecy enforcement to reason about *where* the data is being delivered, rather than *who* is performing the export. *Weir* introduces the alternative of *domain declassification* to allow data owners to articulate trust in terms of the receiver, i.e., the target Web domain. *Weir* allows the data owner to associate a set of network domains (t^D) with its security class (t). When the data in context $\{t\}$ is to be exported to the network, *Weir's* enforcement implicitly declassifies t, if the destination domain is in t^D. The data owner is neither required to explicitly declassify nor trust the exporting application.

In Section 8, we discuss an example where the enterprise only wants data to be exported to a set of enterprise domains, irrespective of the application exporting it. Such a policy allows the user to use the same email application for both the personal and work account, but prevents accidental export of work data to the personal SMTP server. Domain declassification not only addresses the goal of practical declassification in a network driven environment (**G4**), but also prevents the user from accidentally exporting data from a trusted application, but to an untrusted server.

Weir is not the first IFC system to use domains for declassification, although most prior systems to do so consider domains as security principals (e.g., COWL [41], Bauer et al. [5]). For instance, COWL confines JavaScript using a declassification policy analogous to the well-known same origin policy (SOP), i.e., code executes in the context of its origin, and hence possesses the declassification privilege for export to the origin's Web domain. In this case, the origin Web domain is a first class security principal, as it has physical presence on the device in the form of the code running in its context. Thus, in COWL, the declassification privilege is still expressed in terms of the security principal that is sending the data (i.e., the origin). On Android, there is

no direct correlation between Web domains and applications; i.e., Web domains do not have code executing in their context on the device, and hence are not security principals. Thus, *Weir's* approach of expressing trust in the receiver of the data (i.e., the Web domain) rather than the sender is indeed unique among OS-level DIFC systems where Web domains may not be security principals [1, 21, 28, 49]. Hails [15], an IFC web framework for user privacy, may be closer to *Weir's* approach, as it allows users to declassify their data for specific domains. Hails users are prompted to explicitly declassify when network requests to disallowed domains are first made, which may not be feasible on Android (see Section 3).

Weir's enforcement is limited to the device, and may not defeat an adversary controlling the network. While we leave this aspect relaxed for our threat model, we note that DNSSEC or IPsec could be used in such scenarios.

5 Implementation

We implemented *Weir* on Android v5.0.1, and the Android Kernel v3.4. This section describes the essential aspects of our implementation. The source code can be found at `http://wspr.csc.ncsu.edu/weir/`.

5.1 *Weir's* DIFC Policy

Weir derives its policy structure from the Flume DIFC model [21], which consists of *tags* and *labels*. A *data owner* (O) application defines a security class for its sensitive data in the form of a secrecy tag (t). A set of tags forms a secrecy label (S). *Weir* enforces the IFC secrecy guarantee, i.e., "no read up, no write down" [6]. Information can flow from one label to another only if the latter dominates, i.e., is a superset of the former. For instance, data can flow from a process P to a process Q if and only if $S_P \subseteq S_Q$. *Weir* applies this strict DIFC check to direct Binder communication and file accesses.

Each tag t has associated capabilities, namely t^+ (for reading) and t^- (for declassification), which data owners delegate to specific apps, or all other apps (i.e., the global capability set G). At any point of time, a process P has an effective capability set composed of the capabilities delegated to its application (C_P), and the capabilities in G. P can change its label S_P to ${S_P}^+$ by adding a tag t if and only if $t^+ \in C_P \cup G$. Similarly, P can change its label S_P to ${S_P}^-$ by removing a tag t if and only if $t^- \in C_P \cup G$.

As the network interface is untrusted, it has an empty label, i.e. $S_N = \{\}$. Thus, a process P must have an empty label (i.e., $S_P = \{\}$), or the ability to change its label to $S_P = \{\}$ to create a network connection, i.e., $\forall t \in S_P, t^- \in C_P \sqcup G$. Additionally, *Weir* extends Flume's syntax with the domain declassification capability t^D, which is a set of trusted Web domains for tag t specified by the owner O. For a network export to a domain $d \in t^D$, t is implicitly declassified.

5.2 Component Polyinstantiation

When a component calls (i.e., sends an intent message or queries a content provider), the Activity Manager resolves the target component to be called using the static information present in the application manifest. *Weir* does not interfere with this *intent resolution* process. Then, the Activity Manager chooses the actual runtime instance of the resolved component, which is where *Weir's* polyinstantiation takes effect. That is, *Weir* controls component instantiation, without modifying the components themselves. Hence, *Weir* is compatible with all developer manifest options, except ones that control instantiation. Section 7.2 provides a compatibility evaluation for such options. For a detailed explanation on Android's component startup workflow and *Weir's* component instantiation logic, see Appendix A.

