DroidCap: OS Support for Capability-based Permissions in Android

Abdallah Dawoud
CISPA Helmholtz Center for Information Security
abdallah.dawoud@cispa.saarland

Sven Bugiel
CISPA Helmholtz Center for Information Security
bugiel@cispa.saarland

***Abstract*—We present DROIDCAP, a retrofitting of Android's central Binder IPC mechanism to change the way how permissions are being represented and managed in the system. In DROIDCAP, permissions are per-process Binder object-capabilities. DROIDCAP's design removes Android's UID-based ambient authority and allows the delegation of capabilities between processes to create least-privileged protection domains efficiently. With DROIDCAP, we show that object-capabilities as underlying access control model integrates naturally and backward-compatible into Android's stock permission model and application management. Thus, our Binder capabilities provide app developers with a new path to gradually adopting app compartmentalization, which we showcase at two favorite examples from the literature, privilege separated advertisement libraries and least privileged app components.**

I. INTRODUCTION

Android, like other mobile platforms, employs an access control model in which applications (apps) have to request privileges—permissions in Android's jargon—to access user data and system resources. Once granted, those permissions are assigned to app sandboxes, defined by each apps' UID in the system. Thus, the permissions associated with the UID of each process constrain the process' access to (system) services, other apps, or file-system objects.

This design decision to attribute permissions to app sandboxes at the level of UIDs combines characteristics from capability-based access control systems (i.e., the attributes of the subject and not its identity matter) with those of list-based access control models (i.e., the UID forms an ambient authority for all processes executing under the UID). The existence of an ambient authority on Android has been shown problematic for the users' privacy when app developers mix app code with non-trustworthy code from other origins, where particularly advertisement libraries have repeatedly exhibited privacy-intrusive behavior [25], [12], [21], [62], [61] that exploits the fact that the library code inherits all privileges from its host app's UID, i.e., its ambient authority. Moreover, permissions enforced by the Linux kernel through Linux GIDs, such as internet or Bluetooth access [6], are statically assigned at app install-time to the app UID and are hence particularly hard to manage in a flexible manner that allows easy delegation or revocation. We argue that this combination of characteristics that retains an ambient authority generally impedes efficiently separating privileges on Android and makes it unnecessarily hard for developers to efficiently create new, least-privileged protection domains and adopting app compartmentalization best-practices. For instance, every component of an app inherits its app's full permission set and also prior works on privilege separation [52], [59], [71], [29], [18] necessarily have to set up new UIDs with separate permissions. There exist some works that refine the authority: for instance, SEAndroid's type enforcement [60] assigns security contexts to processes; different solutions on information flow control (e.g., [45], [46], [70]) assign security labels to processes; few solutions make app components the principal for permission enforcement [64], [57]; and app virtualization [9], [11] can enforce policies per-process on IPC and syscalls. However, the status quo is amiss with the *practical* needs for *efficiently* enabling privilege separation and providing developers with a path for least-privilege code compartmentalization, as commonly adopted by security-critical apps [54], [35]. Per-process privileges alone are not sufficient, but also efficiently creating new protection domains through delegation and revocation of privileges is key and is currently not sufficiently supported (see also Section VI).

In this paper, we propose object-capabilities as a way to achieve per-process permissions together with the efficient delegation of privileges between processes. Drawing from past and current experiences on object-capability systems, we shift Android's permission model closer to an object-capability system. Capability-based access control has been historically around [19], [23], [68], [37], [38], [49]—where it found use in high-assurance and distributed systems, such as EROS [58], IBM System/38 [28], iMAX 432 [33], CAP [48], or Amoeba [43]—and has recently been proposed for modern end-user systems, such as a new security feature for conventional systems like UNIX/Linux [65], [20], hybrid systems like CHERI [66], or new microkernel-based systems like Google's Fuchsia [24]. Our solution adds to this recent developments. We show that an object-capability system not only fits well to Android's system model and realizing permissions, but also how such a model supports app developers in adopting privilege separation and fine-grained, dynamic permission management for least-privilege operation on Android more efficiently.

At the heart of our paradigm shift for representing permissions in Android is an extension to Android's Binder IPC mechanism. Binder IPC is the primary IPC channel for communication among all apps and between system services

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23398
www.ndss-symposium.org

and apps. Binder allows processes to hold kernel-managed references (or handles) to remote processes that they can call via Binder IPC. The fundamental idea of our solution, called DROIDCAP, is to extend those Binder references to reflect the privileges against the remote process—a form of "extended Binder attributes" akin to the concept of "extended file attributes." Since Binder references are process-specific, just like file handles, every reference forms a unique, kernel-managed "token" that associates a caller process with a remote process and the caller's privileges for that particular remote process. By default, Android allows a free delegation of Binder references between processes via the Binder kernel module. We augment this vanilla delegation with 1) policies that govern how processes can delegate references to other processes in order to prevent leakage of privileges, and 2) management functionality to support delegation of subsets of permissions as well as efficient revocation of permissions from a (delegated) Binder reference. We integrate our Binder-based capabilities into Android's app life-cycle management, permission management, and permission enforcement (e.g., within Android's system services). As a result, in DROIDCAP we have a permission enforcement that relies entirely on the Binder references (or tokens) a calling *process* holds instead of the caller's UID (i.e., no UID-based ambient authority). We also have a clear permission delegation hierarchy between system and app processes that reflects the stock permission granting on Android but allows an efficient (re-)delegation of permissions between processes to create new protection domains (i.e., new processes with delegated permissions). To also cater for permissions enforced through Linux' GIDs facility, we explain the integration of Capsicum for Linux [20] into DROIDCAP to represent file-system-related permissions as Capsicum capabilities for file-handles. In particular, we discuss the peculiarities of Android's Zygote that had to be overcome to put app processes into Capsicum's capability mode.

As a result, DROIDCAP occupies a previously unexplored niche in Android security extensions: By blending permissions with Binder capabilities, we enable efficient least-privilege operation of processes while preserving existing Android APIs and the application model. This backward compatibility presents developers with a path for gradually adopting capability-oriented permissions and decomposing apps into components that form a logical app but run with individual rights. We demonstrate these benefits at the concrete examples of retrofitting an open source messaging app, called *Kontalk*. Evaluation of our solution shows a minimal performance impact compared to vanilla Android and full backwards compatibility to legacy apps.

II. BACKGROUND: ANDROID OS

Android OS uses a modified Linux kernel that supports an efficient Inter-Process Communication (IPC), called Binder. Binder is the primary channel for inter-app communication and talking to (system) services (see Section IV for an in-depth discussion on the Binder framework). On top of the kernel exists Android's application framework, which manages a wide range of system services and facilitates data sharing across apps. The system services enrich apps with a variety of features, such as retrieving GPS location and capturing photos. Pre-installed system apps extend those features with APIs that, for example, make phone calls and manage the Contacts.

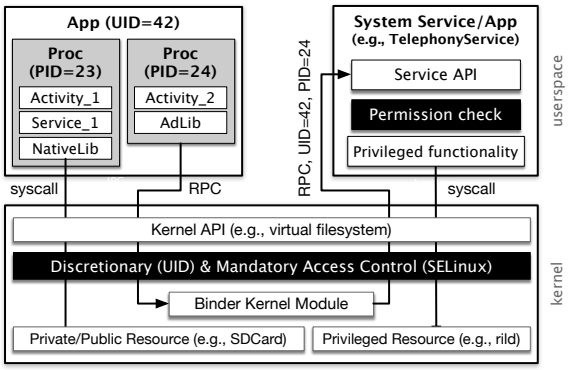

Fig. 1. Android's default security architecture.

Users can further extend their devices' functionality with third-party apps. For this work, the most related system services are: ActivityManagerService (AMS), which controls app's life cycle; PackageManagerService (PMS), which maintains app's metadata including privileges; LocationManagerService (LMS), which provides access to the device's GPS information.

All apps, including the system apps, are composed of four basic components: 1) *Activities* are interfaces that manage user interactions; 2) *Services* execute long running operations and can be bound to and invoked by other apps; 3) *Content Providers* manage access to data repositories stored in apps; and 4) *Broadcast Receivers* receive and handle broadcast messages from the system and other apps.

Android's Security Model

Android's security architecture is depicted in Figure 1. In Android, all apps and system services/apps are sandboxed. Each app runs in its (set of) processes and has a private data directory. This sandbox is defined by the app's Linux UID, which is assigned at app install-time, under which the app's processes execute and for which Linux discretionary access control ensures protection of the private directory. Android further applies the principle of least-privilege *per app sandbox*. Thus, to access resources outside their sandbox, apps have to request and been granted the necessary privileges ("permissions"). A handful of permissions protect system resources that the app processes can directly access with syscalls, such as Internet and Bluetooth sockets. Those permissions are enforced by the Linux kernel using GIDs, i.e., the UID of an app holding, for instance, the Internet permission is assigned at install-time to a Linux GID [6] that has the rights to create/read/write Internet sockets. The bulk of the permissions—more than 100 for Android Oreo [2]—however, are enforced by the system services and apps (see top-right corner of Figure 1). Android applies privilege separation between third party apps and system apps/services, where access to certain system resources, e.g., the radio interface layer daemon (rild), is only allowed to particular system services, like the TelephonyService. Those services, in turn, expose Binder IPC-callable methods to apps as part of the application framework API. Thus, apps that want to make use of those managed system resources, e.g., initiating a phone call, have to make an RPC via Binder to the service API, which then checks if the calling app is privileged enough to request the service, and if so execute on the calling app's behalf (i.e., an intentional deputy). The cornerstone of this

enforcement is Binder IPC, which provides the callee with the UID and PID of the caller. The called service can then consult the PMS whether the calling UID has the required permission.

