DroidDisintegrator: Intra-Application Information Flow Control in Android Apps (extended version)[*]

Roei Schuster
Tel Aviv University
`roeischuster@mail.tau.ac.il`

Eran Tromer
Tel Aviv University
`tromer@cs.tau.ac.il`

April 22, 2016

Abstract

In mobile platforms and their app markets, controlling app permissions and preventing abuse of private information are crucial challenges. Information Flow Control (IFC) is a powerful approach for formalizing and answering user concerns such as: "Does this app send my geolocation to the Internet?" Yet despite intensive research efforts, IFC has not been widely adopted in mainstream programming practice.

We observe that the typical structure of Android apps offers an opportunity for a novel and effective application of IFC. In Android, an app consists of a collection of a few dozen "components", each in charge of some high-level functionality. Most components do not require access to most resources. These components are a natural and effective granularity at which to apply IFC (as opposed to the typical process-level or language-level granularity). By assigning different permission labels to each component, and limiting information flow between components, it is possible to express and enforce IFC constraints. Yet nuances of the Android platform, such as its multitude of discretionary (and somewhat arcane) communication channels, raise challenges in defining and enforcing component boundaries.

We build a system, *DroidDisintegrator*, which demonstrates the viability of component-level IFC for expressing and controlling app behavior. DroidDisintegrator uses dynamic analysis to generate IFC policies for Android apps, repackages apps to embed these policies, and enforces the policies at runtime. We evaluate DroidDisintegrator on dozens of apps.

1 Introduction

1.1 Motivation

The unprecedented connectivity, sensor capability and portability of modern smartphones encourage their use as a primary networking device, entrusted with abundant personal or otherwise sensitive information. Yet users typically run myriad third-party applications ("apps"), authored by unfamiliar or untrusted sources. These apps are typically granted access to the mobile platform's data and sensors. This highlights concerns about privacy breaches: information on the device being

[*]A short version of this paper is to appear in the ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2016 [TS16].

made available to some party contrary to the user's wishes. Integrity of data stored on the device is also at risk, as is the device's behavior (e.g., on authenticated channels to external parties, or by consumption of resources).

In popular mobile platforms (e.g., Android, iOS, and Windows Phone), two main security mechanism are employed to address these concerns. First, apps for these platforms are distributed through a centrally-controlled channel ("app markets") where they undergo a verification process. Second, within the mobile platform, apps are sandboxed at runtime by software and hardware mechanisms, limiting their access to data and system resources. In the following we focus on Android, as a prominent and representative example.

Filtering at app market allows for powerful program analysis techniques to be used offline and ahead of time. In particular, numerous static [CFGW11, LLW$^+$12, GCEC12, FCF09, GZWJ12, ARF$^+$14, MGH15, LBB$^+$15, BJM$^+$15, CFB$^+$15, OLD$^+$15, GKP$^+$, YLL$^+$15, OMJ$^+$13, FAR$^+$13] and dynamic [ZWZJ12, MEK$^+$12, HLN$^+$14, RCE13, EGC$^+$14, GCEC12] analysis techniques have been proposed. The particular analysis methods implemented by the commercial app market curators are not known, though there has been some reverse engineering [OM12, PS12]. The analysis performed by Google's "Bouncer" is for malware-filtering purposes only, and does not provide end-users and reviewers with information about the risk analyzed apps pose, if they are deemed "legitimate". In addition, popular third party markets are not protected by Bouncer [ZWZJ12] .

Another user protection mechanism is the App sandbox, based on permissions granted to the app upon installation. Apps explicitly declare the required permissions. The end-user is prompted to approve some of the permissions during the installation process. Android blocks the app from exercising certain permission-protected APIs if the app doesn't declare the respective permissions.

Still, it is often unclear to end-users, platform maintainers and app reviewers what risk the app actually poses to its users [PXY$^+$13, FGW11, FHE$^+$12]. This depends on how the permission-protected APIs are used by the app. Many enhancements to Android platform security that further constrain apps' behavior have been proposed [ZZJF11, NKZ10, DSP$^+$, FWM$^+$11, BDD$^+$12, SDW12, WHZ$^+$, HNES14, XSA12, SC13, SFE10, NE13, KNK$^+$12] to address this problem. Often, users and app reviewers expect apps to live up to certain restrictions, and it is a common practice among developers to write explanatory text describing the reasons for requesting a permission [PXY$^+$13]. However, these explanations can contain mistakes or falsehoods; moreover they are often absent or hard to find.

Ideally, the Android app permission system would express, as well as enforce, what is stated in these explanations. We address the realization of this ideal with regard to *information flow* within the app. Consider, first, the following motivating examples of real-world apps.

Truecaller. Truecaller is a Caller-ID app. The Google Play description explicitly states: "Truecaller NEVER uploads your phonebook to make it searchable or public". This is, however, not enforced by the app sandbox.

Smart Voice Recorder. The dubious permission to Record Audio (at *any* time) is necessary for any app which uses the phone's Microphone. Smart Voice Recorder records audio at the user's request. It also requests internet access to display ads. There is no reason why the information from the internet should flow into the Record Audio API (e.g., invoke recording of audio by Smart Voice Recorder's web server), but there is no way to enforce the prevention of this flow. This is an integrity issue. There is also no reason why recordings should flow to the internet. This is a privacy issue. Another example given in Appendix A.1.

Android's existing permissions model cannot express the above observations about how apps operate and the risks they do (or do not) pose to users. Moreover, Android's existing enforcement mechanism cannot *enforce* the absence of the aforementioned undesired actions — we merely observe that the app happens to not invoke those actions, but it could easily and silently act otherwise. A more expressive language for specifying app behavior (backed by an enforcement mechanism) would let app developers characterize app behavior more precisely, and let users better judge the potential risk of installing apps.

The inherent problem of the Android permissioning system, observed above, is that information from all sources accessible to the app can flow, at the app code's discretion, to all accessible sinks. Information Flow Control (IFC) is a class of enforcement policies which model entity capabilities as information sources and information sinks, and limit the ability of the governed entities to transfer information from sources to sinks in a more fine-grained way. Thus, applying an IFC technique to Android apps can mitigate the privacy and integrity hazards depicted in the above examples.

A related concern is that of helping users, as well as enterprise IT managers, make informed decisions. Users shouldn't care about, nor guess the implications of, an app having access to their contact data. However they might be alarmed to learn that the app sends their contacts to some web server. Telling users about app information flow brings us closer to informing users about "risks, not resources", as advocated and empirically supported by Felt et al. [FHE+12]. This is evident in the textual descriptions that app developers provide to justify permission use, which often take the form of (informal) declarations of information flow (as for Truecaller, above). The ability to *express* and *enforce* such claims can thus substantially improve security in mobile apps.

1.2 Use Cases and Threat Model

DroidDisintegrator offers a framework for repackaging apps to embed explains app manifests). Our variant of the Android OS can then enforce this policy given the metadata. This mechanism can be deployed in two scenarios (see Figure 1). First, a security-aware developer, concerned about untrusted library code and the possibility of having inadvertently introduced bugs into the code, repackages the app to tighten its running restrictions (possibly after using feedback from DroidDisintegrator to tighten information flow in the app), and then releases the repackaged app. Alternatively, the developer releases the app without repackaging, but an app curator in the app distribution chain (e.g., the app market, or the IT staff in an enterprise), distrusting both the developer's intentions and competence, repackages the app. In both cases, the curators along the app distribution chains, as well as app reviewers (where applicable), can inspect the information flows in the manifest, and base their decisions and recommendations on it.

In this approach, a malicious app wishing to leak information contrary to users' intuition or the textual declarations in its description will not be able to fool curators and reviewers (and subsequently end users), since its policy would have to permit the leaky behavior. We do not identify nor block malicious behavior directly, but by allowing benign apps to declare their (enforced) restricted behavior, we expose the "true nature" of malicious apps which cannot do this and must make transparent their malintent. Conversely, if an app is packaged with a policy that forbids an information leak, then this will be enforced even if the app is buggy, malicious or compromised.

1.3 Component-level IFC

Existing information flow control techniques have not, to date, gained popularity in mainstream apps. Some, such as JIF [Mye99], Fabric[LGV+09], Mobile Fabric [AGL+12] and others [SR03,

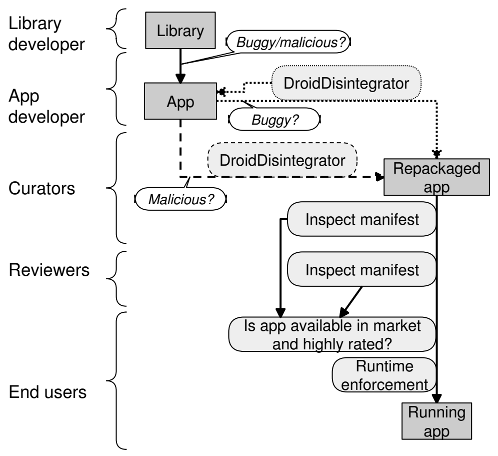


Figure 1: Deployment and threat model. The buggy deployment scenario is marked by dots and the malicious by dashes.

SM03], incorporate alterations in the *programming language* itself, such as variable labeling, to enforce very fine-grained information flow constraints. This approach does not handle native code (common in Android apps). Moreover, it takes a toll on the developer and is hard to incorporate into existing architectures [Zda04]. Ernst et al. [EJM+14] recently adopted, customized, and implemented this approach for Android apps, optimizing on practicality for real-world apps. However, none of these frameworks are (to date) commonly used or supported.

Another approach is to construct policies which attach security labels to processes. By enforcing restrictions on inter-process communication and access to operating-system level resources (files, sockets, etc.) [KYB+07, ZBWKM06, SFE10, SC13], it is possible to guarantee system-level information flow constraints. Some of these solutions are even Android-specific [KNK+12, JAF+]. However, this approach doesn't allow us to reason about information flow within a single-process monolithic app. Therefore we must adopt a different approach.

We observe that mobile app programming in Android offers a very promising intermediate granularity. The Android API defines *components*: functional units that interact with the Android framework and each other. An app is composed of multiple (typically, up to several dozen) components, and the app's execution is, essentially, the invocation of and interaction between its components (with some exceptions; see Section 2.1). Typically and by default, all of the app's components run within a single process.

