Secure Multi-Execution in Android

Dhiman Chakraborty
CISPA Helmholtz Center for
Information Security, Germany
Saarland University, Germany
dhiman.chakraborty@cispa.saarland

Christian Hammer
University of Potsdam, Germany
hammer@cs.uni-potsdam.de

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

ABSTRACT

Mobile operating systems, such as Google's Android, have become a fixed part of our daily lives and are entrusted with a plethora of private information. Congruously, their data protection mechanisms have been improved steadily over the last decade and, in particular, for Android, the research community has explored various enhancements and extensions to the access control model. However, the vast majority of those solutions has been concerned with controlling the access to data, but equally important is the question of how to control the flow of data once released. Ignoring control over the dissemination of data between applications or between components of the same app, opens the door for attacks, such as permission re-delegation or privacy-violating third-party libraries. Controlling information flows is a long-standing problem, and one of the most recent and practical-oriented approaches to information flow control is secure multi-execution.

In this paper, we present Ariel, the design and implementation of an IFC architecture for Android based on the secure multi-execution of apps. Ariel demonstrably extends Android's system with support for executing multiple instances of apps, and it is equipped with a policy lattice derived from the protection levels of Android's permissions as well as an I/O scheduler to achieve control over data flows between application instances. We demonstrate how secure multi-execution with Ariel can help to mitigate two prominent attacks on Android, permission re-delegations and malicious advertisement libraries.

CCS CONCEPTS

• **Security and privacy → Information flow control**; **Mobile platform security**; **Information flow control**;

KEYWORDS

Android, Information flow control, secure multi-execution

ACM Reference Format:
Dhiman Chakraborty, Christian Hammer, and Sven Bugiel. 2019. Secure Multi-Execution in Android. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297469

1 INTRODUCTION

Google's Android is currently the most widespread [46] mobile operating system. This has made Android popular beyond the common use-cases for end-users, including multi-purpose devices for work tasks or even deployment in settings with higher security and privacy settings. This has posed a challenge on how to secure data with multiple security domains of distinct trust and access levels. The research community has proposed a range of solutions to domain isolation and also Google begun supporting isolation of data and apps assigned to distinct (work) profiles [19].

However, Android's vanilla design and past solutions to extend Android's security have been primarily concerned with controlling *access* to data. Equally important is the question of how to control the *flow* of data once released to an app—a longstanding challenge. Specifically for Android, approaches to information flow control (IFC) have considered distributed IFC [32], static and dynamic taint tracking [4, 13, 25, 29, 49], control along user-interface flows [33], or control between application components [6, 50, 52]. Unfortunately, in a practical deployment beyond restricted use-cases [33] and with a strong attacker model, those approaches fall short. Relying on security-critical code within (untrusted) application sandboxes [13, 25, 49, 50] has been shown to be very hard if not infeasible to achieve [23]; while frequent inter-app communication, that is encouraged by Android's system, can easily lead to an explosion of dynamic taint labels [32]. Often IFC-based solutions do not incorporate the existing permission system but establish parallel access control. Another paradigm of providing security is by changing the application model to which app developers have to adapt [12]. Those problems are not unprecedented outside Android's specific setting for information flow on commodity systems, as we discuss in more detail in Section 3. Building on past experience from IFC for non-mobile devices, we propose in this paper a new design direction to establish IFC on Android: the comparatively new instrument of *secure multi-execution* (SME) [11] to achieve non-interference of information flows between Android apps. SME already has shown good results in web browsers [11, 21] and programming languages [27].

We present Ariel, an IFC framework in Android that implements secure multi-execution. With Ariel the execution of every application process is labeled with a security level from a defined security lattice. In particular, we extend Android's vanilla design to allow one instantiation of every app per security level (i.e., multiple executions of apps). Ariel is accompanied with a *secure scheduler*, embedded inside Android's framework, and controls the data flow between different instances of multi-executing applications. Realizing the policy enforcement through our scheduler inside the Android framework forms a reliable, non-bypassable reference monitor protected by a strong security boundary and preserves

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297469

backwards compatibility from the app developers' point of view. The security lattice is specified using a partially ordered data flow model [10] and for our prototype presented in this paper we derived the lattice from Android's permission system. Due to the limited number of Android permissions, Ariel does not face the the problem of policy explosion even if the IFC lattice policy is fine-grained. Nevertheless, our design can be easily retrofitted to accommodate other security policies, for instance, deriving security labels from application meta-data like signatures [44]. Using our prototype, we show how Ariel can be helpful in protecting against applications leaking data unintentionally (confused deputies) or application components containing untrusted, over-privileged third party code.

