Scippa:
System-Centric IPC Provenance on Android

Michael Backes, Sven Bugiel, Sebastian Gerling
{backes,bugiel,sgerling}@cs.uni-saarland.de

Saarland University/CISPA, Germany

Abstract

Google's Android OS provides a lightweight IPC mechanism called Binder, which enables the development of feature-rich apps that seamlessly integrate services and data of other apps. Whenever apps can act both as service consumers and service providers, it is inevitable that the IPC mechanism provides message receivers with message provenance information to establish trust. However, the Android OS currently fails in providing sufficient provenance information, which has led to a number of attacks.

We present an extension to the Android IPC mechanism, called Scippa, that establishes IPC call-chains across application processes. Scippa provides provenance information required to effectively prevent recent attacks such as confused deputy attacks. Our solution constitutes a system-centric approach that extends the Binder kernel module and Android's message handlers. Scippa integrates seamlessly into the system architecture and our evaluation shows a performance overhead of only 2.23% on Android OS v4.2.2.

1. INTRODUCTION

Smartphone operating systems allow end-user customization of the phone's functionality with 3rd party apps. To make this extensibility possible and to simultaneously protect the end-user's privacy, current designs of smartphone operating systems exhibit a complex combination of sandboxing-based privilege-separation and extensive message-based data-sharing. The popular Android OS—the focus of this paper—facilitates the integration of remote services and data into an app using a very lightweight IPC mechanism called *Binder*, which forms the primary channel for inter-app communication. To realize privilege-separation between apps, Android implements app sandboxing by assigning each app a distinct user ID under which the app's processes are executed. To implement the least-privilege principle, *Permissions*, i.e., privileges, are assigned to UIDs. Android ships with a set of pre-defined permissions to protect the Android application framework API, for example, reading the user's address book or retrieving the device's geolocation. It further allows app developers to define custom permissions to protect their apps' interfaces. Protecting an app interface with permissions is realized by 1) statically declaring in a manifest file which permissions are required from a caller to successfully access each app component; or by 2) performing runtime checks in the app components using IPC provenance information provided by Binder, i.e., the calling process' UID.

ACSAC'14, December 08–12, 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-6/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664264.

However, Android's current solution to protecting app interfaces is unsatisfactory. Statically declaring the required (custom) permissions for interacting with app components reduces the scope of access checks to the permissions each app holds and excludes more app-specific information such as the developer ID or package name. This solution is not scalable and makes it virtually impossible to flexibly endorse specific apps for component access: permissions can be either requested by any app or require apps to be signed with the same key, which in turn would require a more flexible public key infrastructure that does not exist. Performing runtime checks to protect app components, on the other hand, is more flexible and allows more fine-grained access control, but requires that the IPC mechanism provides message provenance information to app components. Android's system model does *not* fulfill this requirement for sufficient IPC provenance information for all app component types. While it provides the caller UID to *ContentProvider* and *Service* components, it fails in providing this information to components that are receivers of Intent messages–the most prominent inter-app communication mechanism. Prior work [22, 2, 3, 7, 4] has identified different attacks that can occur as a result of this shortcoming, most prominently *confused deputy attacks* [11].

Contributions. In this paper, we identify the technical root-cause for this shortcoming in providing IPC provenance information—a mismatch between Android's concept for inter-application communication at its middleware layer and at its kernel layer. In this multi-layered communication framework, the Binder kernel module is responsible for providing the sending process' user ID to the receiving process. However, *logical* communication occurs between app components—in the literature commonly referred to as *Inter Component Communication* (ICC) [7, 8]—using different abstraction levels of Binder IPC at Android's middleware layer. These abstractions introduce indirections and message dispatching that cause Binder's IPC provenance information (i.e., the sender UID) to be lost along the ICC control flow between app components.

We then present Scippa, our extension to Android's inter-application communication framework, to remedy this shortcoming of Android's architecture. Scippa builds Binder IPC call-chains for ICC control flows and thus provides the required provenance information to apps. Although Quire [4] first identified the need for provenance information on Android, Scippa is, to the best of our knowledge, the first approach that *directly* addresses the mismatch between the middleware and the kernel-level security design in Android's multi-layered inter-application communication framework.

At the core of Scippa is an extension to the Binder kernel module, which constructs and forwards IPC call-chains across distinct application processes. The kernel module extension is complemented by extensions to the message handling routines in Android's application libraries to propagate call-chains across all components of an app. In contrast to Quire's prototypical implementation, which requires app developers to explicitly pass on call-chain information during ICC, our extensions integrate seamlessly into the system architecture and call-chains are established transparently to apps and app developers. Only when developers want to retrieve call-chains, they must be aware of a new system API.

Scippa enables for the first time determining the ICC caller ID within all types of Android app components. For instance, apps can now identify the sender of received broadcast Intents and thus distinguish spurious notifications from benign ones; they can also detect if a security-sensitive Activity was invoked from a trustworthy caller. This allows apps in general a more fine-grained, flexible self-governing of their interaction with other apps and provides the means to effectively mitigate recently reported attacks such as confused deputy attacks [11, 22, 3]. Additionally, we present and discuss changes to Android's app model to enable the return of finalized call-chains to the sending app. Providing information about how their messages were distributed, both by the system and other apps, gives senders the means to *detect* spurious or malignant distribution of their messages (e.g., message hijacking [2]).

The evaluation of our prototype implementation for Android v4.2.2 shows that the performance overhead imposed on Binder IPC messages is only 2.23% and thus does not impede the overall system performance.

2. BINDER-BASED INTER-APP COMMUNICATION ON ANDROID

We first provide the necessary technical background for Scippa. We introduce the Android OS and describe how it uses Binder-based IPC for different types of inter-app communication. In particular, we explain how Binder transactions are integrated into Android's security design.

2.1 Primer on Android

Android is an open-source software stack for embedded devices. The lowest level of this stack consists of a slightly modified Linux kernel that is responsible for basic services such as memory management, device drivers, or inter-process communication (IPC). On top of the Linux kernel lies the extensive Android middleware: it consists of native libraries (e.g., SSL), the Android runtime with the Dalvik Virtual Machine, and the application framework. The middleware implements the majority of Android's application API, which is complemented by pre-installed system apps at the application layer. The API can be extended with 3rd party apps on top of the software stack.