We hypothesize that by limiting unnecessary inter-component communication and enforcing resource usage at component granularity, it is possible to guarantee IFC constraints within the entire app. This raises the following challenges, which are addressed in this work:

- Defining component boundaries inside an app.
- Defining a policy for limiting inter-component communication and resource usage by individual app components. This policy should guarantee compliance with desired IFC constraints, but still allow operations necessary for the app's "legitimate" behavior.
- Enforcing the policy. This requires the ability to soundly monitor inter-component communication, as well as resource usage in the granularity of a component.

The policy is expressed at component granularity; it allows arbitrary information flow within components, and reasons only about the communication between them.

For policy enforcement, we propose running different components in different processes (when needed) and then leveraging the robust existing mechanisms for process compartmentalization and inter-process IFC. We call this *Application Decomposition.* The Android API supports splitting the app into several processes, each running a different subset of app components [jia]. However, some sets of components are *designed* to communicate with others through process memory [Doca, Docb]; we do not wish to break apps using such patterns. This raises a fourth challenge:

- Learning, *prior* to the enforcement stage in which components are process-separated, which components cannot be separated from others (again, because they are designed to communicate with each other through process memory).

Reasoning about information flow at the level of *components* rather than *variables* or entire *processes* is very natural from the perspective of app design. This places less burden on the developer than other IFC frameworks, in which programs are split into several untrusted subprograms with different privileges, e.g., Jif/split [ZZNM01] and Swift [CLM+07], which require language variable-level annotations, or Hails [GLS+12] and xBook [SBL09], which require a designated runtime and programming environment, and require developers to explicitly declare "components" (in a different sense than Android components). In contrast, the Android component level respects the existing modularity and communication barriers within the apps, which are born out of both the platform's programming methodology and software-engineering practices of programmers. Component permission separation also decreases the amount of code running under each specific permission, rendering apps less vulnerable to confused deputy [Har88] attacks and erroneous API use.

As we show empirically, even in unmodified legacy apps, component-level information flow (properly analyzed) often provides a good approximation of the app's true information-flow behavior and can lead to informative and enforceable policies. In using component-level granularity, our approach is similar to the formal process calculus of Jia et al. [JAF+] (see Section 1.5); we pursue the complementary perspective of implementing an analysis and enforcement framework, and suggest an approach for soundly monitoring inter-component communication inside apps.

1.4 Our Contribution

Use Cases. We suggest a workflow for sound enforcement of component-level IFC. This includes phases of app analysis, policy generation and repackaging of the app's binary (in `.apk` form), and a lightweight runtime component for policy enforcement (an alteration to the Android OS).

Analysis, policy generation and app modification. We implement a framework, DroidDisintegrator, for developers wishing to express information flow constraints in their own apps (in order to increase their appeal to users). DroidDisintegrator learns communication patterns between components, using dynamic analysis techniques (based on and extending the Appsplayground [RCE13] fuzzer and TaintDroid [EGC+14] taint tracker), in order to suggest a suitable mapping of app components into different processes and a suitable IFC enforcement policy. DroidDisintegrator also provides an output that reflects the information flows within the app and thus guides the programmer in identifying changes to the app that will facilitate an even tighter information-flow policy. Finally, DroidDisintegrator repackages the app to encompass both the policy and the component-to-process mapping.

Enforcement. We implement a lightweight reference monitor, within Android, which enforces the component-level policy embedded in the app package.

Applicability to legacy apps. We evaluate the practicality of our solution by using DroidDisintegrator to construct policies for, and repackage, dozens of apps downloaded from Android's app market. DroidDisintegrator automatically analyzed these apps, and in about half the cases found ways to restrict their information flows without breaking their functionality; it then repackaged them to enforce these policies. It reduced the average number of permissions granted to each component to less than a third of the original.

Fail-safe analysis. Precise information-flow analysis is difficult, especially for unannotated legacy code in executable form. DroidDisintegrator can use imprecise analysis (e.g., not including implicit flows) without harming security, because its approach is *fail-safe:* information flows that were not detected in the analysis, and thus reported to the app user as not existing, will not be permitted by the generated and enforced policy. Analysis error can result in either reduced app functionality or an overly permissive advertised policy, but cannot break the security guarantees of the policy. This is fundamentally different than in tools designed for vetting apps by discovering malicious behavior (e.g. FlowDroid [ARF$^+$14] and DroidSafe [GKP$^+$]): while those must detect all maliciously hidden information flows (to rule them out), DroidDisintegrator need only detect the information flow during nominal operation (to preserve app functionality).

1.5 Related Work

On-device enforcement. TISSA [ZZJF11] and Apex [NKZ10] offer fine-grained control over permissions, e.g., per-app user configuration (which can be tweaked at any time), or a "privacy mode" (when at a certain location or time). Android temporarily adopted a similar approach with AppOps, a feature allowing users to dynamically block app permissions, but removed it in 4.4.2 [Ros, XD]. AdSplit [SDW12], AdDroid [PFNW] and Compac [WHZ$^+$] introduced confinement mechanisms to separate advertisement library permissions from those of the hosting app. ASM [HNES14] exposes a callback interface for handling and monitoring system events. Aurasium [XSA12] repackages apps to add policy enforcement code. Both ASM and Aurasium are generic policy enforcement tools that expose flexible interfaces for policy definition.

SEAndroid [SC13, SFE10] enforces mandatory access controls using SELinux at the kernel level, and is now partially integrated into Android mainline. Aquifer [NE13] and Joshi et al. [KNK$^+$12] are mechanisms for inter-app IFC.

Quire [DSP$^+$], IPC Inspection [FWM$^+$11], and XManDroid [BDD$^+$12, BDD$^+$] address the "confused deputy" [Har88] (permission redelegation) attacks. Quire [DSP$^+$] provides authentication code and metadata for data flowing through apps and RPCs, so that the provenance of the data and origin of requests can be verified by the receiving end. IPC Inspection [FWM$^+$11] revokes permissions of apps upon interaction with less-privileged code. XManDroid [BDD$^+$, BDD$^+$12] also monitors inter-app communication and uses a centralized policy to mitigate collusion attacks and confused deputy attacks. These confused deputy mitigation methods can ensure the integrity of data and requests that flow through apps, and prevent these apps from unintentionally fulfilling an untrusted request. The benefits of this partially overlap with those of IFC, e.g., blocking confused deputy attacks that cause a data leak. However, the above do not protect against intentional or unintentional information leaks by apps themselves. Also, in IPC Inspection, permission revocation is at app level, which is coarser grained than component-level and may be over-restrictive.

Static analyzers like Stowaway [FCH$^+$11] and permission-protected API specifications like PScout [AZHL12] and SuSi [ARB13] can be used to learn which permissions are requested by the app and not used at all (they protect APIs not accessed by the app). Removing these from the app's permissions list is a degenerate case of IFC; it prevents information flows from these sources into *any* sink. We focus on enforcing IFC between information sources and sinks which *are* legitimately accessed by some part of the app.

Intra-application IFC. SEDalvik [BBC$^+$13] provides mandatory access control for Java objects in Android. The virtual machine itself monitors the interaction between the objects. This granularity is very fine, resulting in extremely complex policies. No methodology for intra-application IFC policy construction is offered. Moreover, the Java virtual machine in Android (*Dalvik*) is not a sandbox. Malicious apps can run native code and evade SEDalvik altogether. AppFence [HHJ$^+$11] enforces IFC using taint tracking. Taint tracking does not capture implicit flows and can be easily bypassed by an aware malicious developer (DroidDisintegrator also uses taint tracking using TaintDroid [EGC$^+$14], but only for policy generation, and not for enforcement). Jia et al. [JAF$^+$] proposed using component granularity for specifying a DIFC policy, and devised a suitable process calculus. However their proposed intra-application IFC enforcement is "best effort" rather than sound: when the app deviates from conventional Android programming patterns, and even when the app uses some recommended programming patterns [Doca, Docb], there will be undetected and erroneously permitted information flow. Moreover, Jia et al. do not handle callbacks registered by components (see Section 4.2), so a vast amount of code in typical Android apps will go unmonitored.

2 Android Background

2.1 Components in Android

Android apps are composed of components. Each component is of one of the following types: *Activity, Broadcast Receiver, Service or Content Provider.* Each component is declared in the app "manifest" file, along with some of its attributes[1]. The *manifest* is an XML file with explicit declarative specification of some of the app requirements and behaviors. Each component has a specific role to play in the app's functionality. Activities correspond to app "windows" (UI behaviors). Broadcast Receivers are a "mailbox" (each receiver receives and responds to certain messages or broadcast events). Services (not to be confused with system services, see Appendix A.3) represent background operations or "living" objects. Content Providers export content (e.g., table based databases). The behavior of the app is defined as the joint behavior of its components. According to Android Developer documentation, "each one [component] exists as its own entity and plays a specific role — each one is a unique building block that helps define your app's overall behavior". We thus view app components as standalone building blocks. The developer customizes component behavior by extending base classes (e.g., "class Activity") and overriding their methods. These methods serve as callbacks into app code, invoked upon phases in the component life cycle[2]. App component code typically runs within a single process. Components are provided as *Dalvik*

[1]Some components are not declared in the manifest but are registered by the app at runtime, using designated APIs.

[2]It is possible to define entry points to app code that are not a part of a component's life cycle. Examples include class static constructors, and adding app instrumentation code (e.g., for runtime profiling). These and other types of entry points can be modeled as additional (synthetic) components.

Executable (DEX) bytecode, and the app process runs an instance of the *Dalvik Virtual Machine (DVM)* to execute the bytecode.

2.2 ICC and Binder in Android

The Android API supports and encourages communication between components utilizing a series of framework interfaces called *Inter-Component Communication (ICC)*. For example, an Activity (or any other component) can start a new activity or service, send a broadcast message to broadcast receivers, and query content providers. In these cases an Android process (the *System Server*) mediates, and is able to monitor, this interaction. However, it would not be correct to assume that all interaction between components takes place through these interfaces. See Appendix A.5 for more details about ICC.

Android supports RPCs with an elaborate architecture called Binder. The Binder architecture consists of in-kernel code, along with native and Java middleware, which implement *Binder objects*. They can be thought of as system-global objects. To obtain a reference to a Binder object, a process must get its token from the kernel. Instance methods of Binder objects are executed in the process that instantiated the Binder object. Appendix A.3 elaborates on the Binder architecture. We refer to a Binder object instantiated in a process as a *local Binder object* in this process; otherwise, it is a *remote Binder object*.

The ICC API is exposed by the *Activity Manager (AM),* a globally-available Binder object (its token obtainable by any process) which is local in the system server. ICC always crosses app boundaries: to perform an intended action, a component sends an *Intent* to the AM, which resolves the intent's target component (which can handle the action) and sends it the intent[3].