Mandatory Access Control: Android's domain isolation and privilege separation has been reinforced with SELinux type enforcement [60], a realization of mandatory access control. With type enforcement, all subjects (e.g., processes) and objects (e.g., files) are labeled with a security type and the allowed interactions between types are defined in a set of policy rules. On Android, all app and system service processes execute in distinct domains defined by an assigned security type, which reinforces Android's sandboxing and least-privilege, and hardens the system against exploits. In contrast to permissions, which are directly exposed to app developers, who must request them in the app's manifest file, and to end users, who must approve the permission requests by apps, type enforcement is not directly exposed to developers nor users. The SELinux policies are usually static and can only be managed through administrative intervention.

Isolated process: Starting from Android v4.1, app developers are provided limited support for privilege separation via *isolated processes* [1]. Isolated processes are Service components that run under a separate, transient Linux UID that differs from any existing UID and has a separate SELinux type. Hence, this service has no access to the app's private data directory, does not hold any permissions, and cannot access any resources on the file-system. Also the middleware services refuse serving RPC by an isolated process (due to absence of permissions). To allow the host app to still communicate with its isolated process, the app can bind to the isolated process service component to invoke the service's operation. Isolated processes only provide an all-or-nothing privilege separation, i.e., there are no means to fine-tune the privileges of an isolated process or to grant additional privileges after running it.

Permission delegation: Android supports simple forms of delegating permissions between apps. First, *URI permissions*; Data entries in Content Providers are addressed using URIs (e.g., `content://com.android.contacts` for data in the Contacts app) and access to individual entries can be granted on per-URI basis. Second, *Pending Intents*; Intents are the most abstract form of inter-app and intra-app communication. A Pending Intent is a pre-populated Intent object that can be passed to another app, similar to a token. An app holding a Pending Intent can populate the remaining attributes of that Intent (e.g., payload, receiver) and then send it. The Intent will be sent with the permissions of its creator app, effectively allowing apps to temporarily delegate their permissions in a controlled way to other apps via Intents.

III. Related Work and Motivation

We present selected works that extended Android's security policy and provide an overview of systems using object-capabilities. We then relate those concepts to Android's current permission enforcement to motivate the design of DroidCap.

A. Android Security Extensions

Over the last decade, a variety of security solutions for Android have been brought forward [7], including retrofitting Android's permission system and compartmentalizing apps.

Retrofitting Android's Permissions: Research has early on investigated Android's permissions and inter-app communication [22] and a range of extensions to or retrofittings of the stock permission system have been proposed. For instance, *Apex* [47], *Saint* [51], *CRePE* [17], *TISSA* [72], and *Porscha* [50] extend the permission enforcement with more fine-grained, dynamic, or context-sensitive permissions to enable better privacy-protection, DRM, or developer-centric permissions. *TrustDroid* [14] and *XManDroid* [13] establish security domains that are isolated from each other within the system services/apps and the file-system.

Information Flow Control: A few works added information flow control (IFC) to Android, which we also pick up again in our discussion in Section VI. *Aquifer* [46] allows app developers to define secrecy restrictions that protect shared data along user interface flows across apps and uses a custom Linux Security Module to revoke Internet access to restrict data sharing automatically. *Weir* [45] realizes decentralized IFC using lazy poly-instantiation and floating labels of app components to separate memory and storage of different security contexts and allows data owners to set secrecy policies and control the export of their data to the network using (web) domain declassification. *Maxoid* [70] allows data owner app to securely delegate operations on files and ContentProviders to data processing apps, which operate on a custom view of the owner's state through union file systems/copy-on-write SQL proxy and taint tracking. It also prevents delegates from leaking secrets to other apps or the networks by revoking the necessary rights from the delegate's app instance. Jia et al. [31] permit programmers to specify a DIFC policy via (floating) labels on apps and app components. The system enforces the DIFC policy at runtime on inter-component communication between apps and within apps (assuming components run in separate processes).

Privilege Separation: Another active area of Android security research is focused on compartmentalization of apps. The predominant use-case is advertisement libraries, which have repeatedly been shown [25], [12], [21], [62], [61] to cause their host apps to be over-privileged or even to exploit their host app's privileges to invade the user's privacy. Also WebViews increase an app's attack surface by allowing untrusted web content to access device-local resources [41], [16], [39], [32], [44]. The general approach to privilege separation is to execute the library or WebView in a distinct sandbox with different UID and permissions than the host app. *AdDroid* [52], *AdSplit* [59], *CompARTist* [29], and *AFrame* [71] implement this approach for ad libs; *WIRE* [18] privilege-separates WebViews this way; and *Layercake* [56] supports WebViews and ad libs. In contrast, we leverage in DroidCap built-in features for compartmentalizing apps (e.g., *process* manifest attributes of components) introduce Binder capabilities for efficient, least-privilege privilege-separation of those compartments.

Instead of compartmentalizing apps, both *Compac* [64] and *Flexdroid* [57] establish in-app privilege separation by supplying the permission checkpoints in the system services with the call stack of the callee, which allow those enforcement points to distinguish call-sites within apps. Other works [10], [69], [30] use inlined reference monitors (IRM) to establish in-app privilege separation. The app's code is rewritten to add reference monitoring code [10], [69] or redirect permission-

sensitive calls to an external monitor [30] that acts as a filtering proxy. However, reliability of in-app privilege separation is questionable [27]. A recent alternative to inlining the reference monitor is app virtualization [9], [11], which turns the IRM concept inside out by executing the monitored app as a sandboxed child process of the monitor app (e.g., an isolated process), which we also pick up again in Section VI.

B. Object Capabilities

A *capability* is an unforgeable, tamper-proof, and system-maintained reference to a (system-maintained) object together with the capability holder's access rights on that object [19]. Historically, this has been used to control access to memory by wrapping memory pointers in capabilities [23], and in more modern systems (like [58], [36], [63]) capabilities can also reference (or name) resources that are represented as objects (e.g., files or event handles). In a capability-based system, processes can only reference objects for which they have a capability, and processes should only gain capabilities through authorized interfaces that release capabilities only to processes authorized to hold them. By design, capabilities can be easily passed between processes or protection domains, but capability transfer should be constrained by a reference monitor [23].

A number of systems have adopted capabilities [68], [37], [58], [28], [43], [33], [63], [66], [20], [65], [49], [67], [34], [24]. A few concepts from these related works should be highlighted here for the subsequent discussion of our DROIDCAP design. For instance, those systems build on the principle of protection domains that are defined by the capabilities a process holds. The isolation between the domains can be breached in a controlled way, by allowing domains to call operations in other domains and passing *sealed* capabilities [42] to the operation to entitle it to make securely use of the caller's privileges. Many of the systems build on microkernels, whose design is amenable to capabilities because microkernels build on the idea of compartmentalization and message passing. Most systems manage capabilities in kernel protected locations and operations on capabilities, including creation, are only possible through system interfaces. A few systems use a *directory*, which is a table of mappings between a capability name and the actual capability, to allow domains to discover and request capabilities. Directories can enforce security policies about which domain can request which capability and they can also ease the task of (selectively) revoking capabilities [55] when the directory is the only way for sharing capabilities.

Closest to our work is *Capsicum* [65], [20], which introduces a lightweight capability framework for BSD [65] and Linux [20]. Capsicum naturally differs from our solution in the technical realization considering the focus on different systems but forms a complementary building block for our DROIDCAP (see Section IV). Capsicum provides new kernel primitives for putting processes into a capability-mode, allowing developers to more easily adopt least-privilege operation of their applications by compartmentalizing monolithic applications into logical applications. Processes in capability mode are denied access to global namespaces, have restricted syscalls, and can only access file-system objects for which they hold a capability. File descriptors, which by design have some properties of capabilities (i.e., unforgeable, delegable tokens of authority for file objects) are extended with more fine-grained access rights. Moreover, processes in capability mode receive a UNIX domain socket for IPC between host application and sandbox.

C. Motivation for Capabilities on Android

Capabilities have been historically considered as a path to building secure, fail-safe, and flexible protection models [38]. This argument has recently been picked up for modern software development [65] and in particular the benefits of capabilities for supporting app developers in realizing least-privilege and compartmentalized apps.

Unfortunately, Android's default security architecture supports app compartmentalization only poorly. Efficient app compartmentalization requires the ability to transfer access rights between processes via a controlled channel, allowing those processes to conceptually more easily execute with least-privileges and to efficiently establish (and later delete) new protection domains [37] that are defined by the delegated rights. In Android, privileges are bound to the UID as ambient authority; thus, the creation of new protection domains necessarily entails creating new Linux UIDs and configuring their privileges at middleware and kernel level (i.e., permissions, DAC, and SELinux types and policies). UID-based DAC was designed to protect users from each other and MAC enforces static, administrator-managed system policies, i.e., neither was intended to support app compartmentalization efficiently. Related works that use process and file attributes to enforce information flow policies support per-process privileges and privilege delegation along information flows, but have not been designed for general access control to system services, other apps, or file-system objects. We discuss those solutions in more detail in Section VI. A capability-based system as the foundation would instead satisfy the requirements for efficient app compartmentalization by developers and for flexibly delegating access rights between protection domains more fittingly.

When comparing the underlying concepts of prior capability systems with Android's system design, one notices remarkable congruence between those two: First, Android applies a microkernel-like design in user-space where all apps and services run in their own processes that are connected via Binder IPC message passing. Second, apps that would like to make use of system resources have to call methods provided by the system apps/services. Those IPC-based RPC implicitly form a switch of the protection domain from app to a system domain that is privileged enough to access system resources on the app's behalf. In contrast to a capability system, no access rights are being passed on from app to service on those RPC. Third, Binder IPC references to remote processes (see next Section IV) are, just like file descriptors, already a form of kernel-managed capability that can also be passed on via Binder IPC and that gives their holders the authority to send IPC messages to the referenced remote process, although any more fine-grained privilege enforcement is deferred to the called processes (see permission check in Figure 1). To discover and retrieve Binder references, the Binder user-space framework has a central service, *ContextManager*, which is akin to a directory service in capability systems. Fourth, Android's design intends to give app developers the ability to delegate permissions to other apps, e.g., URI permissions and Pending Intents, and to compartmentalize apps, i.e., executing app components in separate processes and creating isolated

processes. However, this delegation of permissions is on a per UID basis and is realized by updating centrally managed access control lists in the app management (i.e., AMS). App components in separate processes either still execute under their host apps' UID (ambient authority) or, in case of isolated processes, with no privileges at all and without the option for fine-grained delegation of privileges.