5.3 File-system Layering

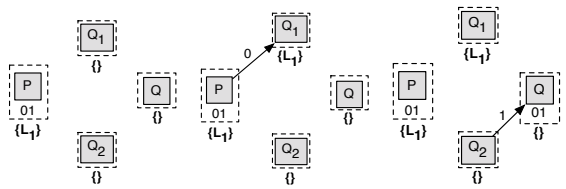
We chose OverlayFS [29] over alternatives (e.g., aufs), as it is in the Linux kernel (since v3.18). As the current OverlayFS patch is incompatible with SELinux, we set SELinux to monitoring mode. This is a temporary limitation, as OverlayFS developers are working towards full integration [45], which is on SELinux's Kernel ToDo list as well [10]. Additionally, we could use a fine-grained block-level copy-on-write file system (e.g., BTRFS [32]). There are advantages to using such file systems, as we describe in the trade-offs (Section 9). Note that while we could get the Android Linux kernel to compile with BTRFS, the build system support tools that are required to build Android's sparse-images for BTRFS (e.g., `ext4_utils` for ext4) are missing. Therefore, our prototype opts for OverlayFS, as it does not require user-space support.

5.4 Process Initialization

On Android, the *zygote* process forks and prepares new processes for applications. When a new process is forked, *Weir* sets its secrecy label in the kernel, and uses zygote to mount the appropriate storage *layer* to the process's mount namespace based on its label. If the process has a non-empty label, *Weir* separates the process's mount namespace from the global mount namespace using the *unshare* system call, and mounts the appropriate OverlayFS copy-on-write layer based on the label on top of the unlabeled file system. New layers are allocated when new labels are first encountered. *Weir* maintains the mapping between a label, its assigned layer and the specific copy-on-write directories used for it.

5.5 Kernel Enforcement

Weir uses a Linux security module (LSM) to track the security contexts of processes and files in the kernel. We



(a) Initial State. (b) P calls Q_1 (c) Q_2 calls Q

Figure 5: Floating label DIFC system: Q receives 1 and guesses 0 for every reply not received.

integrated the multi-LSM patch [35] to enable concurrent SELinux and *Weir* enforcement. The security context of a process contains its secrecy label and capabilities, while that of a file only contains a secrecy label. We now describe the enforcement for file access, Binder communication and network access, as follows:

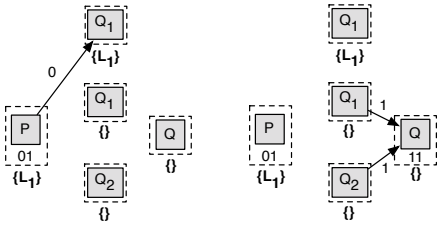
1. Files: *Weir* uses the *file_permission* LSM hook to mediate each file read and write access. The secrecy label of a file (stored in the xattrs) is initialized from the label of the process that first writes it.

2. Binder: *Weir* mediates Binder transactions in the kernel using the Binder LSM hooks. For compatibility, *Weir* whitelists Binder communication with Android system services in the kernel. To prevent apps from misusing whitelisted services as implicit data channels, we manually analyzed all system service API, and modified API that may be misused, e.g., the Clipboard Manager service provides label-specific clipboards in *Weir*.

3. Network: *Weir* mediates the socket *connect* and *bind* operations in the kernel. The tags in the calling process's label that cannot be declassified using its capability set are sent to *Weir's* system service in the userspace via a synchronous upcall, along with the IP address of the destination server. *Weir's* system service then resolves the domain name from the IP address, which is challenging, as a reverse DNS lookup may not always resolve to the same domain used in the initial request. Fortunately, Android proxies all DNS lookups from applications to a separate system daemon. We modify the daemon to notify *Weir* when a process performs a DNS lookup, including the domain name and the IP address returned. During the domain declassification upcall, this mapping is referenced to identify the destination domain. *Weir* allows the connection only if all the tags in the upcall can be declassified for that domain.

6 Security of Polyinstantiation

Floating labels were first predicted to be prone to information leaks by Denning [8]. While language-level floating label IFC models (e.g., COWL [41] and LIO [40,42]) can mitigate such leaks, securely using floating labels is still a challenge for OS-level DIFC systems (e.g., IX [23] and Asbestos [44]). We discuss an attack on an OS-level floating label DIFC system, described in Krohn and Tromer's paper on the non-interference of Flume [20], and show how *Weir* is resistant to such data leaks. We use Android's terminology to describe the attack.



(a) Q_1 is polyinstantiated (b) Unlabeled Q_1 and Q_2 send 1 to Q.

Figure 6: *Weir*: Q always receives data "11"

We describe the attack twice; once in a floating label system without polyinstantiation (Figure 5) and once in *Weir* (Figure 6). Figure 5a shows the malicious components (e.g., services) P, Q, Q_1 and Q_2. P has obtained the data "01", and the accompanying label $\{L_1\}$. P wants to transfer the data to Q, without Q obtaining the label $\{L_1\}$. Note that Q, Q_1 and Q_2 initially have the empty label $\{\}$. Additionally, P and Q have a prior understanding that P will call the i^{th} service of Q to indicate "0" at the i^{th} bit. $Q's$ components are programmed to send a message to Q after a predetermined time if they do not receive a message from P (i.e., indicating a "1"). Since the first data bit is "0", P sends "0" to Q_1, whose label floats to $\{L_1\}$ (Figure 5b). After a predefined time, the component that did not receive a message from P, i.e., Q_2, sends a "1" to Q (Figure 5c). The data leak is successful, as Q knows that the second bit is "1", and assumes the first to be "0", all without acquiring the label $\{L_1\}$. As Android does not place any limits on the number of components, a wider n bit channel is possible with n components.