3 Our Approach

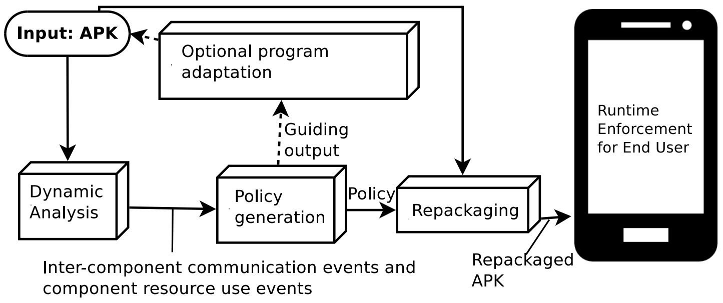
3.1 DroidDisintegrator Workflow

We designed and implemented a full workflow for realizing component-level IFC. It consists of *dynamic analysis* of an app, *generation of an IFC policy* based on the analysis, *repackaging* to embed the policy, and finally, *enforcing* the policy at runtime. In this section we provide a concise overview of the workflow, which is depicted in Figure 2. Section 4 describes DroidDisintegrator, our implementation of this workflow.

Dynamic Analysis. We first collect information about app component interaction in order to construct a policy. The dynamic analysis framework is a device emulator running an Android OS variant designed to monitor three types of events when running apps: app component communication through the process memory, app component communication through other channels, and use of permissions by individual components. An event fuzzer drives the app's behavior in the dynamic analysis framework. Figure 3 depicts the detected events in an actual app.

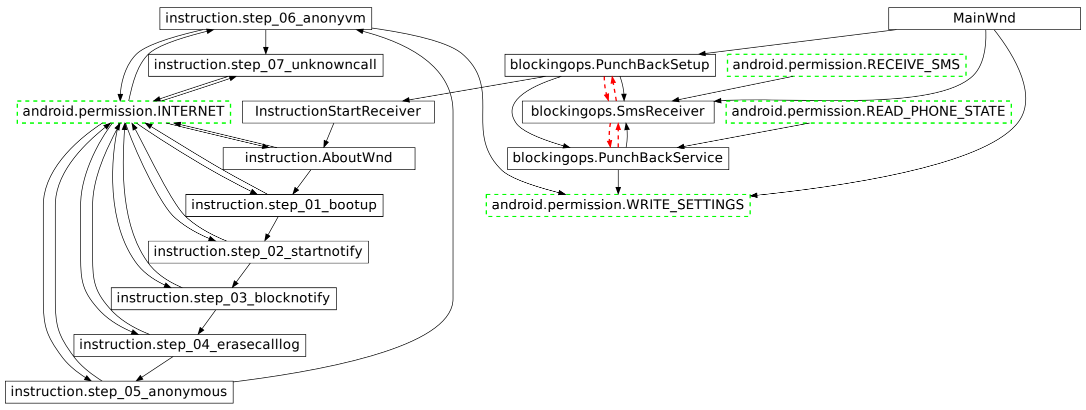
Policy Generation. An app decomposition (see Section 1.3) configuration, as well as an IFC policy, are generated using the information captured during dynamic analysis. The decomposition configuration is a mapping of components into subsets (processes): an equivalence relation is generated

[3]We use the term ICC also when referring to Content Provider APIs, which operate similarly to intents.



Input: APK
Optional program adaptation
Guiding output
Dynamic Analysis
Policy generation
Policy
Repackaging
Runtime Enforcement for End User
Inter-component communication events and component resource use events
Repackaged APK

Figure 2: DroidDisintegrator workflow



instruction.step_06_anonyvm
instruction.step_07_unknowncall
android.permission.INTERNET
InstructionStartReceiver
instruction.AboutWnd
instruction.step_01_bootup
instruction.step_02_startnotify
instruction.step_03_blocknotify
instruction.step_04_erasecalllog
instruction.step_05_anonymous
MainWnd
blockingops.PunchBackSetup
android.permission.RECEIVE_SMS
blockingops.SmsReceiver
android.permission.READ_PHONE_STATE
blockingops.PunchBackService
android.permission.WRITE_SETTINGS

Figure 3: Inter-component communication and component permission use in "GreyThinker", a call blocking utility ("`com.greythinker.punchback`"). Green nodes denote permissions used (sources and sinks). Red edges denote communication through process memory. Black edges denote other communication.

in which two components are equivalent (mapped to the same process) iff it is not possible to assume that some information flows into, or from, one of them and not the other (this is detailed in Section 3.3). A policy contains: (1) Permissions assigned to each component subset (process). (2) Allowed communication directions between component subsets. Such a policy allows us to infer statically which information flows will be possible at runtime (this is detailed in Section 3.4).

The output of the policy generation stage can assist developers in making simple changes that would make the policy more information-flow preventive.

Optional program adaptation. Awareness of the eventual IFC policy allows developers to alter app code to make it conductive to the component-level IFC analysis and enforcement, guided by the output of the policy generation stage.

Repackaging. We change the app's manifest to declare a process for each component, process permissions, and the allowed inter-process communication directions.

Runtime Enforcement. At app installation and runtime on end users' devices, a modified version of the Android OS enforces the IFC policy embedded within the app package. This is done by leveraging process compartmentalization, and adding a lightweight reference monitor. No expensive analysis, such as taint tracking, is done at this stage.

3.2 Discussion: Static vs. Dynamic Analysis

It is natural to consider using static (rather than dynamic) analysis for Inter-Component Communication. Static analysis is less sensitive to platform updates, and does not require driving the app UI, for example manually or by fuzzing. However, state-of-the-art Android static analysis[GKP+, LBB+15, ARF+14] methods explore control flow paths that are unreachable in actual app operation. For example, they consider all sequences of UI event callbacks, including those which cannot occur in the given UI state (the user can only interact with visible Activities). There are also Android-specific difficulties of static app analysis (partially addressed by IccTA[LBB+15]): the asynchronous, user-centric nature of Android apps, and accurately tracking Intent targets. In our workflow, when the above issues result in analysis false positives, it could cause the generated policy to be overly permissive (see Section 4.10). Moreover, static analysis cannot be easily plugged into existing product acceptance tests or unit tests to leverage their app-tailored code exploration, as we would expect developers to do with DroidDisintegrator. For these reasons we chose to use dynamic analysis for our initial implementation.

3.3 App Decomposition Configuration

We wish to map components into processes so that each component resides in one process as described in Section 3.1. The mapping has two goals: not to break app behavior and to reduce the overall number of processes (to reduce unnecessary performance overhead). For the second purpose, we will join components with the same expected privileges into the same process.

Let the relation R_{COMM} be all app component pairs (A, B) s.t. A sends information to B via ICC (sends intent/reads content provider/etc.) or via an IAC channel (Android Properties, file system inode representing a file, pipe, etc.). Let the relation R_{mem} be all app component pairs (A, B) s.t. *information flows* from A to B *through process memory.* We say information flows from$A$ to B through process memory iff A writes data that B reads (for example, A sends B a value using a static class member).

Let R be a relation. We use $R^{+}$to denote its closure and $R^{-1}$to denote its inverse.

Note that if $(A, B), (B, A) \in (R_{\mathsf{comm}})^{+}$, then A and B communicate back and forth. We cannot enforce that one of them has access to less information than the other without breaking app functionality. By segregating them into different processes, we gain nothing.

Note, moreover, that if $(A, B) \in \left(R_{\mathsf{mem}} \cup R_{\mathsf{mem}}^{-1}\right)^{+}$ (one is reachable from the other by traversing R_{mem}), then A and B need to share a memory address space for them to function correctly, and we cannot separate them.

Let $R_{\mathsf{proc}} \equiv R_{\mathsf{comm}} \cup R_{\mathsf{mem}} \cup R_{\mathsf{mem}}^{-1}$, $R_{\mathsf{scc}} \equiv SCC\left(R_{\mathsf{proc}}\right)$ (the Strongly Connected Components DAG of R_{proc}). Each node in R_{scc} corresponds to a subset of the components in the app. Because are no process-memory flows between the subsets, they can be segregated to different processes, and these can be privileged differentially.

The dynamic analysis stage is designed to find R_{mem},R_{comm} and R_{proc} as well as the permissions that each component (and process) requires. R_{scc} can be computed from R_{proc}. The mapping of component nodes in R_{proc} to nodes in R_{scc} corresponds to our desired mapping of components to processes.

3.4 Policies

We wish to construct an IFC policy for inter-component communication and resource use, as defined in Section 3.1.s

Let P_{source},P_{sink} be permissions which protect some APIs. P_{source} protects an information source and P_{sink} protects an information sink. Components, or subsets of components, can access these APIs, in which case we say that they *use* P_{source} or P_{sink} respectively. We then say that P_{sink} is reachable from P_{source} in the relation R if there is a path on the graph spanned by R (where nodes are components or subsets of components) from a node which uses P_{source} to a node which uses P_{sink}. We say that there is a *flow in an app* from P_{source} to P_{sink} if information is transferred from one to the other by the app. We observe that an information flow from P_{source} to P_{sink} exists in the app, *as measured by our dynamic analysis,* only if P_{sink} is reachable from P_{source} in R_{scc}. If such a path does not exist, we wish to enforce the absence of such information flow *in future executions* of the app on the user's device.

DroidDisintegrator thus constructs the following policy: (1) A permission is granted to a process if it is requested by the app and one of the components in the process uses it during the dynamic analysis. (2) Communication between app processes should only be allowed in the direction of edges in the graph spanning R_{scc} (where each node corresponds to a process).

4 DroidDisintegrator

We developed DroidDisintegrator, a set of tools for performing app decomposition as described in Section 3. DroidDisintegrator analyzes and transforms apps to IFC constrained versions of these apps, to be run in an enforcing Android OS. It is an implementation, and automation, of the app decomposition workflow presented in Section 3.1: dynamic analysis (Sections 4.1 through 4.6), policy and app decomposition configuration generation (Section 4.7), repackaging of the application (Section 4.8), and enforcement (Section 4.9).

4.1 Dynamic Analysis Overview

DroidDisintegrator performs dual-faceted tracking of inter-component information flow and tracking of component resource use in apps. Tracking is performed in a modified version of the Android OS, which preserves all Android functionality and augments it with monitoring of app runtime behavior. DroidDisintegrator employs three designated subsystems. The first, *CommTrack*, tracks Android ICC events that cross process boundaries (i.e., events that correspond to R_{comm} edges). The second, *MemTrack*, is based on a dynamic taint analysis infrastructure (TaintDroid [EGC+14]) and tracks cross-component information flows that occur by direct access to the same memory address by multiple components (i.e., events that correspond to R_{mem} edges). The third, *PermTrack,* tracks the use of permissions by individual components. DroidDisintegrator runs the app and logs the events captured by these subsystems.