To summarize, this paper makes the following contributions:

- We show how to create a multi-execution environment for any application and describe the technical challenges in reaching this goal.
- We introduce Ariel, an information flow control solution based on secure multi-execution. Ariel's security lattice is derived from Android's permission system and we introduce a secure scheduler as part of Android's Intent-based inter-app communication to direct flows to app instances of the correct security label.
- We provide a proof-of-concept implementation of Ariel and show how Ariel can help mitigate two very well-known attacks, permission re-delegation and over-privileged advertisement libraries.

2 BACKGROUND

In this section, we will briefly explain the technical background on Android and non-interference.

2.1 Android

We start by providing background information on Android's app sandboxing, inter-app communication, and app launch process.

Application Sandboxing. All Android applications and system services are sandboxed and isolated from each other in terms of processing and storage. The sandbox is defined by the Linux UID assigned at install-time, under which all apps' processes are executed, and applications manage their own private data directory. To access resources outside their sandbox, applications have to request *permissions* from the user. For instance, access to the device geolocation or user's data has to be explicitly requested. Most of those permission-protected resources are managed and protected by dedicated system services (e.g., LocationService) and only a handful are managed and protected by the Linux kernel (e.g., low-level Bluetooth functionality or Internet sockets). Android Permissions are further classified into different protection levels, namely: (1) *Normal:* protects data considered harmless for user privacy and does not need any further approval by the user; (2) *Dangerous:* protects privacy-sensitive data (e.g., making a call, taking pictures,and several others) and requires explicit user-approval either at install time (prior to Android v6) or at runtime (since Android v6); (3) *Signature:* can only be granted to applications signed with the same developer key as the app declaring such a permission; (4) *SignatureOrSystem:* similar to *Signature* permissions but are also granted to apps signed with the system OEM key.

Inter-app communication and app components. Android applications can breach the isolation between them in a controlled manner through system-controlled channels, where Binder IPC is the primary inter-app communication channel. To ease the use of inter-app communication via Binder IPC, Android provides a stack of abstractions, where *Intent* messages form the highest level of abstraction. Intents are message objects that describe an operation to be performed and optionally contain a payload. When sending an Intent, the receiving component can be explicitly stated or be implicitly inferred from the Intent attributes. Whether an Intent message is successfully delivered to a suitable receiver can be constraint by permissions required by the sender and/or receiver.

An application can have four types of components accessed via IPC by other applications: (1) *Service:* Runs in the background and can be started with Intents or be bound to in order to make synchronized remote procedure calls. Service is used for long running processes that do not need to run in foreground. (2) *Activity:* Provides user interfaces and can be started with an Intent. Generally an Activity is associated with one user-interface layout. (3) *BroadcastReceiver*: Runs in background and handles broadcast Intents. (4) *ContentProvider*: Provides access to locally stored, structured data through an SQL-like interface. Although logically the inter-app communication happens directly between the components of different apps, all such communication has to be routed through the ActivityManagerService (AMS), which resolves the corresponding receiver component(s) and also enforces any potential permissions required by either sender or receiver of the message. As such, the AMS forms a reference monitor for inter-app communication that cannot be bypassed and that executes behind a strong security boundary.

Application Launch. In Android, all application processes are forked from a "warmed-up," empty application process called *zygote* that already pre-loaded all necessary user level libraries. Upon application launch, Android forks the zygote process and loads the application into the newly forked process. The zygote process requires root privileges for its operation and before handing control to the loaded application code, all the privileges of the newly forked process are changed according to the granted permission of the loaded app using a *setUID* syscall to change from root to the app's UID.