Weir's polyinstantiation defeats this attack by creating a new instance of Q_1 in a separate process to deliver a call from a label that mismatches its own (Figure 6a). Next, the unlabeled instance of Q_1 and Q_2 both call Q with data "1", as shown in Figure 6a. In fact, for n components of Q, Q will always get n calls with data "1", as *Weir* will polyinstantiate all the components that have been called by P with the label $\{L_1\}$. *Weir's* use of floating labels is resistant to implicit flows inherent to regular floating labels, as labels do not float to the original instance, but to a new instance created in the caller's context.

Jia et al [19] attempt to solve a similar problem,[3] by making the raised label the component's base (i.e., static) label. This defense allows the existing leak, but makes the components that received the message (e.g., Q_1) un-

[3] Refer to page 8 of the paper by Jia et al. [19] for details.

usable for future attacks. Attackers can be expected to beat this defense by coordinating the components used for every attack, and transfering significant data before all the components have restrictive static labels.

Finally, while polyinstantiation is resistant to data leaks in floating labels, we leave the complete formalization of this idea as future work. The intuition behind the formalization is described as follows: Let $\mathscr{L}$ be the type system corresponding to the labels (e.g., type-system for floating labels) and $\mathscr{S}$ be the type system corresponding to information about stacks (e.g., for k-CFA analyis strings of size k that capture information about last k calls). Assume that we have inferencing/propogation rules for both type systems and they are sound. We have an intuition that the combined system (denoted by $\mathscr{L} \times \mathscr{S}$) is sound (the inferencing/propogation rules are basically a combination of both rules).

7 Evaluation

Our evaluation answers the following questions about *Weir's* performance and compatibility:

- **Q1** Is *Weir* compatible with developer preferences that manipulate component instantiation?
- **Q2** What is *Weir's* performance overhead?
- **Q3** Is *Weir* scalable for starting components?

We now provide an overview of the experiments and highlight the results. The rest of this section describes each experiment in detail.

7.1 Experiment Overview and Highlights

Weir does not modify components, but only modifies their instantiation. Thus, we evaluate compatibility with options that control component instantiation (**Q1**), i.e., the singleTop, singleTask and singleInstance activity launch modes described in Section 2.4. We trigger the launch modes in popular Android apps from Google Play, and record application behavior in unlabeled and labeled contexts. We did not observe any crashes or unexpected behavior. Every launch mode worked as expected, while the underlying polyinstantiation ensured delivery of calls to instances in the caller's context.

We measure the performance overhead of *Weir* over an unmodified Android (AOSP) build (**Q2**) with microbenchmarks for common operations (e.g., starting components). Our comparison between the unmodified build, *Weir* (unlabeled instance), and *Weir* (labeled instance) in Table 1 shows negligible overhead. Even in cases where the overhead percentage is large, the absolute overhead value is negligible (<4ms). Further, the negligible difference in the values of *Weir's* labeled and unlabeled instances (i.e., relative to the error) would make a noisy covert channel at best.

As described in Section 5, for every call, Android's intent resolution gets the target component. The OS then chooses a runtime instance from available instances of the target. Hence, the total number of a component's runtime instances only affects its own start time. We evaluate the scalability of a component's start time, when a certain number of its instances already exist (**Q3**). Our results in Figure 7 show a linear increase in the start time with increase in the number of concurrent instances, and low absolute values (e.g., about 56 ms for 100 instances).

7.2 Compatibility with Launch Modes

We randomly pick 30 of the top applications on Google Play (i.e., 10 per launch mode, complete list available at `http://wspr.csc.ncsu.edu/weir/`).

Methodology: For each launch mode, we first launch each application from two separate unlabeled components, and navigate to the specific activity we want to test. With this step, we confirm that the application and specifically the singleTask/Top/Instance activity works as expected. Without closing existing instances, we start the same application from a labeled context and repeat the prior steps. We record any unexpected behavior.

Observations: We did not observe any unexpected behavior, and activities started in their assigned tasks. In the case of singleTask and singleInstance activities, two instances of the same activity ran in the designated task instead of one; i.e., one labeled and the other unlabeled. Intent messages were delivered to the activity instance with the caller's label. This behavior is compatible with singleTask and singleInstance activities, and also maintains label-based separation in memory.

7.3 Microbenchmarks

We evaluate the performance of the operations affected by *Weir* (i.e., file/network access, component/process start) , on a Nexus 5 device. We perform 50 runs of each experiment, waiting 200 ms between runs. Table 1 shows the mean with 95% confidence intervals. Cases with negative overhead can be attributed to the high error in some operations. Specific experimental details are as follows:

Component and Process start: We measure the component start time as the time from the placement of the call (e.g., startActivity) till its delivery. The component is stopped between runs. To measure the process start time, we kill the process between subsequent runs. While the overhead percentages may be high (e.g., for providers), the absolute values are low, and would not be noticeable by a user. Further, the process start time that includes file-system layering in zygote shows minimal overhead.

File access: We perform file read and write operations on

Table 1: Performance - Unmodified Android (AOSP), Weir in unlabeled context, Weir in labeled context.