All three information tracking subsystems assume runtime knowledge of the currently running component. To this end we add a *Component Identity* (CI) register to the Dalvik Virtual Machine (DVM) thread state struct. This register holds a unique identifier of the currently-running component. This requires instrumenting component entry and exit points with register updates, and also tracking callbacks registered by components.

DroidDisintegrator is also integrated with Appsplayground, which contributes automatic app functionality exploration techniques and disguising of the emulator. This allows automatic dynamic analysis of a given app, and makes analysis scalable and parallelizable.

Section 4.2 describes CI register maintenance. Sections 4.3, 4.4 and 4.5 describe the tracking subsystems. Section 4.6 discusses the integration with Appsplayground. Section 4.1.1 discusses an important observation guiding some choices behind the design of CommTrack and MemTrack.

4.1.1 Decomposable Flows

Android ICC facilitates passing Binder objects between components. Components that are in the same process will thus share a regular Java reference to a local Binder object. When components are in different processes, one can have a local Binder object reference and the other a remote reference. The code of the components is oblivious to this distinction, allowing us to separate their processes without loss of functionality. However, in the analysis stage (in which components are typically in the same app process), such ICC events incur process memory flows (R_{mem} edges). In this case we can, by app decomposition, force the information to cross process boundaries (making it possible to monitor) without breaking app behavior. We call these *decomposable* flows.

Another type of decomposable flow occurs when communication via process memory is an API behavior that is transparent to the developer and can be changed in the enforcing version of the OS. For example, the Android Shared Preferences mechanism is a key-value map that is serialized to a file and cached in a static Map variable. Thus, consecutive writes-reads by different components are process memory flows. This mechanism can, however, be managed by a central Binder object in the enforcing OS version. It is thus possible to separate processes of components communicating via shared preferences.

CommTrack and MemTrack are designed to identify and report decomposable flows as R_{comm} edges (by CommTrack) rather than R_{mem} edges (by MemTrack). Another interesting approach (for future work) detecting programming patterns involving process memory flows and automatically modify app code during repackaging to invoke fewer such flows.

4.2 Component Identity

Components in Android, like threads or processes in most operating systems, have entry points and exit points. These define the context in which code execution is performed by the component. Entry points and exit points may be added at runtime, when components register callbacks.

CommTrack, MemTrack and PermTrack rely on the ability to recognize the currently running component throughout the dynamic analysis operation. We maintain this information at the DVM itself, with two extensions: first, a VM register holding the *Component Identity (CI)*, and second, a boolean register, the *Code Flag (CF)* register*,* which indicates whether the currently executing method belongs to a class from the app package (and written by the developer) or from the Android framework, Java library code, etc.

4.2.1 CI Register and object CI

We instrumented the Android OS such that the identity of the component is saved in a designated DVM register on entry and reset to its previous value on exit. The component identity is represented by a 32-bit value uniquely identifying it: the `String::hashCode()` of the component's class name[4]. During interpreting of bytecode that does not belong to any component, the CI register will contain 0.

Static JNI functions are exposed to allow Java code to read and set the CI register content. It is maintained by Android framework code as well as DVM code. The following inductive rules specify how component identity is defined and how the CI register is maintained.

Basis. Every component is implemented, by its author, via a Java class that extends a basic component class (see Section 2.1). Several predefined methods, which are declared by the parent class and which the author can implement in the extending class, are the initial component entry points. The Android framework will call these methods on specific events (e.g., when the user presses the app icon, or when the phone receives an SMS). The CI register is thus updated upon entering and exiting these methods.

Inductive step. Component code can register further callbacks for events at runtime. These callbacks are objects of classes that implement designated interfaces (or extend designated classes) defined by the Android or Java frameworks. When the event occurs, the framework invokes the callback, which is a method of the designated interface (or class). Thus, when component code calls any method and passes an object as a parameter value, the object may contain such a callback method. It is therefore considered as running under this component's identity, and the CI register should be accordingly updated. For example, consider the code in Listing 1: In this code line, an anonymous class implementing the interface `Runnable` is created and an instance is passed to `Thread`'s constructor. The thread is then started and calls the `run()` method of the instance. `run()` is defined as part of a component's code, but is called by a class, that is part of the Java language. During execution of `run()`, the CI register should contain the component identity of the defining component. Such callbacks are very common in Java, and our tracking must handle this case.We therefore mark any object passed to a method by a component. When its methods are entered, the component identity register is switched back to the identity of the passing component.

To reflect this logic, we augmented the DVM to attach a*n object CI* to every Java object. This 32-bit value is initialized to 0 upon object instantiation. The first time component code passes an

[4]Hash collisions are rare but possible. However, they would be detected in the log output, and the analysis could then be re-run with a re-randomized hash.

```
public class ThreadStarter extends BroadcastReceiver {
   @override
   public void onReceive(Context context,
                      Intent intent) {
      Runnable runnable = new Runnable() {
                  void run() {
                        Log.d("anonymousRunnable",
                            "thread␣created");
                  } }
      new Thread(runnable).start();
   }
}
```

Listing 1: Example: callback registration

object as a method parameter value, this object's CI is updated to contain the value in the CI register at the time of the method call. When entering an object's member method, it is the CI register's turn to be updated: if the object CI is not 0, the VM restores the CI register from it. The VM then saves the previous CI register value on the stack and restores it on returning from the method. Once an object's CI contains a nonzero value, it will not be changed again. The object component identity is the identity of the first registering component.

4.2.2 Code Flag Register

Most code running under a component identity is not actually contained in the app package classes. Calls into Java and Android framework libraries still run under the component identity of the calling component. The distinction between component code written by the developer of the app and library code is irrelevant when monitoring high-level ICC or resource use. It is important, however, when using taint analysis for tracking inter-component information flows through the process memory (see Section 4.4).

We thus add another 1-bit flag to the DVM, called the CF. Its value indicates whether the currently running code is a part of the app package (and is part of the implementation of its components) or is part of the Android framework. Upon method entry/exit (and exception unwinding), we update the Framework Code Flag according to the code type (package or framework) of the code pointed at by the instruction pointer. Code type is indicated by the class prefix of its containing method. The prefix is checked against a list of known framework prefixes ("com.android", "java.lang", etc.).

4.3 CommTrack

DroidDisintegrator's CommTracking subsystem monitors and logs all of the intra-app cross-component communication events that correspond with edges in R_{comm}. The subsystem does not track cross-component communication via direct access to shared content in the process address space (which is handled by MemTrack; see Section 4.4). By analyzing the resulting log, we infer the relation R_{comm}. Monitoring falls into the following categories.

Predefined ICC events. A component invokes an operation (e.g., sends an Intent) that invokes one of another component's entry points. To keep track of sender identity in Intent-based ICC, we instrumented the Intent class to contain a *sender component* field, updated by the (instrumented) Intent-sending API implementation. The identity of the intent's sender is checked before entering the invoked component, and the information flow is logged. We similarly instrument activity results and log flows when they are read. We instrument the ContentProvider API, which differs slightly in that it is not Intent based, to log directed information flows on content queries, deletes, insertions, and so forth. We also capture component runtime registration, a form of ICC in itself.

Communication via Binder. As explained in Section 2.2, when a component sends a Binder call to another component in its own process, the event should be recorded as an R_{comm} event. The local Binder object's object CI field will typically contain the identity of the component that instantiated it, and upon the call the identity register will be updated to this identity. In this case, we capture an R_{comm} flow from the calling component to the callee[5].

Other Communication. We deal similarly with other channels and readable/writable resources: filesystem, Shared Preferences and Content Providers. If a component writes to such a resource, which another component later reads from, this is considered a flow between the writing component and the reading one. We intercept and log file system inodes reads/writes, as well as reads/writes to content providers and Android's *SharedPrefs* mechanism (see Section 4.1.1). CommTrack captures the read/write accesses, and an analysis of the log produces the flow events. Similarly, possible flows through the Internet and other information sources that are also sinks are detected by the later analysis (but use of permissions is captured by PermTrack).

4.4 MemTrack

DroidDisintegrator's MemTrack module, which is based on TaintDroid [EGC+14], monitors and logs cross-component communication through process memory (i.e., events that correspond to edges in R_{mem}). We built, on top of TaintDroid's taint propagation, our own tagging of data elements. Our aim is to keep track of reads and writes of data so that if one component writes data that another reads, this event will be logged. Consequently, MemTrack differs somewhat from traditional tainting schemes: data access by component code is a taint sink (for reads) and source (for writes).

Tainting and Taint Storage. Taint is tracked via *taint tags*, which are 32-bit bitmasks. Each bitmask bit is mapped to some component identity. To minimize collisions, only components which actually run within the app are mapped (at runtime) from some available bit; this happens when the CI register is first updated to contain their identifying value. Apps can have more than 32 components, but we found that they very rarely execute more than 32 components at runtime within the same process. (It is straightforward to extend MemTrack's bitmask size, or add a level of indirection, to support that case.)

Taint Granularity. TaintDroid alters DVM's runtime stack and structures underlying Java objects to allocate a 32-bit taint tag for each data element. A tag is maintained for every method local variable, method argument, class static field, class instance field, and array. We used TaintDroid's taint granularity and changed only the tag value semantics.

The one exception is array handling. TaintDroid maintains one taint tag per array. We empiricallyfound that this incurs many falsely reported process memory flows.[6] Assigning a separate

[5]When the called method's object is not a Binder object at all, this mechanism can falsely report flows. However, these falsely reported flows rarely affect the policy construction, because when the object is not a Binder object, an R_{mem} flow between the corresponding components will almost always be reported as well as the R_{comm} one (see Section 4.4).

[6]This stems from the Android framework's use of arrays to hold some objects that become tainted by many components, although different components often only access disjoint array elements. For example, there exists an array of objects representing a UI View object of every Activity in the process. A UI view's reference will typically be

tag for every array element incurred such a large overhead that the analysis became too slow to be practical[7]. We currently regard process memory flows through arrays as false positives, as they most likely are. Note that (as discussed in Section 4.10) this can only result in overly restrictive policies, which can be easily discovered by an additional testing phase, and does not compromise the soundness of the enforcement.