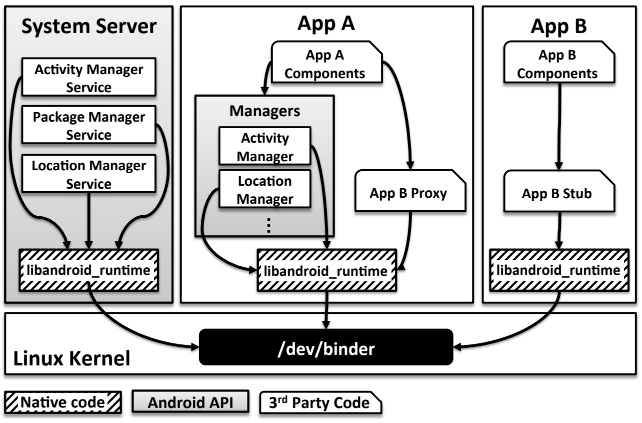

Figure 1: Binder-based inter-component communication.

Android apps are composed of different components and each app is sandboxed by executing it in a separate process with a distinct user ID (*UID*) and assigning it a private data directory on the filesystem. The four basic app components are *Activities* (GUI for user interaction), *BroadcastReceivers* (mailbox for broadcast *Intent* messages), *ContentProviders* (SQL-like data management), and *Services* (long term operations without user interaction). All components can be interconnected remotely across process boundaries by using different abstractions of Binder IPC. These interconnections are commonly referred to as *Inter-Component Communication (ICC) [8]*. To achieve privilege separation between app and realize the least-privilege principle, Android introduces *Permissions*, i.e., privileges that an app must have been granted by the user at install-time to access security- and privacy-sensitive resources.

2.2 Binder-based ICC

Although Android builds on top of a Linux kernel that provides *"classical"* channels such as files or sockets, the primary IPC mechanism on Android is *Binder*. In the following we take a top-down approach to ICC in Android. We show how Android uses ICC and explain afterwards how ICC is implemented as Binder transactions. We refer to external documentation [25] for more details on Binder.

2.2.1 Using Binder-Based ICC on Android

Figure 1 provides a high-level overview of standard Binder IPC in Android when used for connecting components of different apps. Apps can, for instance, either contact system services such as the *Location Manager Service* or communicate directly with each other. All Inter-Component Communication (ICC) builds on top of Binder IPC. User space processes can communicate with each other over Binder IPC via the Binder kernel module that is exposed through the `/dev/binder` sysfs entry. For inter-component communication, the *libandroid_runtime* library, included in all apps, includes an implementation of the Binder communication protocol. Moreover, since application developers usually do not want to deal directly with the low-level mechanics of inter-process communication, Android's design provides different levels of abstraction for Binder IPC. These allow developers to easily make use of Binder IPC at the application level to connect different apps' components (cf. Figure 1 for Stubs, Proxies and Managers).

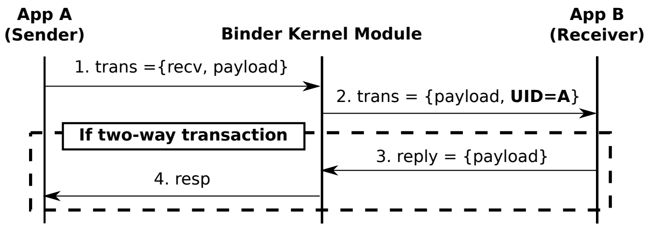

Figure 2: Binder transaction protocol.

Stubs and Proxies. The most basic level of abstraction of Binder IPC are STUBS and PROXIES, which implement remote procedure calls (RPC) via Binder IPC. A PROXY at the caller-side marshals the method parameters into primitive data types and transfers them via IPC to the recipient, where STUB unmarshals the primitives into the original parameters and calls the actual method.

System Services and Managers. MANAGERS are part of the SDK and encapsulate pre-compiled PROXIES for system apps and services like the Location Manager Service that implement the Android application framework API.

Intents. The highest level of abstraction are so-called *Intent* messages. An Intent is a data structure used to provide an abstract description of an operation to be performed by its receiver(s). Common usages of Intents include starting Activity components or broadcasting notifications to apps. Since the sender of an Intent can both explicitly state the target component and implicitly define potential receivers through a description of the intended action, the actual target app(s) must be resolved at runtime. This is the task of the *ActivityManagerService*, which relays all Intents.

2.2.2 Binder Transactions and Integration Into Android's Security Design

Before we explain how Binder acts as a building block in Android's security architecture, we first explain how Binder conducts transactions between two app processes. Figure 2 illustrates abstractly a transaction between *App A* (sender) and *App B* (receiver). To initiate the transaction, *A* writes its transaction data via `/dev/binder` to the Binder kernel module (step 1). The transaction data contains a token (`recv`) identifying the communication peer (i.e., *B*) as well as some payload containing, e.g., the method ID to be executed by the receiver plus the method arguments (e.g., an Intent object). The kernel module then resolves the token to identify the recipient of the transaction, i.e., *B*, and extends the transaction data with the sender UID, i.e., UID of *A*. Afterwards, the module copies the transaction data into the user space of an *IPC Thread* selected from the IPC thread pool of *B* (step 2). If the caller expected a reply (*two-way transaction*), the reply is sent back to the caller via the kernel module (steps 3 and 4). A two-way transaction is implemented as a *closed wait*, i.e., the sender thread blocks until it receives a response and the kernel module ensures that this response originates from the receiver thread.

Providing the sender UID to the receiving component is pivotal for enforcing permissions in Android's security design. First, system services and apps, which implement the application framework API, use this information to perform runtime checks (i.e., `PackageManager.checkPermission(permission, uid)`) whether calling apps hold the required permissions to access their interfaces. Using runtime checks enables these services to enforce permissions flexible and at the granularity of interface functions instead of app components. For instance, the Location Manager Service requires either the Permission `FINE_LOCATION` or `COARSE_LOCATION` depending on the parameters passed to the API call. Second, system services like the *Activity Manager Service* mediate between caller and callee applications (e.g., when an app queries a remote ContentProvider or when delivering an Intent) and these system services use the caller UID when mediating to check whether the caller is allowed to access the callee.

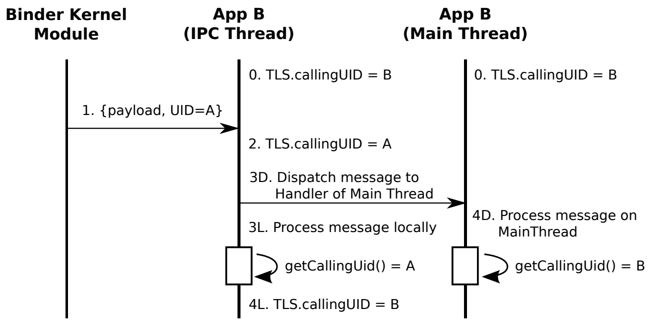

Figure 3: Handling of Binder transactions by the receiver and availability of caller UID.