The bottom line is that Android's design incorporates many of the concepts from capability systems, however, does not form a capability system per se by upholding a UID-centered ambient authority. This prevents app and system developers from efficiently compartmentalizing their code and adopting least-privilege. This motivated us to build DROIDCAP, a retrofitting of Android's basic representation and management of permissions that shifts Android closer to an object-capability system. In DROIDCAP permissions are delegable, per-process *Binder capabilities* and no ambient authority exists.

IV. DROIDCAP

We start by stating our design objectives. We then describe the stock Binder framework and afterward explain how DROIDCAP extends stock Binder to realize capabilities and how we integrate them into Android to change the very way in which permissions are being represented and managed. We conclude this section with case studies for using capabilities.

A. Design Objectives

Since we deploy capability-based permissions, several objectives need to be fulfilled to ensure security and efficiency:

O1. Unforgeable, communicable capabilities: Unforgeability has to hold. Otherwise, processes could easily escalate their privileges by creating unauthorized capabilities. Capabilities have to be represented in a form that can be efficiently passed on between processes of the same or different apps.

O2. Revocable capabilities: The system needs to offer the means to revoke access, either by deleting the capability or clearing the associated access rights.

O3. Controlled creation and delegation: Capabilities should only be created by authorized system components and only be given to processes that are authorized to hold them. Since capabilities in DROIDCAP represent Android permissions, those system components can be the application framework services responsible for managing permissions. Further, processes must only be able to re-delegate capabilities over a system controlled channel, which is constrained by a robust reference monitor that prevents re-delegations that would result in processes holding unauthorized access rights.

O4. Efficient creation of protection domains: The system should support app developers in easily creating new protection domains, i.e., new processes with custom sets of permissions.

O5. Backward compatibility: Capabilities should integrate into app's life cycle, mimicking how Android's permissions are requested, granted, delegated, revoked, and enforced at runtime. Instead of requiring a wholesale adaption of apps to DROIDCAP, app developers should be provided with a path to gradually adopting capability-based app compartmentalization.

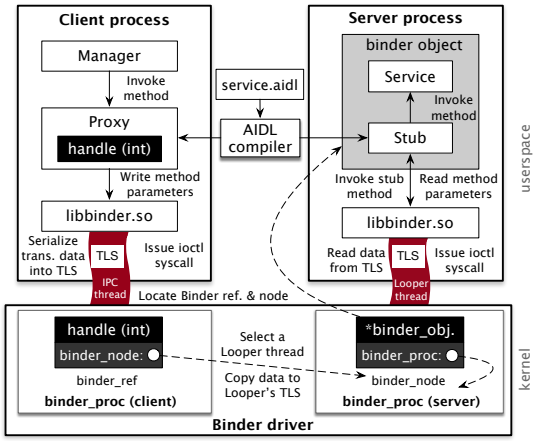

Fig. 2. Inter-app communication with Binder IPC.

B. Overview of Binder IPC Framework

Binder forms the primary IPC mechanism for any kind of IPC in Android. In general, the Binder framework serves two goals: (i) invoking methods across process boundaries; (ii) securely conveying caller's identity to the callee, facilitating a high-level access control based on permissions (see Section II).

The Binder framework consists of three main components (see Figure 2): The Binder driver, a userspace library (libbinder), and APIs of the application framework that build on top of libbinder. An optional component is the AIDL compiler, which generates Stub and Proxy Java classes from interface definitions written in the Android Interface Definition Language (AIDL) to provide app developers with a high-level abstraction of the low-level Binder operations. In the following, we explain the components of direct relevance to DROIDCAP.

1) Binder Driver and Binder Transactions: The Binder driver mediates all Binder IPC interactions facilitating the client-server architecture. Processes that use Binder IPC have to first register with the driver. Server processes register *looper* threads that block waiting for requests to handle, while clients register threads that are used to initiate IPC requests. All communication between threads and the driver are *ioctl* system calls that carry transaction data including targeted server and payload. The driver copies the transaction data from the client's thread-local storage (TLS) to the server's TLS and similarly transfer any result back. When a process registers itself, the driver creates a particular data structure for that process, called *binder_proc*, and stores all information related to the process' Binder operations inside this data structure. This *binder_proc* data structure is also called a process' *Binder context*.

To mediate IPC transactions, the driver introduces a level of reference indirection by issuing *Binder handles* to clients. Every handle designates a server's Binder service. The *Binder handle* is a 32-bit integer value that is unique per process and identifies a kernel-level data structure, called *Binder reference* (*binder_ref*), stored in client process' *Binder context*. Each *Binder reference* maintains a one-to-one mapping to a server's data structure, called *Binder node* (*binder_node*) stored in server's *Binder context*. The *Binder node* keeps a pointer to an object stored in server process' address space, called *Binder object*, which wraps the service that is callable via Binder

IPC. A process can have precisely one *Binder handle* (or *binder_ref*) for each unique *Binder object* (or *binder_node*). Each process can be both a client to many servers and a server to many Binder services. Therefore, its *Binder context* has to hold a tree of *Binder references* and a tree for *Binder nodes*. When a process dies, its Binder context is deleted.

By design, Binder allows Binder objects and handles to be passed on to other processes. The Binder driver detects transaction data that carries *Binder objects* as payload and creates a Binder reference in the recipient's *Binder context* for each transferred *Binder object* and passes the recipient the corresponding *Binder handle*, which can then be used by the recipient to invoke the referenced Binder object. This design preserves the unique identity of *Binder objects* such that they only exist in servers' address spaces. The same behavior takes place when transferring *Binder handles*, where the recipient receives a handle for a new Binder reference that points to the same Binder object as the transferred Binder handle. This guarantees that each process has its own set of references. Additionally, transferring Binder handles by means other than Binder IPC renders the handles meaningless as the corresponding kernel-level data structures for the recipient process will not be created, and dereferencing such Binder handles would always fail. As such, Binder handles form a kind of simple IPC capability that enables their holders to send IPC messages to the process of the referenced Binder object.

During a transaction, the driver injects the sender's UID and PID into the transaction data. The identity is provided by the driver, hence, forming a trust anchor for permission enforcement, where the UID is used to check the sender's permissions (see Figure 1).

2) Proxy & Stub Objects: Proxy and Stub objects implement a contract for remotely calling methods of a service. The Proxy is a client-side object that is used to marshal parameters, initiate requests to a server-side object—the Stub—and then unmarshal returned results. At the server's side, the Stub wraps a remote Binder service and unmarshals invocation parameters, invokes the actual implementation of the service method, marshals the result, and sends it back to the Proxy. At the heart of the Proxy object is a Binder handle, whose value is injected in all transaction data via this Proxy to designate the remote Binder object. The Android SDK provides Proxies for all application framework services, wrapped in easy-to-use *Manager* classes, e.g., LocationManager or ActivityManager, that can be retrieved at runtime (see next section).

C. Management of Binder Objects

Since it is impractical to pre-populate processes with handles of all Binder objects in the system, client processes have to request Binder handles at runtime. To ease the discovery and request of handles, system services implement the logistics for registering, storing, and handing out Binder handles. In Android, two central services take on this role: The *Context Manager* (CM) for handles of system services, and *Activity Manager Service* (AMS) for handles of Content Providers and app-provided Services. Thus, CM and AMS are comparable to *directories* in capability systems.

Both CM and AMS run on dedicated processes— *servicemanager* and *system_server*—and provide their services via Binder IPC. Thus, the Binder handles of both services must be a priori known, or at least, easily obtainable. The AMS is a system service that is registered with the CM and, hence, its handle can be retrieved through the CM. The CM, in turn, is assigned a globally reserved, well-known Binder handle of 0. The Binder driver delivers all transactions with a 0-handle as target to the CM. Thus, processes can use the CM to register their services and to request handles to other registered services. Figure 3 illustrates this service registration and discovery in more detail. We defer the explanation of DROIDCAP modifications in that figure until Section IV-D.

1) Binder Object Registration: To expose a Binder object via IPC to other processes the host process has to register it with CM (or AMS) to allow other processes to retrieve its handle. For example, registering a system service with CM (❶ and ❷) is done using the *addService* method of *ServiceManager* where the Binder object and service name are passed as method parameters. The Binder driver detects that a Binder object is being transferred and creates a corresponding Binder reference in CM's Binder context before copying the transaction data along with the new handle to the CM process. In turn, the CM adds a new entry for the registered service in a local key-value store of service names and their Binder handles. Registering a Binder object of a Content Provider (❸) is conceptually identically, except that it is the AMS whose Binder handle needs to be discovered first in order to register the Content Provider under its URI with *publishProvider*.

2) Binder Handle Discovery: Two SDK APIs, namely *getSystemService* and *getContentResolver*, are used to retrieve Binder handles of system services and Content Providers, respectively. In general, what returns from those APIs is a Manager object that uses a pre-compiled Proxy that embeds a Binder handle to the remote Binder object. For example, calling *getSystemService* with "location" as a parameter (❹) issues a Binder transaction to the CM and finally returns a LocationManager object that can be used later on to invoke operations of the remote LocationManagerService (LMS). At the kernel level, the Binder driver detects that a Binder handle is being transferred (from CM to app), therefore it locates the associated Binder reference in CM's Binder context, inserts a new copy of the Binder reference into app's Binder context, injects the new Binder handle into the data buffer of the transaction data, and finally copies it back to app's TLS. Android's SDK retrieves the Binder handle, builds a Proxy and Manager around it, and returns the Manager object as result of *getSystemService*. The same flow goes for retrieving handles of Content Providers requested in *getContentResolver* (❺), except that the AMS returns the handle.