Operation	AOSP (ms)	Weir (ms)		Overhead (ms)	
		Weir w/o label	Weir w/ label	Weir w/o label	Weir w/ label
Activity start	20.06±4.47	22.22±4.69	20.82±4.87	2.16 (10.77%)	0.76 (3.79%)
Service start	13.94±2.87	14.96±2.85	17.36±4.78	1.02 (7.32%)	3.42 (24.53%)
Broadcast Receiver start	12.92±3.96	11.42±4.44	11.86±3.34	-1.5 (-11.6%)	-1.06 (-8.2%)
Content Provider start	4.54±2.28	7.26±5.32	7.9±4.73	2.72 (59.91%)	3.36 (74.01%)
Process start	127.18±5.62	130.28±5.63	132.98±6.66	3.1 (2.44%)	5.8 (4.56%)
File Read (1MB)	42.38±6.05	43.46±5.44	41.32±5.39	1.08 (2.55%)	-1.06 (-2.5%)
File Write (1MB)	46.8±5.79	47.84±5.42	47.16±5.85	1.04 (2.22%)	0.36 (0.77%)
Network	66.98±3.62	65.68±2.78	69.00±7.04	-1.3 (-1.94%)	2.02 (3.02%)

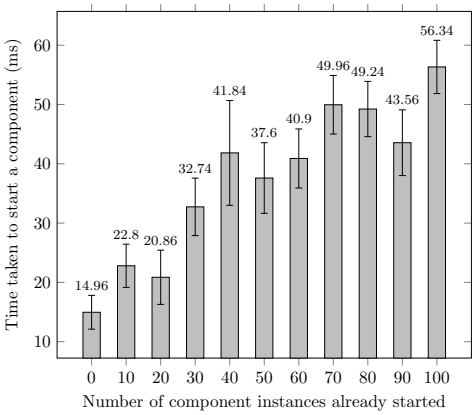


Figure 7: Linear increase in component start time when $0 \rightarrow 100$ instances (in steps of 10) already exist.

a 1MB file using a 8KB buffer. Since the entire check is performed using the process and file labels in the kernel, the overhead value is negligible (e.g., about 0.77% for a labeled file write). We also measure the cost of copying the 1MB file to the labeled layer, i.e., repeating the file write experiment on *Weir* but deleting the file between runs. The extra time taken to copy relative to AOSP is 5.98 ms (about 13% overhead). RedHat's evaluation of OverlayFS further demonstrates its scalability [18].

Network access: We measure the time to establish a network connection using the *HTTPSUrlConnection* API, using domain declassification for the labeled instance. The labeled instance's overhead includes the kernel upcalls and the DNS proxy lookup. The overhead for the labeled instance (2.02 ms or 3.02%) includes the time taken by the DNS proxy to inform *Weir* of the lookup, as well as the synchronous kernel upcalls.

7.4 Scalability of Component Instantiation

We create up to 100 simultaneous instances (in steps of 10) of a service component, each with a different label. At each step, we then invoke the last instance, i.e., from a caller with the last instance's label, and measure the component start time. Note that this experiment presents the worst case scenario; i.e., our prototype does not implement any particular strategy (e.g., least recently used (LRU)) for matching a call with a list of available instances, and a request with the last instance's label will always result in a label comparison with *all* available instances. Figure 7 shows linear scalability, with the highest absolute value being less than 57 ms.

```
// Creating the tag ``t''
domains={``www.bcloud.com'',``smtp.bcloud.com'',...};
createTag(``t'', domains);
```

Listing 1: *BCloud's* policy configuration

```
addTag{``t''};//raise own label to {t}
//perform sharing action ...
removeTag{``t''}//lower own label
```

Listing 2: *BCloud* raises its label

8 Case Study

We investigated the use of labeled enterprise data with an unmodified third party email (*K-9 Mail*) application [9]. With this case study, we demonstrate *Weir's* utility, and motivate the trade-off discussion in Section 9.

Application Setup: We created an enterprise cloud application, *BCloud* that allows the user to sync her work data (e.g., contacts, documents) to the device. Further, we used the popular email application *K-9 Mail* with both user and enterprise data. The setup is as follows:

1. BCloud. We assume that the enterprise policy is to enable the use of third party applications with work data, but to allow export to only enterprise domains. For example, work data must only be emailed using the work SMTP server `smtp.bcloud.com`.[4] Thus, *BCloud* creates a tag t as shown in Listing 1. To set the policy before sharing its data or saving it to storage, *BCloud* may temporarily raise its label to $\{t\}$ (Listing 2), or start itself or other applications with $\{t\}$ using intent labeling (Listing 3). For instance, *BCloud* raises its label before copying the work contacts to Android's *Contacts Provider*.

2. K-9 Mail. We configured *K-9 Mail* for both personal and work email accounts. Like most modern email clients, *K-9 Mail* allows the user to send an email using the work or the personal account, using the *send as* email field. Internally, *K-9 Mail* uses the SMTP server `smtp.gmail.com` for the personal account, and

[4] We used `mail.yahoo.com`, `smtp.mail.yahoo.com` and `imap.mail.yahoo.com` as BCloud's trusted domains.