Taint Sources, Sinks and Taint Propagation. When a component writes to a data element, the VM turns on the bit representing that component in the data's taint tag. When a component A reads a data element D, the taint tag of D is checked by the VM. If it is tainted by components other than A, then we identify (and log) information flows from those components into A. Whenever data is manipulated by bytecode, data taint is propagated. This means that every data access by Java code may be a taint source, a taint sink, require taint propagation or a combination of these. Therefore tainting and logging of flows are carried out by the DVM itself, and their implementation is interleaved with taint propagation.

Taint propagation, tainting and logging occur in the DVM opcode implementation and JNI calls. We deliberately turn off TaintDroid's taint propagation abilities through file system nodes and Binder RPCs, since we monitor only flows of data that do not cross process boundaries (inter-component information flows through these interfaces are monitored by CommTrack).

We classify data access by the DVM opcodes and JNI methods into three types[8]: reading a value (possibly written by another component), writing a value (possibly other components will read), and moving an aggregation (n-ary operator result) of values into a data location (here we read each of them and write into one taint location). We add handlers for these events and instrument the DVM to call them, passing the taint tags. Appendix A.2 specifies how each of these access types is handled.

The handling logic for these data access events changes according to the event type and whether the VM is running component code (which is a taint source and sink) or other code (such as framework libraries, which only propagate taint). This is reflected in the CI and CF register values.

To avoid reporting decomposable flows (see Section 4.1.1) as R_{mem} edges, we nullify taint tags of shared preferencesvalues (by instrumenting the `SharedPreferences` implementation) and local Binder object method arguments[9].

4.5 PermTrack

The DroidDisintegrator *PermTrack* subsystem monitors permission use by individual components. All permission checks occur within the method `checkUidPermission` of the *Package Manager* (PM) system service or the method `checkPermission` of the Activity Manager. Both are Binder methods and can be invoked remotely or locally. An operation typically invokes one of these methods if it wishes to check the permission of the third party requesting the operation, usually also through a Binder RPC. We capture calls to these methods, and when a permission is granted we deduce that

tainted with the respective activity's component identity. The entire array quickly becomes tainted with all Activities' tags although every Activity only accesses its own view.

[7]Pebbles [SBL+14] and Spandex [CGL+14] proposed using lazy taint tagging for arrays. This optimization should be incorporated into DroidDisintegrator as well.

[8]TaintDroid contains JNI method prototype annotations, which specify how information is expected to propagate through the method, between the method arguments and return value.

[9]Thus, we instrumented the DVM to recognize calls to local Binder objects. We do not recursively remove the taints of arguments, as this can cause missed process memory flows.

the component requesting the calling operation *used* the permission (this also requires maintaining information about the calling componentfor every Binder RPC call).

We instrument the socket API to record Internet permission use. Accesses to log files, the SD card, bluetooth and camera devices are recorded as respective permission uses.

4.6 Automated Dynamic Analysis

Appsplayground is a tool for scalable automatic dynamic analysis of Android apps. It employs a Java app that connects to an emulator running a modified version of the OS and governs app behavior exploration logic. Appsplayground was designed to trigger, among other things, information flows from sources to sinks in apps. It features intelligent event triggering for exhaustive exploration of app functionality and disguise techniques that make running inside the emulator less discernible from running inside a physical device. We take advantage of these features by merging Appsplayground with our Android variant.

4.7 Policy Generation

After driving the app in the dynamic analysis emulator, DroidDisintegrator uses information captured by CommTrack, MemTrack and PermTrack to output a graph depicting R_{mem}, R_{comm}, as well as component permission use (see Figure 3 and Figure 5). DroidDisintegrator uses a database with our manual classification (source or sink or both) for each permission. If a new, unclassified permission is encountered, the DroidDisintegrator user is prompted to classify it at this stage. A mapping of components to processes is deduced, as described in Section 3.3 (by computing $R_{\mathsf{proc}} \equiv R_{\mathsf{comm}} \cup R_{\mathsf{mem}} \cup R_{\mathsf{mem}}^{-1}$, and then $R_{\mathsf{scc}} \equiv SCC\left(R_{\mathsf{proc}}\right)$). A policy is generated, as described in Section 3.4.

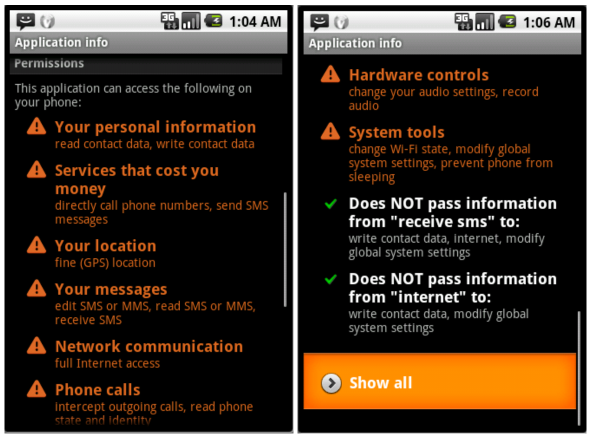
A bipartite graph depicting the source-to-sink information flows in the app is output (depicting the policy's security guarantees). Additionally, the R_{proc} graph (see Figure 5 and Figure 3) is output. This graph provides information about the behavior of the application to the DroidDisintegrator user. These can help a developer identify possible changes in app structure and component interaction so that a tighter policy can be enforced. DroidDisintegrator can thus help a motivated app developer to express security assurances with a structured and enforceable language.

4.8 Repackaging

We unpack the app binary (a signed zip file with the`.apk` extension, or an *APK*), and edit the app manifest. A "`process`" attribute is added to each component tag in the manifest, specifying under which process it runs. Further tags are added to the manifest, encoding allowed inter-process communication and process permissions. We then repackage the app into an APK.

4.9 Enforcement

Package installation. During app installation of a repackaged app on our modified Android platform, when users are prompted to approve app permissions, they are also informed about information flows guaranteed *not* to occur in the app (see Figure 4). These simple-to-understand guarantees are derived from the app's embedded policy, and end-users are not burdened with implementation details about decomposition, components and processes.


1:04 AM
Application info
Permissions
This application can access the following on your phone:
⚠ **Your personal information**
read contact data, write contact data
⚠ **Services that cost you money**
directly call phone numbers, send SMS messages
⚠ **Your location**
fine (GPS) location
⚠ **Your messages**
edit SMS or MMS, read SMS or MMS, receive SMS
⚠ **Network communication**
full Internet access
⚠ **Phone calls**
intercept outgoing calls, read phone state and identity

1:06 AM
Application info
⚠ **Hardware controls**
change your audio settings, record audio
⚠ **System tools**
change Wi-Fi state, modify global system settings, prevent phone from sleeping
✓ **Does NOT pass information from "receive sms" to:**
write contact data, internet, modify global system settings
✓ **Does NOT pass information from "internet" to:**
write contact data, modify global system settings
Show all

Figure 4: Scrolling down the app permission view, presented on app installation for user approval and accessible also from the "Application Management" menu.

App runtime. To run components in different processes, we utilize the optional and rarely used (standard Android-API) "process" attribute, added to component declarations in the repackaged app's manifest. This allows running a component or a set of components in a separate, uniquely named, process. To enforce the IFC policy, we modify Android's ActivityManager service to monitor all cross-process ICC and resource use at process granularity (rather than app granularity). The monitor allows or denies events according to the policy in the app's `Manifest.xml` file, and based on the operation initiator's process UID as well as its PID (identifying the process within the app). Operations requiring permissions are allowed if the policy grants the process this permission. Components are protected with an extra permission to send ICC to their containing process, granted to the appropriate processes. The monitor does not, itself, handle information flow between the processes via Linux system calls; this challenge is well-studied and addressed in prior systems such as Flume [KYB$^+$07], ASM's Aquifer hooks [HNES14, NE13], and others [XSA12, BDD$^+$12]. Thus, we have reduced the hard intra-app IFC problem to the well-studied inter-process IFC problem.

4.10 Implications of Analysis Error

As previously discussed, the dynamic analysis guiding the policy generation is imperfect and relies on some heuristics for handling corner cases or critical performance issues. It is essential to understand their implications. Missed flows (false negatives) can result in overly-restrictive generated policies, possibly breaking app functionality. Reported flows that do not really occur (false positives) can result in too-permissive generated policies, which the app curators/reviewers/users may not approve. But neither break security guarantees: if a policy is approved, the information flow it specifies will be enforced.

Thus, we err on the side of caution: impaired functionality (especially on the relatively coarse whole-component level) can be discovered with manual or automated testing of the app, e.g., by re-running the app analysis and acceptance tests after the decomposition; unsound policy enforcement would have been much less noticeable and harder to analyze.

Notably, we only aim at capturing flows which are forms of inter-component communication intended by the developer or important for the app's legitimate operation. In particular, tracking implicit flows would be of very little benefit to us: they're rarely used for intentional inter-component communication, and most techniques for tracking them would simply incur a vast number of false positives [KHHJ08], making our generated policies trivially permissive. Not tracking implicit flows does not make DroidDisintegrator more vulnerable, but rather the opposite: if an app developer uses implicit flows to smuggle information to circumvent app vetting techniques, DroidDisintegrator will not report this flow in the analysis stage, and simply prevent it in the enforcement stage.

5 Empirical Results

We evaluated DroidDisintegrator on third-party apps from the Android app market, as follows.

We used a snapshot of the Android app market from 2011 (for compatibility with Appsplayground), and chose the 100 apps which use the largest number of permissions. These had 20–48 permissions each, typical for popular apps (e.g., the average number of permissions requested across the 10 most popular free non-game apps as of 11.2014 was $\sim$34). We ran DroidDisintegrator on these apps. In 84 of the cases, our fuzzer successfully finished the experiment. Some of the remaining 16 apps crashed multiple times, suggesting problems more serious than unwanted information flows; in a few cases the fuzzer itself malfunctioned. For these 84 apps, we generated policies and repackaged them.

Most (43) of the successfully fuzzed apps utilized more than 5 permissions, as detected by fuzzing. DroidDisintegrator identified and enforced preventable information flows in 20 of these apps ($\sim$46%). We re-ran these apps under the fuzzer after repackaging, and there was no change in app behavior as far as fuzzing could tell. We manually and heuristically operated them, and did not observe any broken functionality either. The remaining cases were detected to have information flows from all sources to all sinks used; in these cases DroidDisintegrator's policy still revoked the permissions which aren't used at all by the app.