3. PROBLEM DESCRIPTION

As explained in Section 2.2.2, Binder provides IPC message recipients with the UID of their *direct* caller. However, as shown in Section 2.2.1, Android introduced different abstraction layers for Binder IPC to enable an *inter-component* communication on top of the inter-process communication. These abstractions introduce *indirections* and *message dispatches* that cause the caller UID provided by Binder to be lost along ICC control flows or to be insufficient.

3.1 Message Dispatching

To understand how the caller UID provided by Binder can be dropped in ICC, one first needs to examine how an incoming IPC transaction is handled at the receiver side. Figure 3 extends Figure 2 after the receiver (*App B*) has received the transaction (Step 2 in Figure 2 and Step 1 in Figure 3). The IPC thread of *App B* that is selected to handle this incoming transaction copies the sender UID of the transaction (i.e., *UID=A*) into its thread-local storage (TLS; step 2). Any application code executed on this thread can query this sender UID through the `Binder.getCallingUid` function. However, if the thread is *not* processing a Binder transaction such that the TLS gets never updated, this function call defaults to the threads own information (e.g., `Binder.getCallingUid` would return the UID of the thread itself, as set in step 0).

A received transaction can be processed in one of two possible ways and depends on the targeted component type: First, the message could be handled locally in the context of the IPC Thread (step 3L), which preserves the IPC provenance information. A *Service* or *ContentProvider* component, for instance, would be executed by default in this context. As a consequence, these components can call `Binder.getCallingUid` at any time and retrieve the process UID that triggered their current execution. As explained in Section 2.2.2, this is pivotal for enforcing the default permissions in Android's

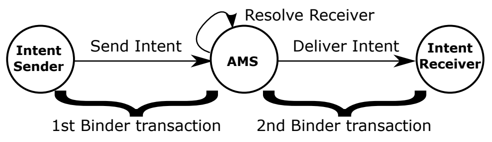

Figure 4: Indirection in Intent-based ICC.

system services. Once the execution of app code on the IPC thread has finished, this thread resets its TLS (step 4L).

Alternatively, the IPC Thread can dispatch the processing of a received message to a worker thread (by default the application's *Main Thread*[1]). The message is dispatched by means of Android's *Handler* mechanism[2] (step 3D). For instance, this worker thread typically performs the handling of received Intents, which includes the processing of received Broadcast Intents, executing IntentServices, as well as Activity-related Intents (step 4D). However, the worker thread is executed in a different context and naturally with a different TLS. Thus, calling `Binder.getCallingUid` on the worker thread will always result in the retrieval of the worker thread's process UID. Dispatching the message will, therefore, effectively drop IPC provenance information: Components executed on the application's worker thread will have no information about which process has triggered their current execution and, hence, cannot distinguish whether their execution is legitimate and whether they can trust any received payload.

3.2 Indirect Communication

Android's model for ICC also introduced indirect communication between components, which renders Binder's IPC provenance information insufficient, in particular in the case of Intent-based ICC. As mentioned in Section 2.2.1 and illustrated in more detail in Figure 4, the *ActivityManagerService* is responsible for relaying Intents between apps. Thus, the actual communication between Intent senders and Intent receivers consists of two distinct Binder transactions. As a consequence, Binder's IPC provenance mechanism will at the receiver's side always identify the *ActivityManagerService* as the IPC caller, instead of the actual origin of the received Intent (i.e., the UID of the Intent sender). An exception from this shortcoming are Activity-related Intents that require a return value from the receiver. In that specific case, the Intent receiver can request the sender ID from the *ActivityManagerService*.

3.3 Provenance Information vs. Permissions

Many attacks [22, 7, 3] that have been reported in the Android security literature can be effectively mitigated if the callee is provided with comprehensive IPC provenance information. In rare cases [7], the cause for the discovered vulnerability was simply a forgotten permission check. In general, however, the situation is more complex. The confused deputy attacks presented in [22] relied on a privileged *BroadcastReceiver* that modified the system state (e.g., Wi-Fi or GPS state) on behalf of any broadcast sender. The receiver did not check whether the sender was entitled to send this command, since the current Android design does not provide the technical means that enable the *BroadcastReceiver* to retrieve the caller's UID and to endorse the caller for this privileged command. As a potential fix, a new permission could be introduced to protect this receiver from receiving Intents from unprivileged apps. However, this approach would be very inflexible: it would require a new permission for every distinct privileged receiver, especially when a receiver holds multiple privileges (e.g., Wi-Fi *and* GPS) or the exact privileged operation depends further on message parameters. We assume these to be the reasons why the vulnerabilities mentioned above are still not fixed in Android v4.2.2, although they have been known since v2.2 [22]. In contrast, provisioning comprehensive IPC provenance information as in Scippa provides the means for a flexible and fine-grained access control to (system) app developers and is, thus, preferable to effectively prevent attacks, such as confused deputy attacks [22, 3].

[1]Also referred to as *UI Thread* or *Activity Thread*.

[2]`http://developer.android.com/reference/android/os/Handler.html`

4. REQUIREMENTS ANALYSIS

In this Section, we define our adversary model, derive requirements and discuss challenges for a comprehensive Binder IPC call provenance on Android.

4.1 Adversary Model

The attacker model for our design of Scippa considers a strong attacker that is able to mount confused deputy, Intent hijacking and Intent spoofing attacks.

Confused Deputy Attacks. We adopt the confused deputy attacker model [11] that was adapted to the specific scenario of Android [4, 22]. In this model, a malicious app with an insufficient set of permissions for its malign purpose tricks a privileged app into executing its privileges on behalf of the malicious app. For instance, Enck *et al.* [7] reported the possibility of sending an Intent to the *Phone* app in order to start a phone call without holding the corresponding `CALL_PHONE` permission. Porter Felt *et al.* [22] discovered several *BroadcastReceiver* components in system services that acted as confused deputies and allowed any app to change, for example, the Wi-Fi or GPS status.

Intent Hijacking and Spoofing. In addition to confused deputy attacks, we consider Intent hijacking and Intent spoofing attacks [2]. To mount an Intent hijacking attack, the attacker registers an app in the system such that (ordered) broadcast Intents or certain Activity-related Intents are delivered to this registered app instead of to the actually intended recipient. Thus, the malicious app is able to receive any information contained in the Broadcast. Additionally, in the context of Activities, the attacker can use this technique to mount phishing attacks.