3) Binder Handle Invocation: When a method of a Manager is called (❻), the contained Proxy marshals the method parameters and passes the data to the Binder userspace library (see also Figure 2). The library builds the transaction data, injects the Binder handle embedded in the Proxy object as the target's handle of the transaction data, and write the transaction to the driver. The Binder driver uses the transaction's Binder handle to locate the Binder reference in client's Binder context. This Binder reference points to the server's Binder object. The driver injects the client's UID and PID into the transaction data and copies the transaction data into the TLS of the server's process. The server's Binder userspace library and Stub take

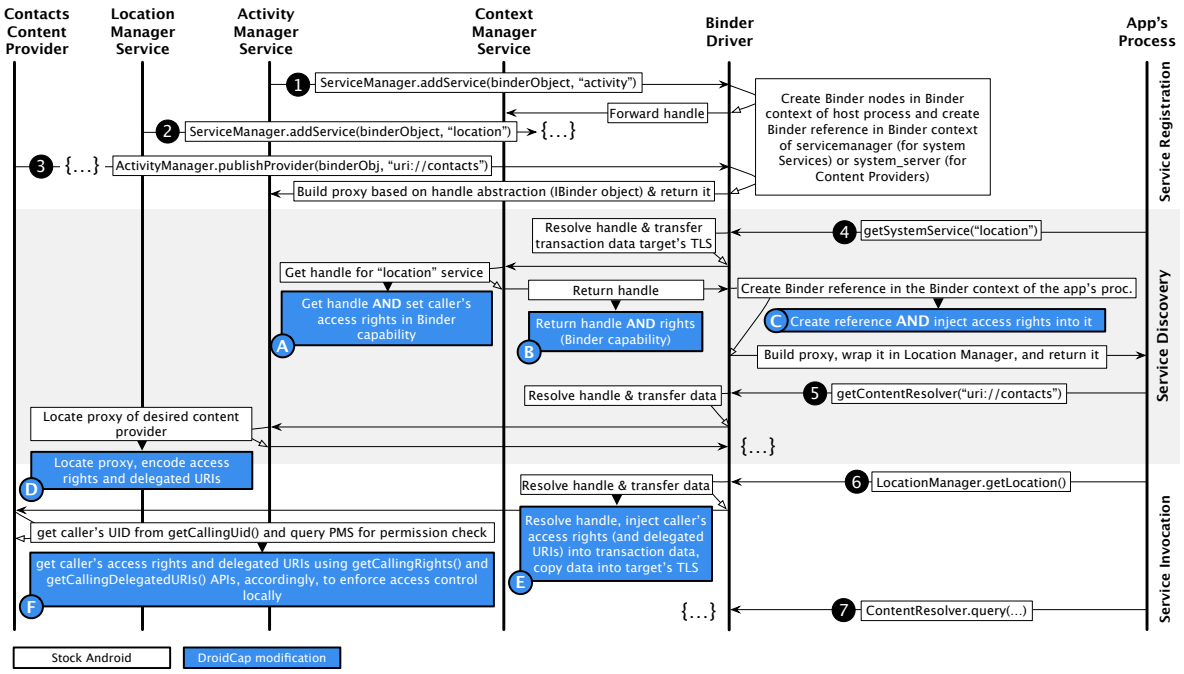

Fig. 3. Delegation, revocation, and invocation of Binder objects/handles. Modifications for capabilities are highlighted.

care of reading the data from the driver and invoking the server's service method. The service uses *getCallingUid* to identify the client based on the UID provided by the driver and queries the PMS to check whether the calling UID holds specific permissions (e.g., permission check in Figure 1). Querying the PMS could be an IPC invocation in itself when the service does not execute on the same process as PMS, for instance, as usually is the case for Content Providers (❼).

D. DroidCap: Design and Implementation

DROIDCAP is a retrofitting to Binder that brings object capabilities to Android's permission enforcement. The core idea behind DROIDCAP is to use Binder handles as capability tokens and to associate Android's permissions as access rights to them. Thus, DROIDCAP does not aim at redesigning Android's permission system but instead realizes a new representation of permissions via object capabilities. This eases privilege separation between app's components and opens an adoption path to app compartmentalization.

In DROIDCAP, privilege separation can be achieved through standard facilities of Android, such as isolated processes or running an app's components on separate processes. Assigning privileges to processes, by means of capabilities, instead of UIDs eliminates the complications associated with the ambient authority (**Q4**). DROIDCAP integrates smoothly in apps' life-cycles and provides the same experience for app developers and users. To benefit from the security features of DROIDCAP, app developers need to design their apps with capabilities in mind and use the security APIs of DROIDCAP.

Our current DROIDCAP design focuses on confining, delegating, and revoking access to system services and content providers, which account for the vast majority of system resources available to apps. Thus, we focus on representing Android's *middleware* permissions as capabilities. In Section IV-E, we present the integration of Capsicum [65], [20] into Android, which can be used to represent file-system permissions as capabilities.

1) Representation of Binder Capabilities: The Binder handle is a token that fulfills the fundamental requirements of object-capabilities: unforgeable, communicable, designating a resource, and uniquely assigned to a process. Since the Binder driver manages Binder handles, the driver is a reference monitor for all Binder IPC, which guarantees the requirements mentioned above. DROIDCAP employs the Binder handle as an object-capability for IPC-callable processes (satisfying **Q1** and **Q5**) and extends its data structure to carry additional capability fields (see Figure 4). Permissions that the capability holder has for the referenced service can be stored in a bitmask (*access_rights*) and a linked list of strings (*str_permissions*), as we explain later. Further, we added a parent field *parent_proc* that points to the process that delegated this Binder reference to the current holder, as well as *delegation_flags* to express delegation constraints (which is inspired by *sealed* capabilities [42]). Binder capabilities, similarly to file descriptors or SELinux security contexts, are assigned to processes within the kernel's process management. In DROIDCAP, processes always start with an empty set of Binder capabilities and gradually receive them at runtime, e.g., through delegation. This is independent of the PID, i.e., this is a *per-process* but *not per-PID* privilege management.

2) Management of Binder Capabilities: We explain how Binder capabilities are managed in DROIDCAP. In this context, we explain our changes to stock Android's services (see DROIDCAP modifications in Figure 3).

a) Creating Binder Capabilities: In our current DROIDCAP design, we deliberately stick close to Android permissions for process privileges to preserve backward compatibility. One

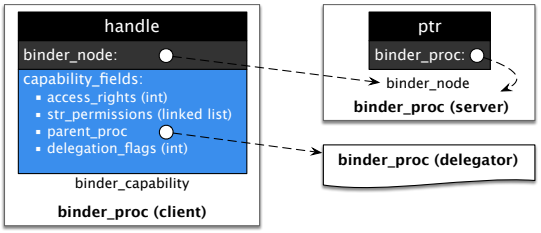

Fig. 4. Binder capability as a combination of Binder reference (to Binder node) and capability fields for permissions, parent process (i.e., delegator), and delegation constraints.

of the challenges DROIDCAP had to resolve is maintaining the sync between access rights in Binder capabilities and permissions, such that any change of an app's permissions should be reflected in the access rights encoded in capabilities held by processes of the app. Since permissions are managed by the PMS, DROIDCAP has to rely on this service to retrieve each app's permissions. However, with Binder capabilities, capability creation cannot be decoupled from the delegation of access rights, since Binder capabilities are attached to Binder references, which are only created when the corresponding Binder object or a handle to that object is being passed on via Binder IPC. To solve this problem, we leverage the central role of CM and AMS for handing out Binder references to app processes (i.e., ❹ and ❺ in Figure 3), which is comparable to directories in capability systems. They form trusted system components that are the only processes able to create Binder capabilities for processes that are authorized to hold the capability (**Q3**). We created a channel between PMS and CM to supply all apps' permissions to the CM the instant they are granted or revoked. Whenever the CM should hand out a Binder handle to a calling process, the CM will use this information to create a Binder capability from this handle (Ⓐ) that carries the access rights of the calling process for the service referenced by the associated handle and then return this Binder capability (Ⓑ,Ⓒ). The capability allows that process to send IPC messages to the referenced service, which will use the capability information to enforce access control. Similarly, the AMS is responsible for handing out Binder handles of Content Providers and non-system services (Ⓓ). Equivalent to how CM couples issuing of Binder handles with computing access rights of the caller, the AMS consults the PMS—as they run on the same process—for the caller's general permissions, while it additionally uses its local database of delegated URI permissions to encode the caller's per-URI access rights before returning the Binder capability to the calling process.

To encode *standard* permissions from the application framework into Binder capabilities, we use the capability's access rights bitmask. Since those are well-known permissions, using a bitmask is a highly space-efficient representation. We have run an analysis on Android's standard permissions and their point of enforcement, and identified permissions enforced at each service. For example, the `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permissions are used to control access to the LMS. Using this information, AMS and CM identify the relevant permissions of the service for which they hand out a capability and encode each permission the requesting app should have in the access rights bitmask. At enforcement points in services, e.g., LMS, access rights are decoded accordingly. However, app developers can define custom permissions in their app manifests and Content Provider can be protected with URI permissions. Since those permissions are unpredictable strings, encoding them into a bitmask is an impractical solution. Thus, DROIDCAP uses the linked list of strings of Binder capabilities (*str_permissions* in Figure 4) to store custom permissions and delegated URI permissions. Upon invocation of the Binder reference, all permission strings are copied to the callee's TLS. To avoid unnecessary performance degradation, only string permissions that correspond to a specific Binder object are stored in the Binder capability that points at that Binder object.