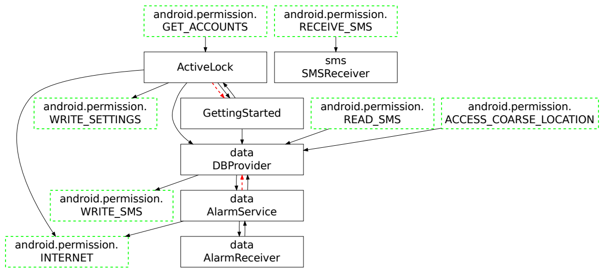
The remaining (41) successfully-fuzzed apps utilized very few permissions (5 or less), as detected by fuzzing, so there were few potential information flows to block. Nonetheless, DroidDisintegrator identified and enforced preventable information flows in 6 of them.

The apps used 15.2 permissions on average, and after decomposition each component process only used 4.2 permissions on average. Thus, we greatly reduced decomposed app permissions (to less than a third of their original number).

The most commonly observed type of a preventable flow is from the `RECEIVE_SMS` source to various sinks (internet, storage, etc.). Inspecting these apps, we observe that the entry points and control flow that handle incoming SMSs indeed have little interaction with the rest of the app. That means the app could (under Android's normal permission semantics) forward users' SMS messages to rogue parties, but actually do not exploit that capability; DroidDisintegrator enforces this behavior and conveys it to the user. This is demonstrated in Figure 5.

Policy Learning Performance. Generaring the policy, including fuzzing and dynamic analysis, took 43min (averaged over 100 apps), running inside a Ubuntu 11.10 VM with 4 cores and 8 GB of RAM, on an Intel Core i7-3720QM 2.6GHz CPU, 32GB of RAM. This is trivially parallelizable, and seems quite practical for developers, app stores, or CISOs.

Enforcement Performance. The enforcement itself, which is performed on the decomposed version of apps, has negligible overhead (merely checking membership of a permission in a set, in a code path that already contains heavyweight RPC). The app decomposition does add an overhead, since it increases the number of app processes (repackaged apps define, on average, 3.8 processes, instead of the typical single process), and causes Android to use serialization for some ICC that originally



android.permission. GET_ACCOUNTS
android.permission. RECEIVE_SMS
ActiveLock
sms SMSReceiver
android.permission. WRITE_SETTINGS
GettingStarted
android.permission. READ_SMS
android.permission. ACCESS_COARSE_LOCATION
data DBProvider
android.permission. WRITE_SMS
data AlarmService
android.permission. INTERNET
data AlarmReceiver

Figure 5: Inter-component communication and component resource use graph for “Executive Assist” (“`com.appventive.ActiveLock`”), a productivity utility. See Figure 3 for notation.

passed its arguments as pointers. Systematic evaluation of these overheads is difficult, due to the interactive behavior of apps, but we provide the following anecdotal evidence that the overhead is low.

We performed quantitative comparisons between two environments: one is an emulator with 10 decomposed apps installed (the *decomposed environment*), and the other is an emulator with the original (non-decomposed) packages installed (*non-decomposed environment*). In terms of process count: when exercising 3 of the installed apps in sequence under the fuzzer, we observe an average over time of 51.1 active processes in the decomposed environment, versus 50.0 in the non-decomposed environment. Device and app responsiveness: 8 of the installed apps are registered to the “boot complete” broadcast event, and 6 of them are registered to the “SMS received” broadcast events. Handling of these events is sequential: one app must finish before the other begins. Averaged over 10 reboots, the handling time for the “boot complete” event (time from event dispatch until the last app finishes) is 309 seconds in both environments. Averaged over 100 received SMSs, the handling time for the “SMS received” event was 0.57 seconds for the non-decomposed environment and 0.61 seconds for the decomposed environment (a 6.5% increase).

6 Conclusions and Future Directions

In this work, we showed how to constrain the behavior and reduce the risk of Android apps, based on a key observation that the modular component-centered design of apps offers a natural granularity at which to apply Information Flow Control. In our implemented workflow, apps are first dynamically analyzed to deduce a policy about their internal information flows, and this policy is embedded in the app installation package. After installation, the device enforces the policy. We implemented the analysis tools required by this process, building and improving upon prior research efforts. We ran our analysis on real-world apps and produced useful enforcement policies for many of them, reducing both the number of possible information flows within the app and the privileges under which each component runs.

Our approach and findings open myriad research directions for extensions to new applications and platforms.

Fuzzing. Policy generation uses dynamic analysis, which relies on exploration of the app’s behavior using fuzzing. Code coverage and performance may be improved using state-of-the-art fuzzing mechanisms such as PUMA [HLN$^+$14] and Brahmastra [BHJ$^+$14], once adapted to triggering information flows.

Taint tracking. Better heuristics and taint-tracking would reduce falsely-reported flows and detect decomposable flows (see Section 4.1.1), and thus tighten the generated policies. Likewise, static analysis can be used. Our system is based on the Android 2.1 branch of TaintDroid, and adapting it to the latest TaintDroid will improve app compatibility and taint accuracy. Analysis support for Android's ART runtime is desirable(and probably feasible [Gro]).

Beyond Android. Our approach may be applicable to other platforms, especially modern mobile device platforms that use event-driven frameworks. While Android's explicit "component" abstraction is especially convenient, units of similar granularity may defined by the programmer or synthesized by analyzing data flow from the app's entry points.

DIFC. Our approach can be extended to *Decentralized* Information Flow Control, allowing discretionary access controls and finer-grained, application-dependent labeling of components and data [KYB$^+$07, KNK$^+$12, JAF$^+$].

Side channels and covert channels. As in most works on information flow control and mandatory access controls, our system does not address the risk of covert platform channels [Hu92], or side channels such as cache attacks [OST06, Per05]. We note that covert exploitation of such channels is harder in the context of curated app stores.

Enhanced Enforcement Capabilities. It is natural to consider building an Android Security Module (ASM [HNES14]) to enhance DroidDisintegrator's enforcement abilities. Alternatively, Aurasium [XSA12] can be used to embed policy enforcement in the app bundle itself.

Android M Dynamic Permissions. In upcoming Android Version 6 (Marshmallow), some app permissions are granted at runtime and on-demand. Component process separation can be extended to leverage such a dynamic security label model, where dangerous information *flows* are reported at runtime and approved by the user on-demand.

Declassifiers. Declassifiers are a powerful feature in IFC, allowing otherwise forbidden information flows to occur under specific, explicit, and carefully-reviewed conditions. For example, the constraint that "information must not flow from the contact list into outgoing SMS messages" is desirable but often too restrictive (e.g., it might forbid the application from auto-completing a contact's phone number when the user composes an SMS). This may be relaxed into the constraint "contact information flowing into outgoing SMS messages must be approved by the user before sending". The relaxation requires a piece of trusted code (e.g., an Activity) that presents to the user whatever the application wants to send in an SMS and (upon user approval) invokes the SMS-sending API. The declassifier's simple, trusted code can be explicitly designated as such, and bundled in source form with the app (along with requisite evidence of consistency with the binary) for inspection by users and app curators.

Acknowledgments. This work was supported by the Check Point Institute for Information Security; by a Google Faculty Research Award; by the Israeli Ministry of Science and Technology; by the Israeli Centers of Research Excellence I-CORE program (center 4/11); and by the Leona M. & Harry B. Helmsley Charitable Trust.

References

[AGL$^+$12] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing mobile code securely with information flow control. In *IEEE Sympo-*

sium on Security and Privacy 2012. IEEE, 2012.

[ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. SuSi: A tool for the fully automated classification and categorization of Android sources and sinks. Technical Report TUD-CS-2013-0114, EC SPRIDE, 2013.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Programming Language Design and Implementation (PLDI) 2014*. ACM, 2014.

[AZHL12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the Android permission specification. In *ACM Conference on Computer and Communications Security (CCS) 2012*. ACM, 2012.

[BBC+13] Aline Bousquet, Jérémy Briffaut, Laurent Clévy, Christian Toinard, Benjamin Venelle, et al. Mandatory access control for the android dalvik virtual machine. In *ESOS 2013*, 2013.

[BDD+] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*.

[BDD+12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Network and Distributed System Security Symposium (NDSS) 2012*, 2012.

[BHJ+14] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *USENIX Security Symposium 2014*. USENIX Association, 2014.

[BJM+15] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2015) 2015*, 2015.

[CFB+15] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Network and Distributed System Security Symposium (NDSS) 2015*, 2015.

[CFGW11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *International Conference on Mobile systems, applications, and services (MobiSys) 2011*. ACM, 2011.

[CGL+14] Landon P Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for android. In *USENIX Security Symposium 2014*, 2014.

[CLM+07] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM SIGOPS Operating Systems Review*, volume 41. ACM, 2007.

[Doca] Android Developers' Documentation. How do I pass data between Activities/Services within a Single Application? `http://developer.android.com/guide/faq/framework.html#3`.

[Docb] Android Developers' Documentation. Local service sample. `http://developer.android.com/reference/android/app/Service.html#LocalServiceSample`.

[DSP$^+$] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium 2011.*

[EGC$^+$14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3), 2014.

[EJM$^+$14] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *ACM Conference on Computer and Communications Security (CCS) 2014*. ACM, 2014.

[FAR$^+$13] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for android applications. *EC SPRIDE, TU Darmstadt, Tech. Rep*, 2013.

[FCF09] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. SCanDroid: Automated security certification of Android applications. *Technical Report, University of Maryland*, 2009.

[FCH$^+$11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security (CCS) 2011.* ACM, 2011.

[FGW11] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *USENIX Conference on Web Application Development (WebApps).* USENIX Association, 2011.

[FHE$^+$12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Symposium on Usable Privacy and Security (SOUPS) 2012.* ACM, 2012.

[FWM$^+$11] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium 2011*, 2011.

[GCEC12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Trust and Trustworthy Computing (TRUST) 2012*, volume 7344. Springer, 2012.

[GKP$^+$] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in DroidSafe. In *Network and Distributed System Security Symposium (NDSS) 2015.*

[GLS$^+$12] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Symposium on Operating Systems Design and Implementation (OSDI) 2012*, 2012.

[Gro] TaintDroid Google Group. Taintdroid for libart. `https://groups.google.com/forum/#!topic/taintdroid/WbcrccvjaKg`.

[GZWJ12] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Network and Distributed System Security Symposium (NDSS) 2012*, 2012.

[Har88] Norm Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), 1988.

[HHJ$^+$11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security (CCS) 2011*. ACM, 2011.