Restrictions. We focus in this work exclusively on *Binder IPC* and exclude other IPC channels such as files or sockets. A solution for these alternate channels would require additional extensions to low-level services such as the filesystem. Moreover, we limit our solution to *direct* IPC and do not consider covert channels as leveraged in collusion attacks [24, 16]. Finally, we do not explicitly consider attacks that compromise the system integrity such as root exploits or attacks against system components. However, we consider 3rd party apps to be in full control of their sandbox. Apps can include native code that is able rewrite the application code.

4.2 Requirements and Challenges

In this section we derive the necessary requirements for Scippa and elaborate on technical and conceptual challenges in context of the Android system model.

Availability of Provenance Information. Due to message dispatching, provenance information is currently only available to app components that are executed in the context of a receiving IPC thread. Thus, the first requirement for a comprehensive solution is to extend Android's app model to propagate IPC provenance information to all app components. A particular challenge to be addressed in this context is to identify the correct IPC context of each thread. The Main Thread, for instance, usually handles workloads dispatched by multiple IPC threads. Hence, its current IPC context depends on the IPC thread it is currently serving.

Building System-Centric IPC Call-chains. By default, Binder provides apps with the UID of their direct caller. However, when considering the indirections in Android's Intent-based ICC, this information is insufficient for callees to identify the initiator of incoming requests. This leads to the second requirement: it is necessary to establish system-centric call-chains for Binder IPC so that we can provide receivers of Binder transactions with provenance information. This would enable receivers to answer questions like *"Who sent this Intent?"*.

Returning Call-Chains to Senders. While Android provides the receiver of a Binder transaction with limited means to retrieve the sender's UID, it does not provide any feedback to the sender on how their message was handled in the system. This missing feedback makes the senders unaware of, e.g., Intent hijacking and phishing attacks [2]. The third requirement is, therefore, to establish a feedback mechanism for IPC senders by returning already established, finalized call-chains to them. This enables the senders to analyze how their message was handled both by the system and other applications and, thus, to *detect* potential hijacking and phishing attacks. A technical challenge is to efficiently address branching of call-chains, which leads to a *1:N* communication (e.g., when broadcasting an Intent).

Tagging Asynchronous Messages. Although Binder transactions are synchronous, the protocols and mechanisms Android deploys on top of Binder can be asynchronous. For instance, *sticky* Broadcast Intents are kept in the system and are delivered even to recipients that register *after* the broadcast was sent. Thus, to effectively fulfill the first three requirements, this asynchronicity needs to be addressed, e.g., by tainting asynchronously delivered messages with their associated IPC provenance information.

5. SYSTEM-CENTRIC IPC CALL-CHAINS

In this section, we describe our extensions to Android's Binder IPC at the kernel and user space to establish call-chains along *direct* ICC control flows. Further, we elaborate on extensions to Android's message handling mechanism to propagate those call-chains between the threads of an app.

5.1 Establishing Call-Chains

At the core of our solution are extensions to the Binder kernel module. In Binder, IPC messages are passed between processes as Binder transactions. Our extensions construct call-chains across app processes by linking the transactions along a direct thread of control for inter-application communication. Figure 5 illustrates recursive Binder transactions between three apps *A*, *B*, and *C*. For instance, *App A* could be an Intent sender, *App B* the *ActivityManagerService*, and *App C* the Intent receiver (cf. Figure 4). Moreover, Figure 5 shows that we differentiate in our design between *two-way*

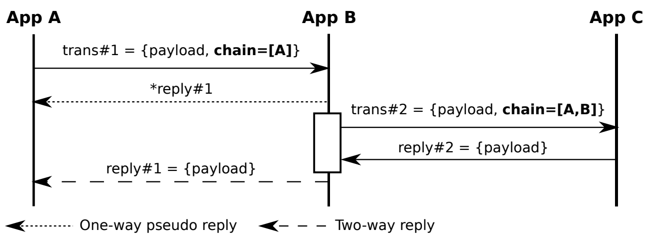

Figure 5: Call-chains during recursive Binder IPC calls.

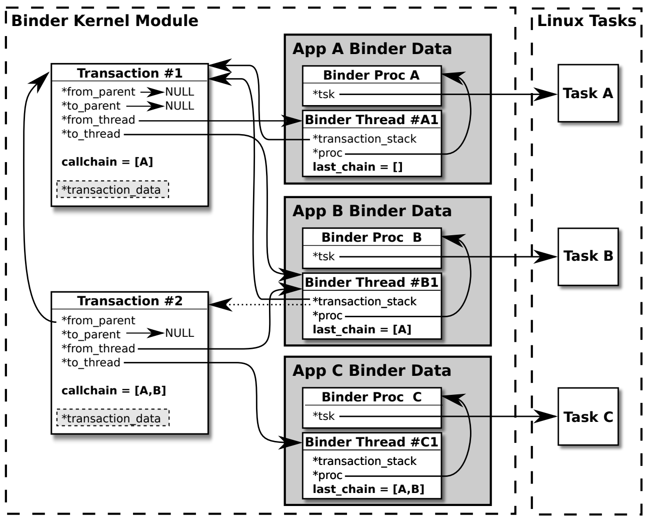

Figure 6: Constructing call-chains in recursive transactions. The dotted line represents the pointer *after* Transaction#2.

and *one-way* transactions. In recursive two-way transactions the second transaction `trans#2` is nested in the first transaction `trans#1`. In contrast, in one-way transactions `trans#1` is finished before `trans#2` is triggered, as illustrated by the pseudo reply `*reply#1`.

Binder transactions. The operations performed within the Binder kernel module in this scenario are depicted in Figure 6. In general, for each process that registers with Binder as a sender/receiver, the kernel module sets up `Binder Proc` information associated with the process as a whole and `Binder Thread` information associated with a particular thread of the application process (i.e., threads from the app's Binder IPC thread pool and the main thread). When *App A* sends a message to *App B*, the Binder kernel module creates new transaction data `Transaction#1`, where `transaction_data` contains the message, `from_parent` and `to_parent` point to parent transactions at the sender's and receiver's side in case of recursive transactions, and `from_thread` and `to_thread` point to the sender's and receiver's involved `Binder Threads`. The `transaction_stack` attribute of `Binder Thread` points to the last processed (i.e., last sent or received) transaction of the associated thread. When an IPC thread of *App B* is ready to receive this transaction, the message is copied from the kernel to the thread's user space. At this point the kernel module provides the receiver thread with the sender UID (cf. Step 4 in Figure 2), which is retrieved from the kernel task information associated with the transaction sender's `Binder Proc`.