`smtp.bcloud.com` for the work account. To assist the user in composing an email, *K-9 Mail* retrieves contacts from the *Contacts Provider* app, and makes suggestions as the user types into the "to" (i.e., sender) field.

Experiment: We opened a document from *BCloud* in the *WPS Office* application. Then, from the *WPS Office* app, we shared the document with *K-9 Mail*. *K-9 Mail's* "compose" window was displayed. We then chose to *send as* the work account, and picked a contact to add to the "to" field. We tried to attach another file, and the "attach" action opened Android's system file browser. We selected a file and returned to *K-9 Mail's* compose screen. We then switched to the home screen without sending the work email. We repeated the entire experiment in the default (i.e., unlabeled) context, with the *send as* field set to the personal account. We then sent both emails. Throughout the experiment, we watched the system log for important events (e.g., network denial).

Observations: We made the following observations, and verified them using the system log:
1. Context-specific instances. As we shared work data (in the context $\{t\}$) with *WPS Office* and subsequently *K-9 Email*, instances of these applications (i.e., processes and components) were started in the work context $\{t\}$, and attached to the internal and external (SD card) storage layer $Layer(t)$. The unlabeled context resulted in separate instances with the empty label ($\{\}$), attached to the default storage layer. Instances in both contexts existed concurrently, without any crashes or abnormal behavior.
2. Context-specific data separation. While attaching another document in the work ($\{t\}$) instance of *K-9 Mail*, we could see all the documents on the default storage layer (i.e., unlabeled files), and documents in work $Layer(t)$ (i.e., added from *BCloud*). On the contrary, in the default context, we could only see the files on the default layer. Further, in the default context, *K-9 Mail* suggested from all of the user's unlabeled contacts, but none of the work contacts. In the work context, *K-9 Mail* suggested from all the work contacts, and the unlabeled contacts that existed before *BCloud* synced its labeled contacts. That is, *K-9 Mail* could not see new records created in the default layer's contacts database after it was copied over to $Layer(t)$.
3. Domain Declassification. In the work context, *K-9 Email* was unable to connect to the SMTP and IMAP sub-domains of `gmail.com`, but could only connect with the domains declassified by tag t. Unmodified *K-9 Email* silently handled these network access exceptions, without crashing or displaying errors messages.

9 Trade-offs and Limitations

This section describes the trade-offs of our approach, motivated in part by the observations in the case study.

```
1 Intent intent = new Intent();
2 // Add ''t'' to the intent's label.
3 intent.addToLabel(''t'');
4 // Add data to the intent ...
5 startActivity(intent);//Call self
```

Listing 3: *BCloud* starts itself with new label

1. Centralized perspective: The user cannot view both labeled and unlabeled data together, unless an application is started in the labeled context (e.g., *K-9 Mail* in context $\{t\}$). We envision modified application launchers and phone settings that allow the user to start applications (e.g., File Browsers) with a certain label by default, for making labeled and default data available together. Our test apps use similar techniques; hence such launchers should not be hard to create. On the other hand, a centralized perspective on more than one non-default context (e.g., $\{t1,t2,t3,..\}$) may require a trusted OS application exempt from polyinstantiation (but subject to only floating labels), as floating labels by themselves are vulnerable to information leaks (Section 6).

2. Updates to default layer: While context-specific versions of files may be generally acceptable, in case of database files (e.g., contacts read by K-9 Mail in the work context) the user may expect new records in the unlabeled context to be propagated to the copy in the labeled context. The lack of updates is mainly a trade-off of our file-level copy-on-write implementation (i.e., OverlayFS). As mentioned in Section 5, a block-level copy-on-write file system (e.g., BTRFS [32]) may mitigate this trade-off, as it would only copy the blocks modified by the labeled context, and newly allocated blocks in the default context would be accessible to the labeled context, although this aspect needs further exploration.

3. Access control denials: Floating labels ensure that inter-component communication is never denied, and that resources (e.g., files, other components) are available in all secrecy contexts. Although apps may be denied network access, research has addressed this challenge in the past (e.g., AppFence [17]). Further, most IDEs (e.g., Eclipse) enforce compile-time checks for proper exception handling, and it is uncommon for apps to crash due to network denial, as observed in the case study as well.

4. Instance Explosion: *Weir* creates separate context-specific K-9 Mail instances, *only for the contexts in use*. The theoretical worst-case count of component instances is equivalent to the number of components multiplied by the number of all existing contexts (not just those in use). Our event-based and "lazy" instantiation makes this worst case practically improbable, unlike approaches that execute *all existing* contexts (see Section 10). On the other hand, a denial of service attack on a particular application component may be feasible, by starting a very large number of its instances in a short amount of time for noticeable impact on the lookup time of that compo-

nent. Our implementation can be modified to detect and prevent unusual rates of component instantiation. Note that polyinstantiation of a component only affects its own lookup time (as discussed in Section 7.1), and cannot be used for an attack with a device-wide impact.

5. Resource Overhead: Polyinstantiation may cause resource overhead in terms of the memory, battery and storage. The memory overhead is manageable as Android's out of memory manager automatically reclaims memory from low priority components. Further, any measurement of the battery or storage use is bound to be subjective with respect to the number of labels, number of apps/components, type of apps (e.g., game vs. text editor), aspects of the user scenario (e.g., user-initiated flows, scenario-specific storage access). An objective large-scale study will be explored in the future.