[HLN$^+$14] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. PUMA: Programmable UI-automation for large scale dynamic analysis of mobile apps. In *International Conference on Mobile systems, applications, and services (MobiSys) 2012*. ACM, 2014.

[HNES14] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security Symposium 2014*. USENIX Association, 2014.

[Hu92] W-M Hu. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy 1992*. IEEE, 1992.

[JAF$^+$] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on Android. In *ESORICS 2013*. Springer.

[jia] Stack Overflow jiangian.lily. Usage of android:process. `http://stackoverflow.com/questions/7142921/usage-of-androidprocess`.

[KHHJ08] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *International Conference on Information Systems Security (ICISS) 2008*. Springer, 2008.

[KNK$^+$12] Palanivel Kodeswaran, Vikrant Nandakumar, Shalini Kapoor, Pavan Kamaraju, Anupam Joshi, and Sougata Mukherjea. Securing enterprise data on smartphones using run time information flow control. In *IEEE International Conference on Mobile Data Management (MDM) 2012*. IEEE, 2012.

[KYB$^+$07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[LBB$^+$15] Li Li, Alexandre Bartel, Tegawendé François D Assise Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and

Patrick McDaniel. IccTA: detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*, 2015.

[LGV+09] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *ACM Symposium on Operating Systems Principles (SOSP) 2009*. ACM, 2009.

[LLW+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting Android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS) 2012*. ACM, 2012.

[MEK+12] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *International Workshop on Automation of Software Test (AST) 2012*. IEEE, 2012.

[MGH15] Martin Mohr, Jürgen Graf, and Martin Hecker. Jodroid: Adding android support to a static information flow control tool. In *Proceedings of the 8th Working Conference on Programming Languages*, 2015.

[Mye99] Andrew C Myers. JFlow: Practical mostly-static information flow control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) 1999*. ACM, 1999.

[NE13] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *ACM Conference on Computer and Communications Security (CCS) 2013*. ACM, 2013.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2010*. ACM, 2010.

[OLD+15] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[OM12] J Oberheide and C Miller. Dissecting the Android bouncer. *SummerCon2012, New York*, 2012.

[OMJ+13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium 2013*, 2013.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conference Cryptographers' Track (CT-RSA) 2006*. Springer, 2006.

[Per05] Colin Percival. Cache Missing for Fun and Profit. *BSDCan*, 2005.

[PFNW] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in Android. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2012*. ACM.

[PS12] NJ Percoco and S Schulte. Adventures in BouncerLand: Failures of automated malware detection within mobile application markets. *Black Hat USA 2012*, 2012.

[PXY$^+$13] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium 2013*, 2013.

[RCE13] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: automatic security analysis of smartphone applications. In *ACM conference on Data and Application Security and Privacy (CODASPY) 2013*. ACM, 2013.

[Ros] Seth Rosenblatt. Why Android won't be getting App Ops anytime soon. `http://www.cnet.com/news/why-android-wont-be-getting-app-ops-anytime-soon/`.

[SBL09] Kapil Singh, Sumeer Bhola, and Wenke Lee. xBook: Redesigning privacy control in social networking platforms. In *USENIX Security Symposium 2009*. USENIX Association, 2009.

[SBL$^+$14] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: fine-grained data management abstractions for modern operating systems. In *Symposium on Operating Systems Design and Implementation (OSDI) 2014*. USENIX Association, 2014.

[SC13] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS) 2013*, 2013.

[SDW12] Shashi Shekhar, Michael Dietz, and Dan S Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium 2012*, 2012.

[SFE10] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security & Privacy*, 8(3), 2010.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1), 2003.

[SR03] Vincent Simonet and Inria Rocquencourt. Flow Caml in a nutshell. In *Proceedings of the First APPSEM-II Workshop*. Nottingham, United Kingdom, 2003.

[TS16] Eran Tromer and Roei Schuster. DroidDisintegrator: intra-application information flow control in Android apps. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2016*. ACM, 2016.

[WHZ$^+$] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in Android. In *ACM conference on Data and Application Security and Privacy (CODASPY) 2014*. ACM.

[XD] XDA-Developers. Dianne hackborn explains appops removal. `http://www.xda-developers.com/source-code-commits-in-android-4-4-2-kot49h-reveal-flash-sms-attack-fix-and-app-ops-removal/`.

[XSA12] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security Symposium 2012*, 2012.

[YLL+15] Wei You, Bin Liang, Jingzhe Li, Wenchang Shi, and Xiangyu Zhang. Android implicit information flow demystified. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Symposium on Operating Systems Design and Implementation (OSDI) 2006*. USENIX Association, 2006.

[Zda04] Steve Zdancewic. Challenges for information-flow security. In *International Workshop on the Programming Language Interference and Dependence (PLID) 2004*, 2004.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Network and Distributed System Security Symposium (NDSS) 2012*, 2012.

[ZZJF11] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming information-stealing smartphone applications (on Android). In *Trust and Trustworthy Computing (TRUST) 2011*. Springer, 2011.

[ZZNM01] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *ACM SIGOPS Operating Systems Review*, volume 35. ACM, 2001.

A Further details

A.1 A Motivating Example: SMSPopup

We describe our manual information-flow analysis of the open-source app SMSPopup. The app features customizable pop-up dialogs for displaying incoming SMS and MMS messages, and requests several permissions. We model these permissions as information sources and sinks as done in previous Android works [ARB13, AZHL12, FCH+11]. The app thus requests the privilege (and will be able at runtime) to leak information from every such source to every such sink. However, our manual inspection indicates that SMSPopup does not actually use all of its information flow capabilities: there is no need, in the frame of the app's operation, to transfer information from every source to every sink. Figure 6 shows the information flows allowed by the app's requested permissions, vs. the actual information flows exhibited by the app.

In particular, note that the device permissions allow the app to send contact list information via outgoing SMS messages, which would be a privacy issue — but the app never invokes this information flow. The device permissions allow the device's logs to interfere with SMS sending APIs through the app, which would be an integrity issue — but the app never invokes this information flow.

A.2 MemTrack Data Access Handling Logic

As explained in Section A.2, data accesses by the DVM, in opcodes or by JNI calls, are classified into three categories: Read a labeled element, Write a labeled element, and Move an n-ary operand

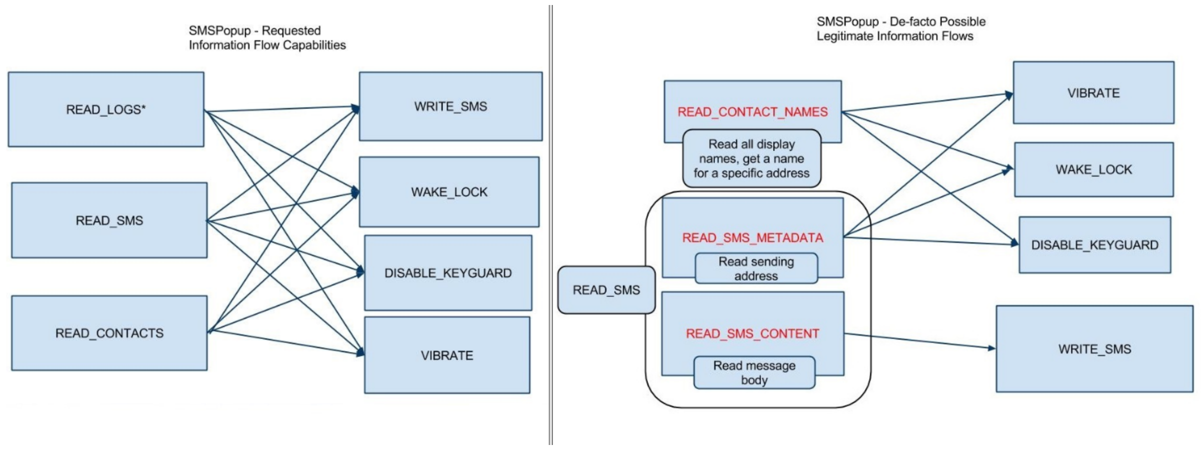


Figure 6: SMSPopup — Requested vs. Used Information Flow Capabilities

result (for which the arguments are labeled data elements) into some labeled element. The handling logic for these events is detailed in Listing 2.

A.3 Android Binder

Understanding our analysis framework and its implementation requires understanding how processes and components actually communicate in Android, using a fairly complicated architecture called "Binder". We therefore provide a brief overview of the Binder architecture.

The Android Binder architecture consists of a driver, along with native and Java middleware, which implement *Binder objects*. Binder objects support passing their reference between processes in a mechanism transparent to the programmer. They can be thought of as system-global objects. Instance methods of Binder objects are executed in the process which instantiated the Binder object.

A Binder object reference is not necessarily globally available. Interestingly, two app processes can share a new reference to a global object (instantiated within one of them) only through RPCs, as described below. This implies that the two processes must have previously shared a reference to some common Binder object. For this reason, there are a few well-known Binder objects, which we'll refer to as *system services* (not to be confused with a *Service* component, described in Appendix A.5). These are typically created within the System Server process during system startup. After initiation they are explicitly registered with the "*Service Manager*" under a specific name — this is done using a designated system call into the Binder driver. Any process can attain a proxy reference to such a well-known named service, by initiating another designated system call to the Binder driver.

To define a global object, a programmer has to define an interface for the remote object, the *remote interface*, and separately, an implementation. The interface has to extend the `IInterface` interface.

The interface methods can only receive and return objects of specific types. Some of the basic types — primitive types, `String`, `CharSequence`, `List`, `Map` — may be passed to and from a remote method. Moreover, any user-defined `Parcelable` object (see below) can also be specified in method prototypes as an argument/return value. Additionally, other remote interfaces may be specified — which means Binder objects (implementing these interfaces) can also be passed to and from RPC methods.

```
/* This is the "code flag" (explained in Section 5.1.2).
   False when in framework code, true in application code. */
boolean code_flag

/* DVM register containing the identity of the current component
   (or NULL if there is none) */
component_identity current_component_identity

// TAINT LOGIC:

read(Taint taint)
{
  if (code_flag) {
     for identity in componentIdentitiesInBitmask(taint)
       logFlowBetweenComponents(identity, current_component_identity)
  }
}

write(Taint taint)
{
  if (code_flag) {
    taint.setBits(
      taintBitForComponentIdentity(current_component_identity))
  }
  else {
    taint.setBits(0) // Values written by framework are not tainted
  }
}

move(Taint taint_from_1, Taint taint_from_2, ...,
       Taint taint_from_n, Taint taint_to)
{
  if (code_flag) {
     read(taint_from_1)
     read(taint_from_2)
     ...
     read(taint_from_n)
     taint_to.setBits(
       taint_from_1.getBits() |
       taint_from_2.getBits() |
       ... |
       taint_from_n.getBits() |
       taintBitForComponentIdentity(current_component_identity)
       )
   }
   else { // Standard taint propogation
     taint_to.setBits(
       taint_from_1.getBits() |
       taint_from_2.getBits() |
                ... |
       taint_from_n.getBits()
     )
   }
}
```

Listing 2: Propogation primitives pseudocode.