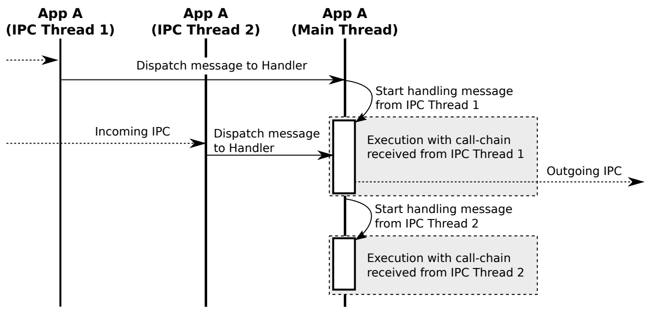
App A (IPC Thread 1)
App A (IPC Thread 2)
App A (Main Thread)
Dispatch message to Handler
Start handling message from IPC Thread 1
Incoming IPC
Dispatch message to Handler
Execution with call-chain received from IPC Thread 1
Outgoing IPC
Start handling message from IPC Thread 2
Execution with call-chain received from IPC Thread 2

Figure 7: Message processing by Handler on Main Thread.

Constructing Call-Chains in Two-Way Transactions. As shown in Figure 5, *App B* issues a recursive call to *App C* while handling the call from *App A*. In Figure 6 this is depicted as `Transaction#2`. The attributes of `Transaction#2` are set the same as those of `Transaction#1`. However, since `Transaction#2` is the most recent transaction for *App B*, its `transaction_stack` pointer is adjusted accordingly. The dotted line represents the pointer *after* `Transaction#2`.

In SCIPPA, we attach the current state of the call-chain to each transaction data. To this end, we extended the `transaction` data structure with new attributes to hold a call-chain. Every time a recursive call is made, the kernel module copies the call-chain from the direct predecessor transaction to the new transaction (or starts a new call-chain if no predecessor exists) and appends the UID of the current sender to the call-chain. In addition to the UID, we also save the sender's process and thread ID in the call-chain and assign every new call-chain a unique ID.

Constructing Call-Chains in One-Way Transactions. For one-way transactions, the mechanism depicted in Figure 6 is not applicable, since `Transaction#1` is freed before `Transaction#2` is allocated. Thus, when continuing the chain, the data structure for the direct predecessor of `Transaction#2` no longer exists. To establish this missing link between transactions, we save the last received call-chain as part of the `Binder Thread`. When the application code executing on the IPC thread sends the next transaction (i.e., transaction *B-C*), the saved call-chain is continued. This solution is based on the observation that IPC threads are at any given time either executing app code as direct consequence of the last received IPC call, or, when this execution has finished, return to a state in which they can receive the next call and hence next call-chain.

5.2 Intra-App Call-Chain Propagation

As explained in Section 3, certain types of IPC messages are dispatched by the receiving IPC thread to the Main Thread (or a dedicated worker thread) by adding it to the Main Thread's Handler message queue. The Main Thread handles queued messages sequentially (cf. Figure 7). Thus, the current IPC context of the Main Thread directly depends on the message currently processed. To propagate the call-chain received by the IPC thread to the Main Thread, we extended the `Message` and `Looper` classes of Android's application libraries to attach the received call-chain to new messages for the Handler. The Main Thread's `Handler` class is extended to always update its current IPC context with the call-chain of the message it processes next. Thus all outgoing IPC during this processing continue the call-chain correctly.

5.3 Asynchronous Call-Chain Propagation

One particular challenge for establishing call-chains that include broadcast Intents are *sticky* broadcasts. As long as the sender app does not cancel the broadcast and does not get uninstalled, sticky broadcasts are stored by the *ActivityManagerService*. These broadcasts are even delivered to relevant Broadcast Receivers that register in the system *after* the broadcast Intent has been sent. To address this asynchrony within the control flow from the Intent sender to the receiver, we modified the *ActivityManagerService* to tag stored sticky broadcasts with the call-chain at the time the broadcast Intent was stored. Additionally, we modified the *ActivityManagerService*'s logic for delivering sticky broadcasts to adjust its current IPC context according to the call-chain stored with the sticky broadcast before sending and to restore its original IPC context afterwards. As a result, sticky broadcasts continue the call-chain so that its receivers can now identify the original broadcast sender.

5.4 Accessing Call-Chains from User Space

To provide the current call-chain information to the user space, we pass this information as part of the `binder_transaction_data` from the kernel to the IPC thread that receives the transaction. We extended the Android runtime library (cf. Section 2.2.1) to extract the call-chain from the transaction data and to subsequently store it in the thread local storage (TLS). From there it can be retrieved by any application code that is executed on the same thread. Similar to the default `getCallingUid` function, we introduce a new API function `getCallChainUids` to retrieve the call-chain. App developers can then retrieve information about those chained UIDs from the system—including the UIDs' package names, developer signatures, or permissions—and implement a fine-grained access control based on that information.

As explained earlier, some scenarios require that the user space is able to set its current call-chain. Therefore, we extended the `Binder` classes with functions to set the call-chain of the current thread. When a new Binder transaction is triggered, the set call-chain is passed to the Binder kernel module as part of the send transaction data. To prevent the user space from forging or modifying call-chains, a token-based approach—i.e., user space processes only hold a *read-only* reference, bound to their UID, to retrieve the call-chain from the kernel space—could enable the kernel to verify that call-chains retrieved from user space were originally created by the kernel and hence to discard illicit chains.

5.5 Returning Call-Chains to Message Senders

Our extension to the kernel module propagates finished branches of call-chains back to the initial sender. We extended the Binder protocol with a new flag `BR_CALLCHAIN` to send a finalized branch of a call-chain back to the app that started this chain. To distinguish branches of different call-chains, the kernel module additionally provides the call-chain ID to the user space. While this mechanism returns all finalized branches to the sender, ongoing work extends Android's application model to efficiently store and manage this information. We are in the process of implementing a new application component type dedicated to this task.

6. EVALUATION

In this section we evaluate and discuss the implementation of SCIPPA in terms of effectiveness and performance impact.