6. Consistency Issues: To a remote server, the instances of an application in Weir are analogous to instances running on different devices (e.g., a user logged in from two devices). Hence, any data consistency issues in such scenarios are not a result of polyinstantiation.

7. Covert Channels: *Weir* mediates overt communication between subjects and objects, but does not address covert channels existing in Android. A clearance label [6, 40, 44] can be used to defend against adversaries using covert channels by preventing access to certain tainted data in the first place. While a clearance label can be easily incorporated into *Weir*, setting the clearance policy for third party applications with unpredictable use cases is hard, and needs further exploration from a policy specification standpoint. Finally, unlike IFC systems that focus on preventing untrusted code within a program from exfiltrating data (e.g., Secure multi-execution [11]), *Weir's* focus is inter-application data sharing. Hence, compartmentalizing an application using clearance is outside the scope of this paper.

8. Explicit labeling of messages and files: On Android, an indirect message through the OS (e.g., intent message) is required before a bi-directional Binder connection can be established between two instances. *Weir* allows floating labels on such indirect communication (but not on direct Binder calls), and polyinstantiation ensures that the two instances at the end of a bidirectional Binder connection have the same label, which is sufficient for synchronous Binder messages. Hence, labeling of individual Binder messages does not provide additional flexibility, unlike in explicit labeling DIFC systems (e.g., COWL [41], Flume [21]). Note that *Weir* allows explicit labeling of indirect messages (i.e., intent labeling). Further, explicit labeling of a file with a label that is different from its creating process instance would place it on an incorrect layer. Such incorrectly stored files will not be visible to future instances started with matching labels, and may cause unpredictable application behavior. Thus, our design trades the flexibility in explicitly labeling files for stable context-sensitive storage.

10 Related Work

In Section 3.1 we described prior DIFC proposals for Android (i.e., Aquifer [28], Jia et al. [19] and Maxoid [46]). We now describe other relevant prior research.

DIFC: Myers and Liskov presented the Decentralized Labeling Model (DLM) [26] that allowed security principals to define their own labels. Since then, numerous DIFC systems have been proposed that provide valuable policy and enforcement models [20, 21, 25–27, 33, 44, 49, 50]. Language-based DIFC approaches (e.g., JFlow [25] and Jif [27]) provide precision within the program, but rely on the OS for DIFC enforcement on OS objects (e.g., processes, files, sockets). On the contrary, coarse-grained OS-level approaches (e.g., HiStar [49] and Asbestos [44]) provide security for flows between OS objects, but cannot reason about flows at the granularity of a programming language variable. While *Weir* is also an OS-level DIFC approach, which means it cannot achieve precision at the program variable level, context sensitive enforcement ensures that *Weir* always has higher precision than traditional OS-level DIFC. Further, while Laminar [30, 33] provides both language-level as well as the OS-level enforcement, it requires applications to be modified to use the precise language-level enforcement. This is not an option for backwards compatible DIFC on Android. Finally, *Weir* does not require general-purpose applications to explicitly define flows as in Laminar, HiStar and Flume [20, 21], as inter-application communication in Android tends to be unpredictable.

Secure multi-execution: Secure multi-execution [11] was proposed to determine and enforce that a program's execution is noninterferent, i.e., to eliminate unlawful data flows by untrusted code *within* a program. The approach achieves noninterference using multiple concurrent executions at all points in the lattice, removing statements that do not match the labels of specific executions. On the contrary, lazy polyinstantiation creates only one instance in the security context of the caller. Unlike secure multi-execution where the multiple executions are treated as a part of the same program instance, polyinstantiation treats multiple executions as unrelated context-specific instances separated in memory and storage. Our approach is more suitable for Android's inter-application data sharing abstractions, while secure multi-execution may be useful to prove non-interference for a general program. Further, secure multi-execution only assumes a finite, predefined label set. This assumption is violated in DIFC systems, where the label set is often

large and not known a priori, and executing all labels at once is impractical.

Faceted Execution: Jeeves [48] and Jaqueline [47] ensure that security principals see different views of data based on their secrecy contexts, using a technique defined as faceted execution. The result of *Weir's* approach is similar; i.e., each security principal can only see data at its own secrecy context. For faceted execution, the copies of data have to be specified by the programmer a priori, which is acceptable if the security of different users using a single program (e.g., a conference submission site) is to be defined. On the contrary, on Android, *Weir's* approach of allowing applications to operate unmodified, and creating context-specific copies on the go, is more practical. To elaborate, data in terms of *Weir* is not the value(s) of a programming language variable, but the instances of components in memory and file system layers per label. Finally, just like secure multi-execution, faceted execution is more suitable when the IFC lattice is small (e.g., two labels) or finite, and may not be feasible for DIFC, where tags can be created at runtime.

Coarse-grained Containers: Approaches such as Samsung Knox [34] and Android for Work [2] protect enterprise data by isolating groups of applications into different containers. Containers cannot compensate for the lack of data secrecy guarantees, as they do not address threats within the container, i.e., the accidental export of secret data by a trusted application or the potential compromise of a trusted application. Virtual phones (e.g., Cells [4]) are similarly inadequate for data secrecy.