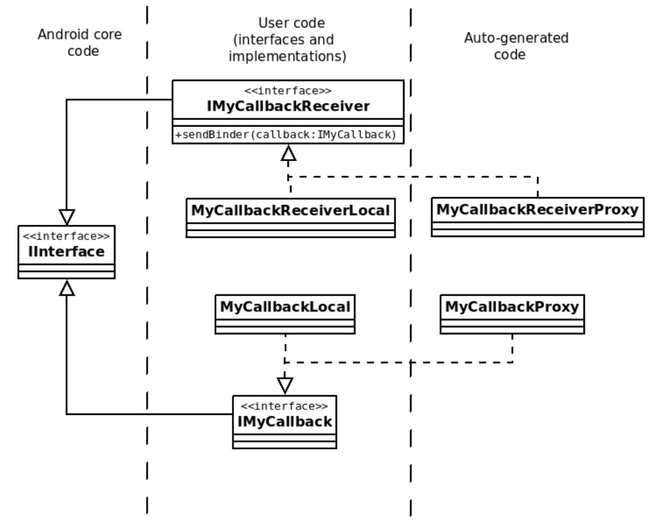


Figure 7: Binder object class hierarchy (simplified)

For the basic types (primitive types, String, etc.) — the Binder framework contains methods that flatten objects of these types into buffers, and methods that unflatten the buffers back into Java values. This enables passing such an object as a buffer between two processes through the kernel (through the Binder driver).

`Parcelable` is an interface similar to Java's `Serializable`. It contains a `toParcel()` method and a (static) `createFromParcel(byte_array buffer)` method. This facilitates flattening `Parcelable` objects into buffers to be passed between processes similarly to more basic types.

The case of passing Binder objects between processes is more complicated. It is, however, a crucial part of the architecture, because it is the main facility that is used to actually share Binder objects between different processes. We therefore explain it in Section A.4.

A.4 Binder Object Propagation

A key to understanding how Binder objects work is understanding how a reference to such global objects is created and shared between processes.

Binder objects implement some (app-specific) remote interface, which itself extends the `IInterface` interface. As mentioned in Section A.3, the remote interface method prototype can specify arguments or return values that implement the `IInterface` interface. Thus, Binder objects (which always implement it) can be passed around between processes as arguments to methods of remote objects.

Figure 7 is a (simplified) example of a class hierarchy, for the two example remote interfaces `IMyCallbackReceiver` and `IMyCallback`. Recall that the user of the Binder framework defines only the interface and its implementation.

Figure 8 depicts the process of instantiating a Binder object (implementing `IMyCallback`) in process A, and passing a reference to Process B. Process B can then invoke an RPC of `IMyCallback`, to be executed in Process A.

Within Process A, the Binder object reference for the `IMyCallback`-implementing object is simply a reference to the instance of the implementing class. We call this reference a *local reference.*

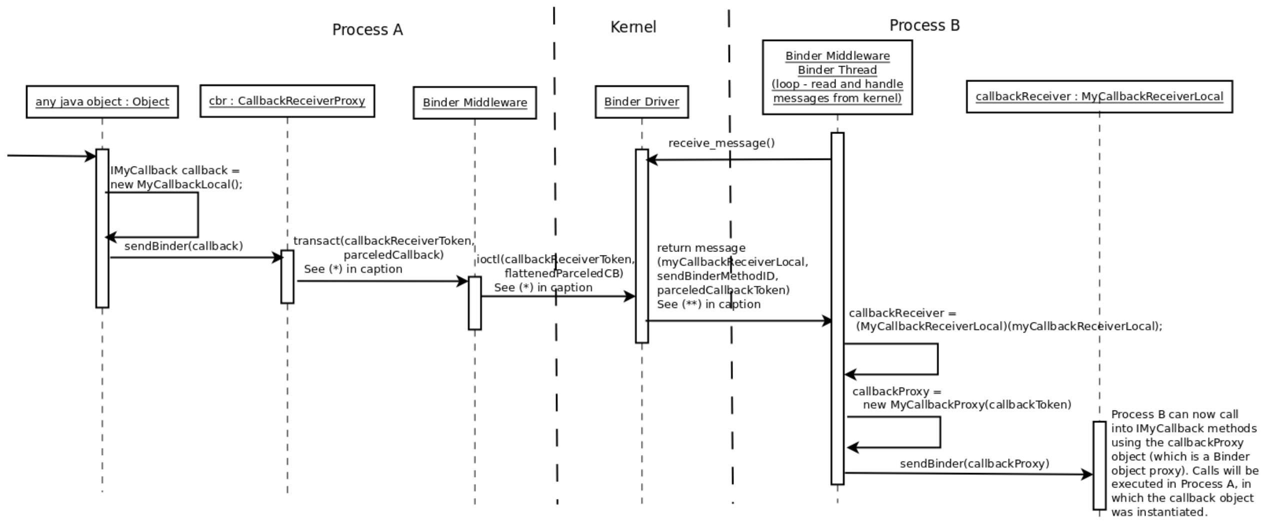

Process A
Kernel
Process B
any java object : Object
cbr : CallbackReceiverProxy
Binder Middleware
Binder Driver
Binder Middleware Binder Thread (loop - read and handle messages from kernel)
callbackReceiver : MyCallbackReceiverLocal
receive_message()
IMyCallback callback = new MyCallbackLocal();
sendBinder(callback)
transact(callbackReceiverToken, parceledCallback) See (*) in caption
ioctl(callbackReceiverToken, flattenedParceledCB) See (*) in caption
return message (myCallbackReceiverLocal, sendBinderMethodID, parceledCallbackToken) See (**) in caption
callbackReceiver = (MyCallbackReceiverLocal)(myCallbackReceiverLocal);
callbackProxy = new MyCallbackProxy(callbackToken)
sendBinder(callbackProxy)
Process B can now call into IMyCallback methods using the callbackProxy object (which is a Binder object proxy). Calls will be executed in Process A, in which the callback object was instantiated.

Figure 8: Binder object propagation. (*) `parceledCallback` is a list of RPC arguments. Here, they only contain one argument: a pointer to `callback`. (**) `parceledCallbackToken` is a buffer of flattened `Parcelable` objects which are the method arguments. Here, it's just a token to `callback`.

So, to create a Binder object and attain a local reference — Process A simply has to instantiate an object implementing this class (i.e., `MyCallbackLocal`).

Assuming that Process A has a reference to a Binder object residing in Process B, called a *proxy reference*, it can pass any local reference as an argument to this remote object's methods if doing so adheres to the object's remote interface.

`IMyCallbackReceiver` does define a method that receives an `IMyCallback` object as an input argument. We assume Process B holds a local reference to an `IMyCallbackReceiver` object, and Process A has a Remote Reference to this object. Therefore, Process A can initiate an RPC into its `IMyCallbackReceiver` reference, which will send Process B a reference to the Binder object implementing `IMyCallback`.

In Process B, like in all processes that use the Binder infrastructure, there is at least one *Binder thread*. This thread reads messages from the Binder driver and calls into message-handling code. In our scenario of an RPC from Process A to Process B, the Binder thread in Process B receives from the kernel a message containing a pointer to the `IMyCallbackReceiver` object, a method identifier and the parceled arguments — including the proxy reference to the Binder object. The Binder middleware casts the pointer into an `IMyCallbackReceiver`, extracts the arguments from the parcel, and calls the method indicated by the method identifier.

Under the hood, a proxy reference is actually a handle to a unique token identifying a Binder object. This handle is received from the kernel, and is wrapped in a class implementing the remote interface. The implementation of each interface method actually invokes a system call that passes the handle, method identifier and parceled arguments to the in-kernel Binder driver (see step 3 of Figure 8). The method identifier and parceled arguments are sent to the process which instantiated the Binder object.

All of this is transparent; the user of the object does not need to be aware of RPCs underlying proxy's interface methods.

A.5 Android Inter-Component Communication (ICC)

Components communicate with each other using the system API for "Inter-Component Communication". We use the term *ICC* to denote ICC via Android system services, not to be confused with Binder IPC or RPC. Component code always runs within a process designated for the app. However, components exist globally and ICC crosses app boundaries. In fact, every startup of the app process is a result of cross-app-boundaries ICC with some other app (except for some apps launched by the system on boot). The other app (sender) in such scenarios will typically (but not necessarily) be one of the built-in Android OS apps. For example, the "Launcher" app sends a "Launch" event to the activity corresponding with the icon pressed by the user. The other app can also be just another app installed on the device, e.g., a social network app can broadcast a global event once a notification is received — Broadcast Receivers are subscribed to such events. This provides flexibility for developers.

ICC is mediated by the `ActivityManagerService` (*AM*), which exposes an interface for component interaction. When an app wishes to use a component (i.e., to use some service, access some information, send some message, pop a UI window), it will invoke one of the AM Binder object's methods to request this. Ideally, the request should *only* specify an indication of *what* to do, which of the four component types is expected to implement this behavior, and the *arguments* necessary to do it (in other words, calls into system services specify policy rather than mechanism).

This specification is usually called an *Intent*, and class Intent is designed such that its instances represent such a request. It is up to the system service to resolve which component is most suitable to perform the request, instantiate (if necessary) a process for the app containing this component, and finally send the designated Binder object within this process an indication of exactly which component entry-point to run. For example:

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_VIEW, Uri.parse("tel:123456"));
startActivity(intent);
```

In this case, the call into startActivity indicates to the AM that the calling code intends for an activity to "display the phone number 123456". Typically, an Activity from the phone's Dialer app will pop up. It is possible for other apps to register for such intents as well. Intent resolution in these cases generally has intricate rules, some of which have been discussed in previous work [CFGW11].

Content Providers (which expose databases on the device) are not accessed via an Intent object. Instead of an Intent, a URI of the accessed content is passed (via the API) to the `ActivityManagerService`, which resolves content provider targets.