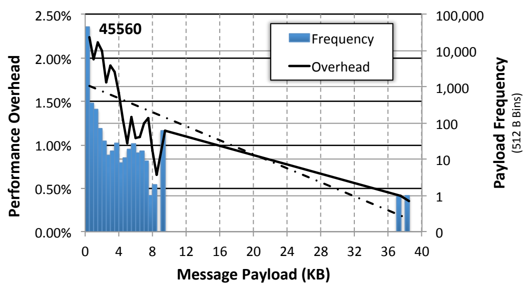

Figure 8: Performance overhead of Binder transactions vs. payload size and frequency breakdown of payload sizes.

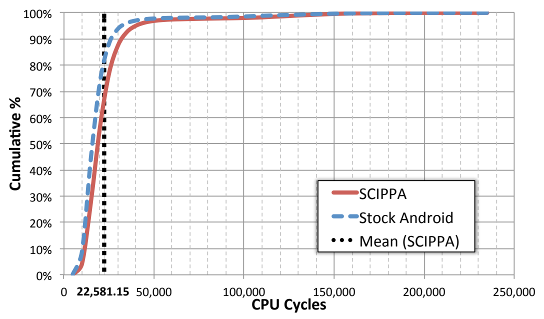

Figure 9: CDF of CPU cycles for Binder transactions on Scippa and stock Android.

6.1 Experimental Methodology

All our experiments were performed on a standard Galaxy Nexus development phone (Dual-core 1.2GHz Cortex-A9 CPU, 1GB RAM). We implemented Scippa as a modification to the Android OS code base of v4.2.2_r1.2b and the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1*. Since the resolution of the Linux time facilities were too coarse grained for our microbenchmarks of Binder transactions, we leveraged the ARM *Performance Monitoring Unit* to calculate the performance overhead in CPU cycles. Since the PMU counts cycles per CPU core on our dual-core platform, we modified the Binder kernel module to acquire a spinlock for the duration of each measurement, thus eliminating the possibility that the current Binder thread is re-scheduled on the other CPU core while executing the measured code segment. Additionally, to be able to estimate the performance overhead in seconds, we adjusted the CPU frequency governor to always clock the CPUs at the maximum rate of 1.2GHz. Finally, to reduce the level of white noise in our measurements, we did not run any other apps except for our benchmark apps. Thus, all measurements approximate the *lower* bound for the actual overhead.

6.2 Performance Impact

To evaluate the performance impact of Scippa, we performed i) microbenchmarks of transactions in the Binder kernel module and ii) reimplemented relevant parts of the user space benchmarks of the closely related work *Quire* [4].

Transaction microbenchmarks. We performed microbenchmarks within the Binder kernel module for building and continuing call chains as described in Section 5. The results of our benchmarks are based on the measurements of 52,777 Binder transactions. Figure 8 presents the relative overhead vs. the data payload of the transaction. The maximum overhead was 2.23% and this overhead decreased with increasing payload size, where the memory copy operations for the data buffer outweigh the call-chain operations. However, when taking the frequencies of different payload sizes into consideration, the weighted average remains at 2.23%.

This small performance overhead is further illustrated in Figure 9, which shows the cumulative frequency distribution of the CPU cycles required for performing Binder transactions in Scippa and stock Android. On average, transactions on stock Android required 18,850 cycles and 99% of the measured transactions required less than 115,000 cycles. On Scippa this performance merely decreases to an average of 22,581.15 cycles per transaction, which translates to 18.82μs in our experimental setup, and 99% of the transactions required less than 130,000 cycles.

It should be noted that 0.58% of the measurements in our data set were extreme outliers that were in the range of 3 to 5 magnitudes higher than the rest of the measurements. These outliers occurred in both stock Android and Scippa benchmarks. We could trace these outliers back to thread blocking during parsing of the binder transaction header, i.e., *operations independent from our* Scippa *modifications*. However, since these rare outliers significantly distorted the mean and the margin of error in our measurements, we excluded them from the results presented here.

User space benchmark [4]. Since applications in Scippa have to retrieve, parse, and set the received call-chain as part of their Binder IPC thread operations (cf. Section 5), we measured the overhead of Scippa from the application layer perspective. To this end and to provide a better comparison with existing work, we re-implemented the test cases presented in the closest related work *Quire* [4]. In this test, the Service components of several test apps interact a) to pass a message with variable size payload roundtrip between two apps and b) to send a message without payload roundtrip between multiple apps to build call-chains of different lengths.

Figure 10 presents the average performance per roundtrip versus the message payload as computed from 11,000 measurements per payload size. The payload size ranges from 0 bytes to 6,336 bytes in 64 bytes increments. In our data set, Scippa imposed between 3.70-25.33% overhead, which is comparable to Quire's performance (21% slowdown).

Figure 11 shows the average performance per roundtrip vs. the call-chain length with a max length of 9 (i.e., 10 apps involved) and 11,000 measurements per length. Scippa's overhead is 12.70-26.73%, which is again comparable to Quire (20-25% slowdown).

6.3 Binder IPC Provenance

In this section, we provide statistics on the call-chains observed in Scippa during our tests and evaluate how well these call-chains provide the necessary IPC provenance information to efficiently mitigate the different attacks introduced in our requirements analysis (cf. Section 4).

Call-chain Statistics. Table 1 summarizes statistics on call-chains observed during our testing. All margins of error are for a 95% confidence. We logged in total 54,670 call-chains with an average length of 1.56. All chains had at least two branches with 2.59 being the average number of branches per chain. Figure 12 provides a breakdown of the branch

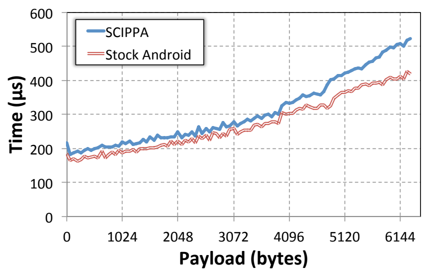

Figure 10: Single Service call roundtrip time vs. payload size.

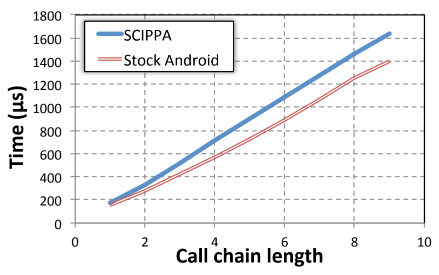

Figure 11: Service call roundtrip time vs. call chain length.

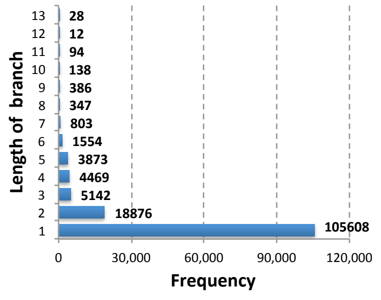

Figure 12: Break down of observed call-chain lengths.