Transitive Enforcement on Android: Android permissions lack transitive enforcement, and are susceptible to privilege escalation attacks [7, 16]. IPC Inspection [14] enforces transitivity by reducing the caller's effective permissions to those of the least privileged component in the call chain. Quire [12] provides the call chain information to applications being called, to prevent confused deputy attacks. Like floating labels, privilege reduction is additive, and may severely restrict shared components.

Fine-grained Taint Tracking on Android: TaintDroid [13] detects private data leaks via fine-grained taint tracking on Android, but is vulnerable to implicit flows. CleanOS [43] and Pebbles [38] use fine-grained taint tracking on memory and storage to evict and manage private data respectively. For tracking data in databases, both approaches rely on modification to the database library, which may not be secure as the library executes in the memory of the enforcement subject.

11 Conclusion

Android's component and storage abstractions make secure and practical DIFC enforcement challenging. To address these challenges, we present *lazy polyinstantiation* and *domain declassification*. We design and implement a DIFC system, *Weir*, and show a negligible performance impact as well as compatibility with legacy applications. In doing so, we show how secure and backwards compatible DIFC enforcement can be achieved on Android.

Acknowledgements

This work was supported in part by the NSA Science of Security Lablet at North Carolina State University, NSF CAREER grant CNS-1253346, NSF-SaTC grants CNS-1228782 and CNS-1228620, and the United State Air force and Defense Advanced Research Agency (DARPA) under Contract No. FA8650-15-C-7562. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] ALJURAIDAN, J., FRAGKAKI, E., BAUER, L., JIA, L., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-Time Enforcement of Information Flow Properties on Android. Tech. Rep. CMY-CyLab-12-015, CyLab, Carnegie Mellon University, 2012.

[2] ANDROID. Android for Work. `https://www.android.com/work/`.

[3] ANDROID DEVELOPERS . Tasks and Back Stack. `https://developer.android.com/guide/components/tasks-and-back-stack.html`.

[4] ANDRUS, J., DALL, C., HOF, A. V., LAADAN, O., AND NIEH, J. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 173–187.

[5] BAUER, L., CAI, S., JIA, L., PASSARO, T., STROUCKEN, M., AND TIAN, Y. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)* (Feb 2015).

[6] BELL, D. E., AND LAPADULA, L. J. Secure Computer Systems: Mathematical Foundations. Tech. Rep. MTR-2547, Vol. 1, MITRE Corp., 1973.

[7] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC)* (2010).

[8] DENNING, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM* (1976).

[9] DEVELOPERS, K.-. M. K-9 Mail. `https://github.com/k9mail`, 2015.

[10] DEVELOPERS, S. SELinux Kernel ToDo. `https://github.com/SELinuxProject/selinux/wiki/Kernel-Todo`, 2015.

[11] DEVRIESE, D., AND PIESSENS, F. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy* (May 2010).

[12] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium* (2011).

[13] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[14] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (2011).

[15] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012).

[16] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the ISCO Network and Distributed System Security Symposium* (2012).

[17] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[18] JEREMY EDER. Comprehensive Overview of Storage Scalability in Docker. `https://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/`.

[19] JIA, L., ALJURAIDAN, J., FRAGKAKI, E., BAUER, L., STROUCKEN, M., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract). In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2013).

[20] KROHN, M., AND TROMER, E. Noninterference for a Practical DIFC-Based Operating System. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009).

[21] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (2007).

[22] LAGEMAN, M., AND SOLUTIONS, S. C. Solaris Containers-What They Are and How to Use Them.

[23] MCILROY, M. D., AND REEDS, J. A. Multilevel security in the UNIX tradition. *Software: Practice and Experience* (1992).

[24] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* (2014).

[25] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (1999).

[26] MYERS, A. C., AND LISKOV, B. A Decentralized Model for Information Flow Control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (1997).

[27] MYERS, A. C., AND LISKOV, B. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology* (2000).

[28] NADKARNI, A., AND ENCK, W. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2013).

[29] NEIL BROWN. Overlay Filesystem. `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`.

[30] PORTER, D. E., BOND, M. D., ROY, I., MCKINLEY, K. S., AND WITCHEL, E. Practical Fine-Grained Information Flow Control Using Laminar. *ACM Trans. Program. Lang. Syst.* (Nov. 2014).

[31] REPS, T. W. Program Analysis via Graph Reachability. *Information & Software Technology 40*, 11-12 (1998).

[32] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)* (Aug. 2013).

[33] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2009).

[34] SAMSUNG ELECTRONICS. An Overview of Samsung Knox. `http://www.samsung.com/global/business/business-images/resource/white-paper/2014/02/Samsung_KNOX_whitepaper_June-0-0.pdf`, 2013.

[35] SCHAUFLER, C. LSM: Multiple concurrent LSMs. `https://lkml.org/lkml/2013/7/25/482`, 2013.

[36] SHARIR, M., AND PNUELI, A. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. 1981.

[37] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 1991.

[38] SPAHN, R., BELL, J., LEE, M., BHAMIDIPATI, S., GEAMBASU, R., AND KAISER, G. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of the USENIX Operating Systems Design and Implementation (OSDI)* (2014).