Determining which permission has to be encoded for the current caller to CM/AMS can be handled in one of two ways. For backwards compatibility (**Q5**), the permissions can depend on the caller's UID, i.e., if any of the processes of a UID requests a handle, the capability's access rights are set to those of the caller's ambient authority. This preserves the same access control enforcement as on default Android. For apps that explicitly make use of capabilities, the app developer can state in the app's manifest dedicated `process` tags and `use-permission` tags *per declared component*. The AMS and PMS manage at runtime a logical mapping of those components to their processes, and hence can report to CM/AMS the permissions of the calling *process ID* that will receive the Binder handle. Hence, CM/AMS can configure the Binder capability per-component.

3) Controlled delegation: Delegation of Binder capabilities has to be controlled (**Q3**) to avoid processes from holding unauthorized access rights. At the same time, DROIDCAP has to support different paths for delegation and has to consider the technical intricacies of the Binder framework.

a) Delegation flags and parent field: Once a Binder capability has been created and transferred to a process, this process might want to transfer the Binder capability to other processes. Without constraints on the delegation of capabilities, this would undermine the security policy. For instance, a process could delegate a Binder capability to another process that never was authorized to hold those access rights (and would not have received them from the CM or AMS). Thus, delegation of Binder capabilities is subject to constraints enforced by the Binder driver. Those constraints are expressed primarily through *delegation_flags* and the *parent_proc* fields of Binder capabilities: 1) A capability delegated from one process to another must in any case carry a subset of the access rights of the sender. 2) A capability with *no_delegation* flag set cannot be delegated to other processes. A capability with *limited_delegation* flag set can only be delegated to another process of the same logical app. This flag can only be further restricted but not relaxed during delegations. CM and AMS create capabilities with *limited_delegation*. 3) Every capability carries a parent field that points to the process that delegated it and the Binder driver sets this field during delegation. Changes of the access rights of a capability, such as down-/upgrading and merging, can only be done by the parent process (within the limits of the parent process' access rights) or the system (i.e., CM and AMS), which always takes precedence. For simplicity, we keep only a single parent.

b) Inheritance: Binder capabilities, in contrast to file handles, are *not inherited* by forking processes. The Binder driver data structures are not integrated with the kernel's process management and, hence, new processes always start with an empty set of Binder references/nodes.

c) Capability merging: Every Binder reference and hence capability is a one-to-one mapping to a Binder node and for every Binder object exists exactly one Binder node. Thus, if a process would receive a Binder reference to a Binder node for which it already holds a reference, Binder would simply abort the redundant operation. However, with Binder capabilities, the access rights and string permissions of two capabilities for the same Binder node would be merged if the parent field matches, i.e., we assign the union set of the access rights and string permissions to the existing capability. Supporting merging of capabilities from different parents, would require per-parent access rights lists to support different revocation strategies (e.g., consensus or priority).

d) Delegation paths: DROIDCAP provides three paths for inter-process delegation of Binder capabilities:

(1) Bound services: A process holding a Binder reference to another process executing a bound Service can transfer a capability directly as part of a Binder transaction. This path is used, e.g., to delegate capabilities to isolated processes.

(2) Intents: A process can attach a capability to an Intent message. Since all Intent messages are routed to their destinations through the AMS, the capability is transferred first into the AMS' Binder context and from there to the Intent receivers' context. To ensure that the parent field of the capability is correct, AMS uses its special role and instructs the Binder driver to not update the parent field when transferring the capability to a receiver. This mechanism is used, for instance, when delegating a URI permission to another app via an Intent.

(3) ioctl command: A process can also instruct the Binder driver to delegate a capability to another process for which the delegator knows the PID. If the delegatee already has a capability for the same Binder node, the privileges are merged. If not, this capability of the delegatee is initially dormant, i.e., the process has no handle to use it until a transaction transfers a reference to the same node to the delegatee. This path is used during the *grantUriPermission* method. If an app grants access to a URI to another process and that process has not yet a ContentResolver to the corresponding ContentProvider, the capability is dormant until the ContentResolver is requested from the AMS (❺ in Figure 3).

4) Revocation: To support the dynamic permissions of stock Android where users can revoke app's permissions at runtime, DROIDCAP enables revocation and downgrading of access rights at runtime. We found that in stock Android an app's processes are killed when permission settings change. Although this is a rather crude approach for revocation, it is effective and in line with stock Android for revocation of Binder capabilities in DROIDCAP. Binder capabilities are removed from the kernel when the owning process dies (**Q2**) and they have to be requested again when the process restarts, with changes on permissions encoded.

To support app developers in revoking delegated capabilities (**Q2**), we introduced new commands to the *ioctl* call to

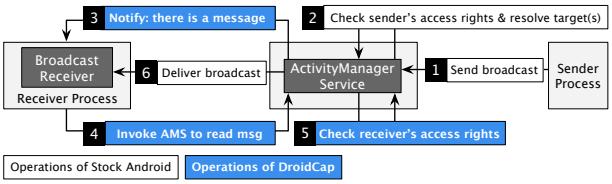

Fig. 5. Enforcing permissions of broadcast sender and receiver in the AMS

the Binder driver. First, if the delegating process has a Binder handle to the delegatee process (e.g., bound service), the delegator can present the driver with its handle to the delegatee's process and the capability of the delegator that was passed on. Using those credentials, the delegator can instruct the driver to revoke access rights from the delegatee's capability. If the Binder driver can locate a corresponding capability in the delegatee's Binder context and that capability's parent field points back to the delegator process, the Binder driver can revoke (or increase) the delegatee's access rights within the bounds of the delegator's access rights. Second, if the delegator has no handle to the delegatee's process (e.g., delegation via Intents or *grantUriPermission*) but has a PID or UID (e.g., the delegatee's packagename is known), the delegator can instruct the Binder driver to revoke a capability or access right of the UID's processes or process under the PID. Third, the delegator can purge a capability by instructing the driver to remove a capability it holds from all processes in the system to which the delegator transfered that capability.

5) Invocation: From developer's point of view, invoking a Binder capability is identical to invoking a regular Binder handle. However, the driver additionally injects the caller's access rights and delegated URIs from the caller's capability into the transaction data before passing the request to the target process (Ⓔ in Figure 3). The target process, e.g., a system service or a Content Provider, can use two new API methods *getCallingRights* and *getCallingDelegatedURIs* to check the calling process' permissions (Ⓕ). Given that *getSystemService* and *getContentResolver* are the only ways to retrieve Binder handles of system services and content providers (except for delegation within logical apps), it is guaranteed that those permissions were authorized by the system.

As such, our current DROIDCAP design diverges from a *pure* object-capability system, where the access to the target service would be enforced within the Binder driver, while we defer the enforcement of capabilities to the target processes. We adopted this design from stock Android to avoid mixing middleware semantics (i.e., permissions) with kernel-level semantics and to ease backward compliance to stock Android.

In addition to checking the access rights of the calling processes, the application framework enables apps to control which processes are allowed to handle specific events initiated by the caller. For example, an app might require a Broadcast receiver to hold specific permissions where sending the Broadcast itself requires a permission from the sender. In stock Android, this situation is handled by the AMS (see Figure 5), which receives the send broadcast request (❶), resolves target Broadcast Receiver and checks sender's permissions to broadcast this specific Intent (❷), and finally delivers the Broadcast message (❻). To cover this scenario in DROIDCAP, without diverging from the object-capability model where access rights

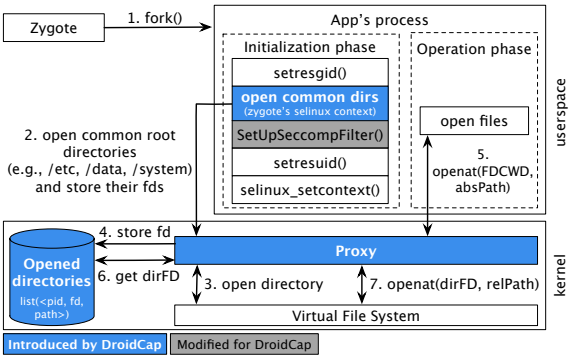

Fig. 6. Sandboxing app processes using Capsicum capability mode

of Binder capabilities are checked on IPC-recipient's side, we introduce a callback from the AMS to the Broadcast Receivers, triggering them to invoke the AMS (3) and request that Broadcast message to be delivered (4). At this point, the AMS can check the access rights of the receivers' capabilities to authorize delivery of this Broadcast according to the sender's specification (5) and delivers the Broadcast message only if successfully authorized (6). This extension to the Android SDK preserves backward compatibility with existing apps.

E. Capsicum Capabilities

The bulk of Android's permissions are enforced by the application frameworks services and apps, which are called via Binder IPC. However, a handful of permissions are enforced by the Linux kernel (using GIDs) and do not involve IPC, hence, excluding Binder references as technical realization of capabilities for those file-system-related permissions. To also cater for those file-system permissions, we integrated *Capsicum for Linux* [20] into DroidCap and ported the Capsicum userspace library [3] into Android's middleware. Although Android is based on Linux, this integration was a non-trivial technical task, which we elaborate on the following.

Generally, the capability mode, as implemented in *Capsicum for Linux* project, enforces four conditions: 1) A process that entered the capability mode cannot exit it and this mode is inherited by all its child processes. 2) In capability mode, a process has only access to whitelisted syscalls with whitelisted parameters (e.g., *open* syscall is prevented whereas *openat* is allowed). Blocking syscalls forces developers to rely only on file descriptors, which are wrapped as capabilities, for file-system operations. Whitelisting policies are enforced by Linux *seccomp*. 3) A process in capability mode can create new file descriptors only under the directories for which they were delegated access to. File descriptors are either delegated statically, by means of opening them before entering the capability mode (e.g., using *open* syscall), or dynamically by receiving them over IPC (e.g., sockets) from another process. 4) To invoke a syscall on a file descriptor, this file descriptor has to be associated with the necessary capabilities, otherwise the call will fail (e.g., reading a file and changing its offset requires the *CAP_READ* and *CAP_SEEK* Capsicum capabilities). Those capabilities are enforced when resolving the file descriptor to a file struct in the kernel.