General:		**Branching:**		**Dispatching:**	
#Call-Chains:	54,670	#Chains with branches:	54,760 (100%)	#Chains with dispatching:	3,237 (5.91%)
Chain length:	1.56 ± 0.01	#Branches (total):	141,330	#Dispatches (total):	24,966
Max. length:	13	#Branches (per chain):	2.59 ± 0.08	#Dispatches (per chain):	7.71 ± 1.92
		Max. number of branches:	1,194	Max. number of dispatches:	2,784

Table 1: Call-chain statistics.

lengths and shows that the maximum observed length is 13 and that chains with a length of one are most frequent. The max number of branches for one chain is 1,194. Additionally, 5.91% of all chains contained at least one dispatch between an IPC thread and the Main Thread. On average, each chain contained 7.71 dispatches with 2,784 being the highest number of dispatches observed for one chain.

Attack mitigation. To verify that Scippa fulfills the requirements stated in Section 4.2, we developed a set of interacting test applications, which implement different combinations of inter-component communication that model the scenarios that have been reported to be prone to attacks such as confused deputy attacks [22] or Intent hijacking [2]. This includes common inter-app communications such as starting Activities, Broadcast Intents, or binding and calling Services including IntentServices. In general, all called components were able to retrieve the call-chain of the direct thread of control that lead to their invocation and thus identify the initiator of the call-chain. Also, all senders were successfully notified by the kernel module about finalized call-chain branches. Based on this information, we were able to implement a per-UID access control that is more fine-grained and flexible than Android's static Permission system.

Since the most well-known reported confused deputy attacks rely on Broadcast Receivers [22], we briefly elaborate on call-chains for broadcast receivers in our testbed. Figure 13 shows an established call-chain for a single Broadcast Intent send by the app with UID 10043 (lower left corner). The Intent is sent to the *ActivityManagerService* as part of the system server with UID 1000, where the task to send this broadcast is dispatched to a dedicated thread (1000:403:777 → 1000:403:420). This thread delivers the Broadcast Intent in parallel to all *dynamically* registered receivers (upper left rectangle) and in *order*[3] to all receivers registered statically through the applications' manifests (lower right rectangle). Each app receives the Intent via an IPC thread and processing of the Intents by the Broadcast Receiver components is dispatched to the apps' Main Threads.

As a consequence, each Broadcast Receiver is able to retrieve the branch of the call-chain that lead to its invocation and, hence, to identify the sender of the broadcast. For instance, the receiver of UID 10047 retrieves the call-chain 10043 → 1000 and the receiver of UID 10045 retrieves the chain 10043 → 1000 → 10044 → 1000 that shows all receivers previous to itself in the ordered delivery. Using this information, Broadcast Receivers in Scippa can now efficiently evaluate their trust in received messages and their senders, which allows them to react accordingly by, e.g., refusing to accept spurious messages. In addition to our test cases, we also verified that the privileged Broadcast Receiver that was reported as a confused deputy in [22] is now able to identify the broadcast Intent sender and hence to apply fine-grained access control depending on the Intent payload (e.g., GPS vs. Wi-Fi control commands). That eliminates the confused deputy vulnerability without the need to split its component interface or to introduce new permissions.

Additionally, the sender received from the kernel module four `BR_CALLCHAIN` notifications about the call-chain branches that ended with the apps with UIDs 10045 through 10048. Thus, the sender is able to identify the receivers of its broadcast and to determine if its broadcast was potentially hijacked by an unintended receiver [2]. In case of ordered broadcasts, it can even determine which app was responsible for cancelling the further delivery of the broadcast.

6.4 Discussion and Limitations

The most important limitation for the effectiveness of our approach is that the call-chain can be lost if communication between threads occurs over channels currently not covered by Scippa. With message dispatching, Scippa covers one of the major communication channels between Android application threads, but other channels exist (e.g., `notify`). Future work has to address these channels through extensions to the Dalvik VM and Java language classes in the Android framework. For instance, initial experiments have shown

[3]Broadcast receivers registered through the manifest are *always* served in ordered fashion, but intermediary receivers only can stop further delivery when the *ordered* flag is set.

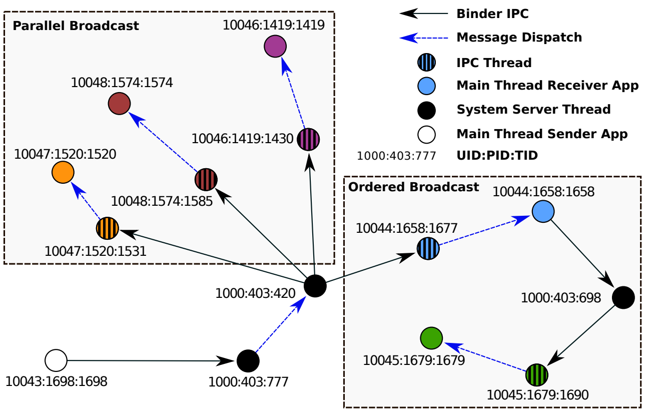

Figure 13: Call-chain for parallel and ordered broadcasts.

that it is possible to forward the call-chain to newly spawned threads, e.g., for *IntentService* components or *AsyncTasks*.

Similarly, SCIPPA currently only covers direct control flows for ICC. Hence, indirect control flows are an open problem. Providing an efficient solution to address indirect control flows is an orthogonal problem and affects other approaches such as dynamic taint-tracking [6] as well.

Since our solution relies on code within the app sandboxes to forward call-chains on message dispatching, this code base is prone to attacks by malicious apps. While forging and modifying call-chains can be prevented by implementing a token-based approach (cf. Section 5.4), the deliberate dropping of call-chains cannot be prevented. However, because UIDs are attached by the kernel to the call-chains during the sending of IPC calls, a malicious app can only hide previous hops in the call-chain but not itself. Thus, a malicious app cannot fool a receiver into trusting it by hiding its predecessors in the call-chain [4] and this is primarily a problem if multiple malicious apps collude [16, 24, 1].

7. RELATED WORK

IPC-based domain isolation. Thread-migrating IPC has been used in high-assurance systems [15, 28, 26] as building block for domain-based isolation by factoring applications into smaller domains. Domains are usually compartmentalized at process boundaries and IPC is used to connect them. The security of IPC has been investigated from different angles, e.g., for synchronous IPC [27] or language-based security [9]. Android's design borrows ideas from this literature (e.g., UID-based app sandboxes, connected via thread-migrating IPC). However, in this paper we identified and addressed misaligned security assumptions when *Inter-Component Communication* is built on top of IPC.