2.2 Non-Interference

If a user wants to keep data confidential, she can define policies, such that the data tagged confidential will be invisible to other users. This includes data affected by the confidential data as well. This policy allows a program to modify private data as long as it is not leaking any part of it [40]. The policy that governs this kind of data accesses is called a *non-interference policy* [17]. An attacker is allowed to see public data as output but not confidential data. The usual way of showing a program is non-interfering is by showing that the attacker is unable to distinguish the output between two runs of the program that differ in their confidential input [18].

3 RELATED WORK

We present and discuss related works on Android security extensions, general non-interference through IFC, IFC and process centric policies in Android, and de-centralized IFC in Android.

3.1 Android Security Extensions

Over the last decade [1], a variety of Android security extensions have been brought forward. At this point, we focus on the extensions we deem most relevant for our work. For instance, *Scippa* [7] and *Quire* [12] made the call chain of inter-app communication available to IPC message receivers, allowing them to detect untrusted input and commands from unauthorized apps that have been relayed (e.g., via deputy apps). However, *Quire* relies on app developers for forwarding this information, while *Scippa* tries to establish the chains system-centric but still has to rely on code within application sandboxes. Hence, both approaches are not ideal for controlling information flows. With *IPC inspection* [15] the permissions of IPC receiving apps are reduced to the intersection of their permissions with those of the sending application. To better support parallel calls, receiver apps are also multi-instantiated. However, this solution targets specifically the problem of confused deputies by preventing a receiver app from exercising privileges its caller does not hold. Different solutions addressed the problem of over-privileged, privacy-invading advertisement libraries [9, 14, 20, 45, 47]. Generally the solution is to establish privilege separation between the application and the ad-library by executing the library in a different process under a different UID with distinct permissions. For instance, *AdDroid* [35], *AdSplit* [42], *CompARTist* [26], and *AFrame* [55] implement this solution. In contrast, IFC with Ariel does not separate the privileges, but controls data flow to and from the library execution, hence preventing a library from misusing its inherited permissions.

3.2 Non-interference through IFC

IFC helps enforcing data flow channels between two or more security principals in the system according to policies. Enforcing IFC policies has been an active area of research since the introduction of the Bell-LaPadula or BIBA models for security enforcement and the most commonly targeted policy of IFC is to ensure *non-interference*. Two of the very well known flow enforcement techniques are 1) static information flow, using language-based technologies and type systems [4, 6, 29, 40, 51, 52, 54] ; and 2) dynamic information flow, using runtime monitoring techniques [5, 24, 34, 41]. But there is an elemental conflict between static and dynamic analysis. It is very hard to provide enough conviction to prove superiority of one over the other [39], since they suffer the usual drawbacks of dynamic and static analysis techniques. Although intuitively dynamic information flow enforcement seems more accurate due to its capability to handle information flows at runtime with real inputs, the execution paths that are not taken by the current execution are impossible to observe and it is hard to reason that the enforcement will in fact enforce the policy correctly on all paths. In contrast, static analysis inherently includes all possible execution paths, but often ends up with an over/under estimation of the result due to the absence of real inputs and the inherent drawbacks of statically analyzing code. To bridge the gap between static and dynamic analysis, SME was introduced [11, 22]. In contrast to monitoring data leakage at runtime or detecting leaks in a program statically beforehand, secure multi-execution prevents any data leaks during the execution of the program by considering the program code as blackbox that is neither instrumented for runtime analysis nor verifiable beforehand. Due to this reason, preventing leaks during program execution demands strict policies implemented in the operating system and a reliable reference monitor to enforce the policies. Our approach is integrated into the operating system [14, 32], enforcing a runtime security protocol and treating application sandboxes as blackboxes, as we explain in more detail in Section 5.

3.3 Information flow control in Android

There are different approaches for information flow control based security for Android. *R-Droid* [6] leveraged static analysis with optimized slicing. *FlowDroid* [29], *IccTA* [4] implement static taint tracking for Android applications. *TaintDroid* [13] implements a dynamic taint tracking system to detect (undesired) data flows between applications. *AppFence* [25] extends TaintDroid to not only track data but also enforce security policies that restrict the flows of the tainted data. However, both *TaintDroid* and *AppFence* rely on taint tracking and propagation logic within the untrusted application sandboxes, putting their reliability at high risk. *MOSES* [38] shows an implementation of context aware security profiling between different types of data groups, i.e., private and corporate. The segregation of data is done by virtualization and directory poly-instantiation. But only one security-profile is active at a time, requiring a high frequency of context switching between security profiles if used for IFC or very fine-grained security policies.