1) Integration into Android: Starting from Android 8, *seccomp* policies restrict the number of syscalls available to apps to those used in the bionic library, and Android P enforces even more restrictions [4]. Fortunately, this significantly reduces the effort required for porting the Capsicum's *seccomp* policy to Android. By comparing the Android and Capsicum policies we find that Android's policy is more relaxed than Capsicum's. For our integration, we used the policy of Android 9 after making sure none of the whitelisted syscalls is likely to break the conditions of the capability mode mentioned above. The only modification to Android's *seccomp* policy was setting the *OPENAT_BENEATH* flag in the process. Capsicum requires this flag to prevent processes from escaping the directory referenced by the file descriptor passed to the **at()* syscall, i.e., the relative path is not allowed to contain ".." nor to start with "/". Using this policy, when *zygote* calls *SetUpSeccompFilter()* (see Figure 6), Capsicum's capability mode becomes effective immediately. Naturally, app processes will crash in capability mode unless they are supplied with necessary file descriptors for the operation. However, supplying processes with necessary file descriptors is a non-trivial task and the primary challenge for putting app processes into Capsicum's capability mode: Capsicum assumes that the creator of a process in capability mode has an operational relationship with that process (e.g., same developer) and knows exactly which file descriptors to pass on; however, for *zygote* it is impossible to predict all required file descriptors an app might need, given the complexity of Android apps that are composed of several libraries running on different threads and continually accessing the file-system. Passing necessary file descriptors to app processes in capability mode at runtime—they cannot open them themselves—is not an option, since it would require an all-privileged delegator process that can open all requested files and delegate their descriptors to app processes. Not only would this be inefficient, but also be a direct violation of least-privilege design.

Instead, we solve this challenge with a combination of statically opening possible root directories of needed files prior to entering capability mode and relying on SELinux' mandatory access control to constrain app behavior. We implemented a proof-of-concept prototype of this solution as part of DroidCap. We instrument *zygote* (see Figure 6) to statically supply forked processes with valid file descriptors for common root directories (i.e., `/dev`, `/system`, and `/data`). To do so, we relax SELinux' policy to allow *zygote* and apps to create file descriptors for root directories with necessary POSIX permissions to be able to use them in **at* syscalls. Allow such operations might seems decremental to Android's security, however, the existing SELinux rules restrict the resources accessible by the process inside those directories. For example, there is an SELinux rule that permits apps to access the `/dev/socket/netd` interface but there is no rule to allow access to `/dev/kmsg`. However, using those file descriptors requires all call-sites to switch to *openat()* syscalls instead of *open()*. Since the default *seccomp* policy prohibits *open()*, all call-sites in fact already use *openat()*, but unfortunately Android's libraries show a persistent pattern of using *openat()* with *AT_FDCWD* as the first argument and an *absolute* path as the second argument, which yields the same semantics of using an *open()* syscall. Due to the additional *OPENAT_BENEATH* flag in capability mode, this patter fails and causes apps to crash due to denied *openat()* syscalls. To avoid refactoring the entire Android SDK to change those calls to be compliant with the *OPENAT_BENEATH* flag, we implemented a proxy for

openat() syscalls inside the kernel that draws from the design of syscall interposition techniques [53]. When *zygote* opens the root directories (step 2), we intercept the call (see *Proxy* in Figure 6), open the directories, and store their file descriptors along with the paths they reference in a special table for the designated process inside the kernel (step 4). When the *openat()* syscall is invoked (step 5), we again intercept the call in the proxy, get the root directory of the absolute path, retrieve the file descriptor for that directory from the table of root directories (step 6), and rewrite the call such that a relative path is opened under the file descriptor of the directory (step 7).

2) Delegation of Capsicum Capabilities: Transferring Capsicum capabilities over sockets is the natural way for delegation, but since the Binder framework also enables transfer of file descriptors, it can be instrumented to transfer Capsicum capabilities making delegation/revocation more convenient to app developers. However, in either case, the kernel has to be aware of all delegated Capsicum capabilities to enable revocation, and more importantly, to prevent arbitrary accesses to file-system resources. To understand how delegation works, we consider a scenario where an app delegates access to the Internet, enabling the receiver to open an Internet socket. Upon transferring of the Capsicum capability to the receiver, the kernel creates a new entry inside the table of delegations. The entry identifies the source, the target, delegated rights, and the inode of the file descriptor. Further checks are made to ensure the source has access to the delegated object with the rights to be delegated or has the necessary privileges (e.g., Zygote).

When the receiver of the delegation tries to make use of the delegated capability to the Internet driver and tries to create a new Internet socket, it opens the driver using the *openat* syscall. The kernel would immediately retrieve the source's inode identified by the relative path and the file descriptor of the parent directory, and compares it with the one delegated and previously stored in the table of delegations, if any. This is to ensure that both operations (i.e., delegation and opening of delegated capability) address the same resource. Checking the table of delegations is an inevitable operation to guarantee that all delegations are considered. Given that DAC would prevent the receiver process from opening the file because it does not hold the necessary DAC privileges (identified by the group ID), our kernel extension overrules DAC. Then, the kernel sets the delegated rights on the file descriptor forming a Capsicum capability, and finally marks it with the *delegated* flag before returning it. This form of delegation happens when Zygote forks new processes whose apps are granted the Internet permission. To enable usage of delegated Capsicum capabilities without triggering a clash between security policies of Capsicum and DAC, we disable GID-based checks for file-system accesses only when the file descriptor in use is a delegated Capsicum capability, identified by the *delegated* flag. This technique can work for delegating access to files in the private directory.

3) External Storage: Given Android's unconventional mechanism [5] in mounting different storage devices that differ in file-system permissions based on the granted runtime Android permissions for reading and writing to the external storage, access to files stored on the external storage using Capsicum capabilities would require integration with the *vold* daemon which is beyond the scope of our prototype.

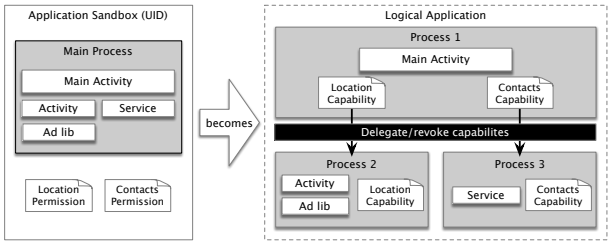

Fig. 7. Compartmentalizing an app to form a logical app.

F. Case Studies

In this section, we present two case studies that show DROIDCAP's potential in supporting app developers in compartmentalizing their apps using Binder capabilities. We defer the discussion on how to retrofit existing apps to utilize DROIDCAP's new security features to Section V-B.

1) Isolated processes: Isolated processes are easily declared in the app's manifest and are realized as a service component that has no privileges in the system and can be bound to by the host app processes. By default, an isolated process cannot retrieve any Binder handles from the CM and AMS and also cannot successfully open any file handles. Thus, isolated processes force the app developer to make an all-or-nothing decision: either compartmentalized code executes with all privileges of the ambient authority or with no privileges at all. In DROIDCAP, the app developer can gradually increase the privileges of an isolated process by passing Binder capabilities to the bound service of the isolated process. Since the transient UID does not matter for permission enforcement but only the access rights of the process's Binder capability, this provides a path to the developer to start from a completely deprivileged protection domain to build a least-privilege component. Adding Capsicum to DROIDCAP would further allow delegation of file handles to isolated processes.

2) Interstitial AdMob advertisements: Different security extensions [52], [59], [29], [71] proposed privilege separation for advertisement libraries and virtually all of them relied on sandboxing the libraries in a separate app (UID) through system modifications [52], [59], [71] or app rewriting [29]. We analyzed the most popular AdMob advertisement library and found that for interstitial ads the library does not have any tightly coupled operational or state dependencies with its host app. Thus, interstitial ads can be deployed in a separate Activity component that executes in its own process and that is simply invoked by calling the Activity. With DROIDCAP, this easily allows us to compartmentalize the app into a logical app as depicted in Figure 7, where the advertisement library runs in a separate process to which the host app delegates only a subset of its privileges (here *location*) while withholding other permissions (*contacts*). On DROIDCAP this is accomplished using the Binder driver APIs or, more efficiently, using the per-component *use-permission* attribute in the app manifest.

V. EVALUATION

In this section, we present the performance, security, and functionality evaluation of DROIDCAP. We tested DROIDCAP on different Android (6.0, 7.1, 8.0, 9.0) and Linux kernel versions (3.4, 4.4, 4.9). However, all subsequent evaluations

TABLE I. PERFORMANCE (IN CYCLES) FOR BINDER TRANSACTIONS.

System	Mean	σ	Min	25%	Median	75%	Max
Android	34,679	46,360	9,866	16,404	22,004	32,961	369,388
DroidCap	36,231	53,786	9,626	16,327	21,877	34,314	453,305

used Android 9 (android-9.0.0_r1) and a Linux kernel v4.9. We ran all tests on a HiKey960 device that has an octa-core 1.8 GHz Cortex-A53 CPU and 3 GB RAM.

A. Performance Evaluation

For kernel and native layer micro-benchmarks, we used the ARM Performance Monitoring Unit to capture CPU cycle counts. Measurements at the application framework utilized the system timer via *System.nanoTime*. Through all measurements, CPU clock was fixed at 1.875 GHz. All results are the average computations of 50,000 trials, unless stated otherwise. All margins of error are computed for a 95% confidence level.