Provenance frameworks. Establishing provenance information has been primarily investigated for data provenance [29, 17] and in distributed systems [30]. Specific to smartphones, the *SPADE* framework [10] has been ported to Android [13]. It uses the Binder debug interfaces to profile the IPC on Android and generate useful traces for device auditing. SCIPPA provides very similar information, however, in contrast to SPADE on Android, SCIPPA also provides the links between IPC channels along direct ICC control flows and thus valuable information for a more detailed auditing.

A recent approach called *EPIC* [21] uses static analysis to detect all potential Intent-based communication channels between application components. The information created by SCIPPA includes this information as well. However, SCIPPA only reflects actual runtime behavior. A combination of these two approaches could lead to a more comprehensive app testing, in which static analysis shows all potential channels and SCIPPA fills the gaps in this analysis (e.g., when a target cannot be resolved statically).

Android security. Research has established a large body of literature on Android security. With respect to preventing confused deputy attacks, related work [22] has proposed the poli-instantiation of apps in ICC to reduce the callee's privileges to the ones of their caller. *XManDroid* [1] monitors all ICC communication and applies at runtime Chinese Wall security policies to prevent communication that could lead to a dangerous information flow. While poli-instantiation and XManDroid rely on strictly restricting privileges or communication channels, SCIPPA and closely related work (*Quire* [4]) rely on provisioning IPC provenance information to callees to enable them to apply fine-grained access control on their ICC interfaces. This allows them to securely provide APIs to other apps. In contrast to SCIPPA, however, Quire's prototypical implementation requires developers to explicitly extend all STUB and PROXY interfaces in order to construct call-chains. SCIPPA abstains from a developer-centric approach for establishing IPC provenance information and implements a system-centric solution that builds call-chains transparently to developers. Moreover, in Quire's developer-centric approach, call-chains are created and extended within the app code and, hence, this approach requires verifiable statements to establish trust in and authenticity of chains. In SCIPPA, the kernel creates and extends the call-chains and only when chains are propagated through the user-space back to the kernel, a lightweight cryptographic mechanism (e.g., tokens) is required to ensure authenticity of chains. Additionally, as a kernel-based solution, SCIPPA covers even cases in which apps do not use STUBs, but instead app code (e.g., native libs) communicates directly with the Binder kernel module.

Besides confused deputy attacks, related work has proposed a solution [14] to mitigate Intent hijacking by applying heuristics-based access control for Intents to prevent their unintended delivery. While this is a system-centric, preventive security extension, SCIPPA provides a different trade-off. SCIPPA adds measures that allow an application to detect (not prevent) such attacks, but in turn provides a higher precision than heuristics due to its call-chain information.

Information flow control. Concepts from decentralized information flow control [20, 19], e.g., as implemented in the DEFCON [18] and Asbestos [5] operating systems, have been applied within different solutions on Android. Most prominent solutions based on dynamic taint analysis include *TaintDroid* [6], *AppFence* [12], and *Paranoid Android* [23]. In contrast, SCIPPA does not aim at restricting information flows of sensitive data at information sinks, but instead aims at providing apps with IPC provenance information that enables them to effectively apply access control for sensitive data and functionality.

8. CONCLUSION

In this paper, we presented SCIPPA, our architecture for provisioning Binder IPC provenance information on Android. It allows app components to identify the sending app of incoming IPC messages despite indirections and message dispatching. Using this provenance information, apps are now able to effectively apply per-sender access control to their interfaces. In contrast to related work, SCIPPA constitutes a system-centric approach that directly addresses conceptual shortcomings in Android's multi-layered inter-application communication. We presented an implementation of SCIPPA based for Android v4.2.2 and the evaluation of our prototype showed that SCIPPA imposes only minimal overhead when compared to stock Android. In addition, we deem the lessons learned from SCIPPA valuable for the design of future multi-layered OS security architectures that rely on thread-migration and that support liberal inter-app communication.

The source code of SCIPPA can be retrieved from `http://infsec.cs.uni-saarland.de/projects/scippa/`.

Acknowledgements

We thank the anonymous reviewers for their comments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA). Sven Bugiel thanks Rose Hoberman for her valuable editorial comments.

9. REFERENCES

[1] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS'12*. Internet Society, 2012.

[2] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys'11*. ACM, 2011.

[3] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *ISC'10*. Springer, 2010.

[4] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security'11*, 2011.

[5] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP'05*. ACM, 2005.

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

[7] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS'09*. ACM, 2009.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[9] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys'06*. ACM, 2006.

[10] A. Gehani and D. Tariq. Spade: Support for provenance auditing in distributed environments. In *Middleware 2012*, volume 7662 of *Lecture Notes in Computer Science*. Springer, 2012.

[11] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[12] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *CCS'11*. ACM, 2011.

[13] N. Husted, S. Qureshi, D. Tariq, and A. Gehani. Android provenance: diagnosing device disorders. In *TAPP'13*. USENIX, 2013.

[14] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in Android. In *SPSM'12*. ACM, 2012.

[15] B. W. Lampson and H. E. Sturgis. Reflections on an operating system design. *Commun. ACM*, 19(5):251–265, May 1976.

[16] C. Marforio, H. Ritzdorf, A. Francillon, and S. Čapkun. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC'12*. ACM, 2012.

[17] P. McDaniel, K. Butler, S. McLaughlin, R. Sion, E. Zadok, and M. Winslett. Towards a secure and efficient system for end-to-end provenance. In *TAPP'10*. USENIX, 2010.

[18] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. Defcon: high-performance event processing with information security. In *USENIX ATC'10*, 2010.

[19] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP'97*. ACM, 1997.

[20] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, Oct. 2000.

[21] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *USENIX Security'13*, 2013.

[22] A. Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security'11*, 2011.

[23] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *ACSAC'10*. ACM, 2010.

[24] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS'11*. Internet Society, 2011.

[25] T. Schreiber. Android Binder – Android Interprocess Communication. `https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf`, 2011.

[26] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The Multics kernel design project. In *SOSP '77*. ACM, 1977.

[27] J. S. Shapiro. Vulnerabilities in synchronous ipc designs. In *IEEE SP'03*, 2003.

[28] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP '99*. ACM, 1999.

[29] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, Sept. 2005.

[30] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Trans. Comput. Syst.*, 12(1):3–32, Feb. 1994.