[39] STATISTA. Number of available applications in the Google Play Store from December 2009 to February 2016. `http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`

[40] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell* (2011), Haskell '11.

[41] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014).

[42] STEFAN, H., STEFAN, D., YANG, E. Z., RUSSO, A., AND MITCHELL, J. C. IFC Inside: Retrofitting Languages with Dynamic Information Flow Control. In *Proceedings of the 4th Conference on Principles of Security and Trust (POST 2015)* (2015).

[43] TANG, Y., AMES, P., BHAMIDIPATI, S., BIJLANI, A., GEAMBASU, R., AND SARDA, N. CleanOS: Limiting Mobile Data Exposiure with Idle Eviction. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

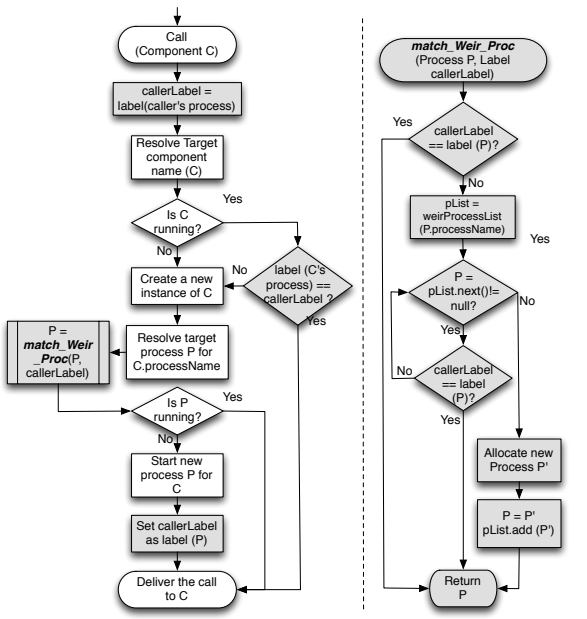


Figure 8: Flow of the Activity Manager starting a component. The areas modified or added by *Weir* are shaded.

[44] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS)* (2007).

[45] WALSH, D. SELinux/OverlayFS integration. `https://twitter.com/rhatdan/status/588338475084029953`, 2015.

[46] XU, Y., AND WITCHEL, E. Maxoid: transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM.

[47] YANG, J., HANCE, T., AUSTIN, T. H., SOLAR-LEZAMA, A., FLANAGAN, C., AND CHONG, S. End-To-End Policy-Agnostic Security for Database-Backed Applications. *arXiv preprint arXiv:1507.03513* (2015).

[48] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2012).

[49] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation* (2006).

[50] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIERES, D. Securing Distributed Systems with Information Flow Control. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2008).

A Component Polyinstantiation Logic

In this section, we describe *Weir's* changes to the Activity Manager service's component and process assignment logic. Figure 8 shows the workflow inside the Activity Manager when a component *C* is called. The shaded blocks form *Weir's* label checks and polyinstantiation logic. Note that the figure portrays the high level steps followed by the Activity Manager, common to all components. When a call arrives, *Weir* first gets the label for the caller's process from the kernel and stores it in *callerLabel*. The Activity Manager then resolves the target component *C* using the information in the call. At this point the Activity Manager only knows the name and type of the target component (e.g., the content provider *C*). The Activity Manager then checks if there is a runtime instance of *C* in its records. If a runtime instance exists and is executing in a process with a matching label, the call is delivered to the running instance. Otherwise, *Weir* forces the Activity Manager to create another runtime instance, for this new *callerLabel*.

Without *Weir*, the Activity Manager would always deliver the call to the existing instance.[5] *Weir* modifies the Activity Manager's internal bookkeeping structures to be consistent with its polyinstantiation; i.e., it enables the Activity Manager to manage multiple runtime records for the same component. For example, the ActivityManager uses a direct mapping between a service's name and its runtime instance, to store records of running services. *Weir* modifies this mapping to one between the name and a set of services.

At this stage, the system has a new component instance that needs to be executed in a process. The Activity Manager selects the process based on the *processName* extracted from the "android:process" manifest attribute. A runtime record of the resolved process *P* is then sent to *Weir* for process matching (i.e., the *Match_Weir_Proc (P, callerLabel)* subroutine). *Weir* first checks the label of the existing process *P*, and if it matches, returns *P* itself. If not, *Weir* retrieves its internal list of processes associated with *P*. This list constitutes the processes that were created in the past to be assigned instead of *P* for specific caller labels. *Weir* checks if the list contains a process with a label matching the current *callerLabel*; this step ensures that components with the same *processName* as well as *callerLabel* are executed in the same process. If *Weir* fails to find a matching process in the list, it allocates a new process for the *callerLabel*, and adds it to the list of existing processes mapped to the specific *processName*. This process is then returned as *P* to the Activity Manager. The Activity Manager then starts *P*, if it is not already started, and *Weir* sets its label in the kernel. Note that if the process is already started (i.e., the original *P* was matching, or a matching process was found in *Weir's pList (P.processName)*, the Activity Manager does not restart it. Finally, the component instance is executed in the assigned process, and the call is delivered to it.

[5] Except in the case of standard and multi-process activities.