To compare DROIDCAP with stock Android, we measured the required CPU cycles for 500k Binder transactions in the Binder driver for both systems under equal setups, i.e., both systems boot up, do not start any app, and execute a sequence of taps on the screen using Android's application exerciser monkey to generate transactions. Since both measurement series included a few extreme outliers (of three or more magnitudes lower and higher than mean), we eliminated in both series the measurements below the 5th percentile and above the 95th percentile. Table I summarizes the overall results. On average DROIDCAP induces an overhead of 3.44% (1,552 cycles) to transactions. A closer look at the types of transactions reveals that 6.15% of all transactions contained Binder handles and 10.67% contained Binder nodes—both of which require more processing from the driver. The weighted average considering the frequencies of transaction types is 1,538 cycles or 3.41%. Using Mann-Whitney and Kolmogorov-Smirnov tests, we can attribute those differences in means ($U{>}10^{11}$, $p{<}.001$) and frequency distribution ($D{=}0.037$, $p{<}.001$) to our modifications. In general, both systems exhibit a very skewed frequency distribution, a very large standard deviation, and a median that is higher than the 75th percentile.

We performed further micro-benchmarks that separately measure the operations of DROIDCAP to gain deeper insights on the contributing factors to the overall overhead. Retrieving and copying the access rights of the caller process to the callee's TLS adds a negligible overhead of 61.78 CPU cycles ($\sigma{=}26.77{\pm}0.37$) in comparison to the 34,679 cycles consumed for each Binder transaction in stock Android. However, once the access rights are copied to the address space of the callee process, it can use those information locally for access control enforcement. To understand the positive impact of this feature we consider the following two examples: First, the *Contacts* content provider requires only 10.99μs ($\sigma{=}4.14{\pm}0.1$) to retrieve access rights from its TLS, decode them, and enforce access control of *READ* and *WRITE* permissions, where in stock Android it has to issue a round-trip IPC to the PMS that takes 226.40μs ($\sigma{=}51.16{\pm}1.14$). Second, although LMS is hosted on the same process as PMS, thus permissions checks are answered process-locally, DROIDCAP achieves an almost seven-fold gain (11.18μs) for access control in comparison to the 77.02μs required for permissions checks in stock Android.

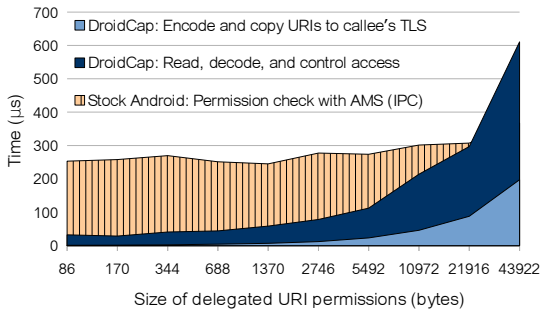

Fig. 8. Overhead for URI permission enforcement.

In the following, we consider the penalty incurred by copying string permissions to callee's TLS upon the invocation of a Binder capability that carries them by focusing on URI permissions as an example (see Figure 8). In general, this is an expensive operation that produces a linearly increasing overhead (slope=0.0142) with increasing number of delegated URIs. Although this increase is considerable in comparison to the average time of a Binder transaction, our measurements show that DROIDCAP would still outperform stock Android for overall access control enforcement when the calling process holds delegated URIs of size 21.4kb or less. Since URIs are strings, it is hard to give an exact number of URIs that have to be delegated in DROIDCAP to incur a penalty. However, as an example, a process that has been granted 53 URIs, each of which is a 100 ASCII-characters long, would yield almost the same overall overhead for access control check in DROIDCAP (296.83μs) in comparison to the round-trip IPC check to the AMS in stock Android (307.15μs). Unfortunately there exist no statistics on the average number of delegated URIs between apps, but we would argue that a developer who delegates this amount of URIs might not follow best practices. If indeed this high number of delegated URIs is needed, DROIDCAP might require the app to make a lookup in the capability attribute using ioctl calls to the Binder driver instead of copying them.

The average overhead of the Binder kernel module for delegating and revoking a Binder capability using bound services and Intents is 562 (σ=427$\pm$3.72) and 2,851 (σ=2,396$\pm$22.57) CPU cycles, respectively. Note that we ignore measuring the user space overhead, because such operations are identical to standard IPC operations where a Binder handle is being transferred as part of the transaction data. Delegation and revocation of a single string permission through direct *ioctl* using a known PID of the delegatee or the Binder handle that was previously delegated consumes on average 34,035 cycles (σ=14,038$\pm$123), which translates to 17.72μs that are used for looking up Binder reference(s) and string comparisons.

Specific to the current design of DROIDCAP is the reporting of permissions from PMS to the CM. In worst case, where all system permissions are granted to a single app, the aggregated overhead of granting/revoking a new permission as measured from the PMS is 36.51μs ($\sigma{=}11.26{\pm}0.2\mu s$). In general, reporting permissions is a rare operation because users tend to grant apps permissions upon their request and rarely revoke them afterwards. When processes request Binder capabilities from the CM, DROIDCAP takes 0.29μs ($\sigma{=}0.11\mu s$) to compute the access rights. A well designed app should perform this operation only once over the lifespan of the calling process.

B. Security Evaluation of Kontalk

We retrofitted an open source messaging app, called *Kontalk*, to use Binder capabilities. The app requests 24 permissions, 11 of which are dangerous, and it can access the Internet. The app consists of 37 components and includes about 30 third-party libraries.

We started compartmentalizing the app by executing each component in its own process and then reconstructing the connections between the components via IPC to preserve app's functionality. Unfortunately, compartmentalizing existing apps is, in general, a hard problem [65], since components might share a state, and requires app developers to adopt distributed app development to build *logical apps* consisting of different processes connected via message passing. Ideally, app developers should be supported in this task of compartmentalizing their apps, for instance through new tools and frameworks [26]. Android apps are not different in this regard. Through dynamic and static analysis of the top apps on the Google Play-Store, we found that none of those apps could be *automatically* compartmentalized. We attribute that to several programming practices that expect the app to run on a single process and which are widely used in the analyzed apps, such as (i) sharing class variables and singletons across components, (ii) invoking app's Binder services locally instead of binding and invoking them over IPC, (iii) broadcasting messages within the process using *LocalBroadcastManager*, and (iv) sharing primitive data via *SharedPreferences*, which, by design, do not synchronize reads and writes requests from different processes. Thus, to compartmentalize apps, we need to replace those impeding practices with secure alternatives that are semantically equivalent. Our strategy is to use a ContentProvider to share primitive class variables and SharedPreferences data, bind to local Binder services, and use explicit Intent broadcasts.

To identify the permissions that are used by each component in Kontalk, we used an existing permission map [8] and complemented it with missing permissions enforced at the filesystem and the native layer. We find that 17 out of the 37 components do not need any permission. The other 20 components need between 1 to 8 permissions each. This is a fitting example of violating least privilege because in the normal execution on stock Android, each component and library inherits all granted permissions. After retrofitting Kontalk to use Binder capabilities, every component runs with least privileges. This also concerns the privileges of third-party libraries included in those components, e.g., *TrueTime* and *BarcodeScanner* inherited privileges were reduced to no-permissions, and Camera and Internet, respectively. Using Binder and Capsicum capabilities combined would further reduce the *BarcodeScanner* privileges to only the Camera.

C. Backwards Compatibility

To evaluate DROIDCAP's backward compatibility, we dynamically tested the top 50 apps of 44 categories on Play using the monkey exerciser on stock Android and afterwards on DROIDCAP. We fixed the seed and the number of events (1000) for the monkey to produce the exact sequence and types of events for all apps. We filtered all apps that already crashed or failed to install on stock Android, which left a test set of 1,752 apps. Our results show that about 2% of the apps failed the test on DROIDCAP due to runtime exceptions, which we manually inspected and found to not be caused by DROIDCAP's changes but uncontrolled variants, such as network delays and resources loading time. This demonstrates the backwards compatibility of DROIDCAP with existing apps.

VI. DISCUSSION

Different solutions for type enforcement, information flow control (IFC), and app virtualization also realize fine-grained, flexible per-process/per-component privileges and consider the delegation of access rights. However, despite their apparent parallels to our work, their objectives differ from DROIDCAP's: that is providing app developers with a path for efficiently compartmentalizing their apps and applying the principle of least privilege to their apps' components. Nevertheless, we find that there are great opportunities for Binder capabilities and those solutions to complement each other.

Type enforcement: SELinux' security types are assigned to processes and not UIDs and hence restrict each process' access rights. Research, such as *FlaskDroid* [15], has shown how this type enforcement can be extended into the application framework services and apps, also providing a per-process permission enforcement. But, SELinux (or mandatory access control in general) was not designed to support efficient app compartmentalization but to enforce static system policies that are highly inflexible in practice and require administrative intervention for updates. Moreover, SELinux types do not scale as efficiently as capabilities, e.g., every possible permission combination would require a dedicated security type, easily causing a type explosion. Nevertheless, mandatory access control (with SELinux on Android) has been considered as a strong building block for capability systems [65], [38], which enforces basic information flow policies that can be statically analyzed and that harden the trusted computing base for capability management. Also in our integration of Capsicum, we rely on the beneficial combination of capabilities and SELinux (see Section IV-E).

IFC: Different works (e.g., [45], [46], [70], [31]; see Section III) address the intricate problem of controlling propagation of sensitive data once it was released to another, data processing app and preventing accidental or deliberate leakage of that data. Data flow labels, in most cases floating labels, on apps (components) or processes determine which data the app (component) or process can access or to which sinks data can be leaked. The supported policies of those solutions are usually very flexible. However, the goal of controlling *information flow* differs from the *access control* a capability system aims at and, hence, the existing IFC solutions do not fit well for the setting we are concerned with. For instance, delegation of access rights in the IFC solutions is often coupled to label propagation on explicit data flows but does not fit well to delegating or even representing access rights to call system services. Further, labels are used to separate different security contexts in memory and on storage for non-interference. Changing the labels (e.g., endorsement) is usually tied to restarting the app (component) or process, or poly-instantiating the app (component) or process. We think controlling information flows *after* data has been released is a very valuable complement to an access control system like Binder capabilities that governs *if* data should be released or accessed, and it might be interesting

to explore in the future to which extent Binder capabilities can further enhance IFC solutions. Moreover, supporting inter-component IFC within an app can also motivate developers to consider least-privilege data dissemination between their apps' components and add to the encouragement for app compartmentalization by DROIDCAP.

App virtualization: App virtualization solutions like *Boxify* [9] put a reference monitor between a sandboxed app and the system. The primary goal of app virtualization is to provide app sandboxing without modifying the app or the system. App virtualization differs first of all in its declared purpose from DROIDCAP, since it does not encourage app compartmentalization by app developers but declares app developers as the opponent whose apps have to be constrained. Further, while Boxify allows enforcement of powerful policies on the app's interaction with other apps, system services, or the file system, its design comes also with inherent performance issues for which object-capabilities could be an easy way out. For instance, Boxify has to proxy all file-system access by a sandboxed app, since sandboxed apps are isolated processes, which adds a complete IPC round-trip time to the syscall. Using capabilities, access to the app's private directory—a subdirectory of the Boxify app's private directory—can be simply delegated to the sandboxed process. Similarly, if Boxify does not need to enforce policies on an app's access to system services, it can simply delegate a Binder capability to the sandboxed app to allow it to access the system service without the additional round-trip via the Boxify app. Furthermore, Boxify's highly-privileged Broker by-design violates the principle of least privilege to be able to mediate any potential app.

General attacker model: Lastly, it should be emphasized that DROIDCAP currently primarily targets app developers that want to design their apps more defensively by compartmentalizing them or privilege-separating untrusted code. But, like other compartmentalization solutions like [18], [29], [52], [59], [71] our attacker model does not include malicious developers. DROIDCAP by itself cannot prevent malicious or colluding [40], [13] apps, although it prevents unauthorized delegation of capabilities between apps.

VII. CONCLUSIONS

We presented DROIDCAP, a retrofitting of Android's Binder IPC mechanism to establish Binder capabilities that associate each IPC handle with the access rights to the referenced remote process. We integrated Binder capabilities into Androids' app management, such as requesting Binder handles to application framework services/apps, and permission enforcement, i.e., using the Binder capabilities for enforcing permissions. Additionally, we complemented DROIDCAP with a prototypical integration of Capsicum providing the foundation for file-system permissions as capabilities. We presented our approach for putting apps into capability mode, a non-trivial task that required integration with Android's zygote and SELinux. As a result, we created a permission enforcement that allows per-process permissions that can be easily delegated to create new protection domains, and hence opened a path to efficiently adopt app compartmentalization and least-privilege operation. In particular, our solution removes the UID-based ambient authority of Android's stock design. The key observation that our DROIDCAP conveys is that the design of modern (mobile) systems is highly amenable to capability based access control and that capabilities can support defensive mobile app development better than the current permission model.

To make capabilities and app compartmentalization more accessible to app developers, future work should investigate how developers can be supported (e.g., tools and frameworks), in particular in light of the inherently modular design of mobile systems. Furthermore, our current design makes a trade-off between pure object-capabilities and backwards compatibility to Android's permissions, and moving Android to a *pure* object-capability system might be worthwhile.

Acknowledgements. We like to thank our anonymous reviewers for their valuable comments and Lucas Davi for shepherding this paper. This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy, and Accountability (CISPA) (VFIT/FKZ: 16KIS0345).

REFERENCES

[1] "Android manifest file: Service," https://developer.android.com/guide/topics/manifest/service-element, last visisted: 05/03/2018.

[2] "AndroidManifest.xml (android-8.1.0_r23)," https://android.googlesource.com/platform/frameworks/base/+/android-8.1.0_r23/core/res/AndroidManifest.xml, last visited: 05/03/2018.

[3] "Capsicum userspace library," https://github.com/google/capsicum-test/tree/dev/libcaprights, last visisted: 07/07/18.

[4] "Security behavior changes," https://developer.android.com/about/versions/pie/android-9.0-changes-all, last visisted: 06/08/2018.

[5] "Storage - Android Open Source Project," https://source.android.com/devices/storage/, last visisted: 06/08/2018.

[6] "platform.xml," https://android.googlesource.com/platform/frameworks/base/+/master/data/etc/platform.xml, 2017, last visisted: 05/03/2018.

[7] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "SoK: Lessons Learned From Android Security Research For Appified Software Platforms," in *IEEE S&P*, 2016.

[8] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *USENIX Security*, 2016.

[9] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *USENIX Security*, 2015.

[10] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard - enforcing user requirements on Android apps," in *TACAS'13*, 2013.

[11] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *ACM SPSM*, 2015.

[12] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," in *IEE MoST*, 2013.

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

[14] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on Android," in *ACM SPSM*, 2011.

[15] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *USENIX Security*, 2013.

[16] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *WISA*, 2014.

[17] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-related policy enforcement for Android," in *ISC*. Springer, 2010.

[18] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha, "Secure integration of web content and applications on commodity mobile operating systems," in *ASIACCS*, 2017.

[19] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[20] D. Drysdale, "Capsicum object-capabilities on linux," https://github.com/google/capsicum-linux, last visisted: 07/07/18.

[21] W. Enck, D. Octeau, P. McDaniel, and C. Swarat, "A study of android application security," in *USENIX Security*, 2011.

[22] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE SP*, vol. 7, no. 1, pp. 50–57, 2009.

[23] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, no. 7, pp. 403–412, Jul. 1974.

[24] Google, "Fuchsia is not linux: A modular, capability-based operating system," https://fuchsia.googlesource.com/docs/+/master/the-book/, 2018, last visited: 04/23/2018.

[25] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *WiSec*, 2012.

[26] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with soaap," in *ACM CCS*, 2015.

[27] H. Hao, V. Singh, and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android," in *ASIACCS*, 2013.

[28] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "Ibm system/38 support for capability-based addressing," in *IEEE ISCA*, 1981.

[29] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android," in *ACM CCS*, 2017.

[30] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android," in *ACM SPSM*, 2012.

[31] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-time enforcement of information-flow properties on android," in *ESORICS*, 2013.

[32] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *ACM CCS*, 2014.

[33] K. C. Kahn, W. M. Corwin, T. D. Dennis, H. D'Hooge, D. E. Hubka, L. A. Hutchins, J. T. Montague, and F. J. Pollack, "imax: A multiprocessor operating system for an object-based computer," in *ACM SOSP*, 1981.

[34] P. A. Karger, "Improving security and performance for capability systems," Ph.D. dissertation, University of Cambridge, Oct. 1988.

[35] D. Kilpatrick, "Privman: A library for partitioning applications," in *USENIX Annual Technical Conference, FREENIX Track*, 2003.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *ACM SOSP*, 2009.

[37] B. W. Lampson and H. E. Sturgis, "Reflections on an operating system design," *Commun. ACM*, vol. 19, no. 5, pp. 251–265, May 1976.

[38] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Comput. Surv.*, vol. 8, no. 4, pp. 409–445, Dec. 1976.

[39] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *ACSAC*, 2011.

[40] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *ACSAC*, 2012.

[41] V. S. Martin Georgiev, Suman Jana, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *NDSS*, 2014.

[42] J. H. Morris, Jr., "Protection in programming languages," *Commun. ACM*, vol. 16, no. 1, pp. 15–21, Jan. 1973.

[43] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A distributed operating system for the 1990s," *Computer*, vol. 23, no. 5, pp. 44–53, May 1990.

[44] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *IEE MoST*, 2015.

[45] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC enforcement on android," in *USENIX Security*, 2016.

[46] A. Nadkarni and W. Enck, "Preventing accidental data disclosure in modern operating systems," in *ACM CCS*, 2013.

[47] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android permission model and enforcement with user-defined runtime constraints," in *ASIACCS*, 2010.

[48] R. M. Needham and R. D. Walker, "The cambridge cap computer and its protection system," in *ACM SOSP*, 1977.

[49] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A provably secure operating system: The system, its applications, and proofs," *Computer Science Laboratory Report CSL-116, Second Edition, SRI International*, 1980.

[50] M. Ongtang, K. R. B. Butler, and P. D. McDaniel, "Porscha: policy oriented secure content handling in android," in *ACSAC*, 2010.

[51] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," in *ACSAC*, 2009.

[52] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *ASIACCS*, 2012.

[53] N. Provos, "Improving host security with system call policies," in *USENIX Security*, 2003.

[54] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *USENIX Security*, 2003.

[55] D. Redell and R. Fabry, "Selective revocation of capabilities," in *Proc. International Workshop on Protection in Operating Systems*, 1974.

[56] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *USENIX Security*, 2013.

[57] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: enforcing in-app privilege separation in android," in *NDSS*, 2016.

[58] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *ACM SOSP*, 1999.

[59] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *USENIX Security*, 2012.

[60] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *NDSS*, 2013.

[61] S. Son, G. Daehyeok, K. Kaist, and V. Shmatikov, "What mobile ads know about mobile users," in *NDSS*, 2015.

[62] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *IEE MoST*, 2012.

[63] Trustworthy Systems Team (Data61), "seL4 reference manual v. 7.0.0," https://sel4.systems/Info/Docs/seL4-manual-7.0.0.pdf, Sep. 2017.

[64] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in Android," in *ACM CODASPY*, 2014.

[65] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: practical capabilities for unix," in *USENIX Security*, 2010.

[66] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE S&P*, 2015.

[67] M. V. Wilkes, *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1979.

[68] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Commun. ACM*, vol. 17, no. 6, pp. 337–345, Jun. 1974.

[69] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for Android applications," in *USENIX Security*, 2012.

[70] Y. Xu and E. Witchel, "Maxoid: Transparently confining mobile applications with custom views of state," in *EuroSys*, 2015.

[71] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *ACSAC*, 2013.

[72] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh, "Taming information-stealing smartphone applications (on Android)," in *TRUST*, 2011.