3.4 Secure Multi-Execution

SME is a new instrument in the information flow control toolbox. SME was first introduced by Devriese and Piessens [11]. They introduced SME with a small example implementation in the Google Chrome V8 benchmark suite. The first full-scale browser prototype of SME is *FlowFox* [21]. *FlowFox* enforces policies to mitigate three types of threats: 1) leaking session cookies, 2) history sniffing, and 3) library tracking. Although *Flowfox* has substantial memory overhead, it proved that a powerful and precise policy lattice can refine the same-origin policy yet being compatible with existing websites. *Monad ME* [27] has an SME implementation in the Haskell compiler. An extensive proof on SME was done by Rafnsson and Sabelfeld [37]. But an implementation of multi-execution based on a security policy architecture is still absent in the mobile operating system domain. We will provide more technical background information on SME in the context of our system design in the following Section 4.

3.5 Multiple faceted information flow

Austin et al. [5] presented Multiple Facets (MF) for dynamic information flows as a novel information flow security in a dynamic flow. MF uses faceted values, one for each security level. In MF, assignment to a public variable in a context depending on a secret value is skipped (based on Fenton strategies [16]). If the faceted values are implemented properly, MF can simulate SME with the primary

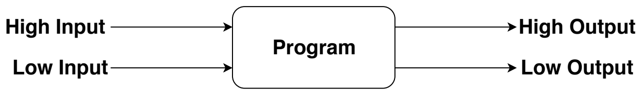

Figure 1: Normal execution with non-interference for a security lattice with two levels.

benefit of providing higher performance. Still, an attacker might deduce high information based on program termination. In comparison to SME, MF only can guarantee termination-*in*sensitive non-interference, which is a strictly weaker property than termination-sensitive non-interference [8], guaranteed by SME

3.6 De-centralized IFC (DIFC)

DIFC was first introduced by Myers and Liskov [31]. DIFC works by labeling all variables, objects, storage locations, and so on that are known as *slots* that store *values*. The model represents users and authoritative entities as *principals*. The label on a value cannot be changed but a new instance of the value can be created with a new label. There are a few implementations of DIFC available for Android [28, 33, 53]. One implementation in Android that has the most properties in common with our implementation of SME is a DIFC implementation in Android by Nadkarni et al., called *Weir* [32]. *Weir* has two interesting aspects, 1) lazy poly-instantiation and 2) forking existing process. *Weir* first taints all applications with security labels and upon an IPC call between application processes, the receiving application process is forked with a null taint and then its taint set updated with the taint of the incoming IPC call. This creates more than one instance of the receiving application, one with all the permissions and others with tainted access. This will make the forked process available to the incoming IPC call with limited privileges. However, this solution is limited by the number of possible forks. If the IPC sender has too fine-grained taints, then for each and every call a new receiver process will be forked (taint explosion), clogging the system and degrading performance. It is unclear from the current DIFC implementation how those poly-instantiated processes are cleaned-up. In contrast, SME with a fixed lattice has a finite number of required multi-instantiated apps and hence a fixed upper-bound for the number of processes.

4 SECURE MULTI-EXECUTION IN ANDROID

In this section, we will briefly explain the general design pattern of *secure multi-execution* (SME) and how it preserves non-interference. We describe later in Section 5 how the theoretical concept of SME can be coupled with Android's system design.

General concept. Secure multi-execution consists of two parts: (1) a multi-execution setup and (2) security labeling for data.

The labels define usually the *confidentiality* of data and are partially ordered, where one label is above another label if it represents a higher level of *confidentiality*. The security lattice defines the order of the security labels and data is allowed to flow upwards [10]. For instance, public data can become be private but not vice-versa. The main idea of SME is to execute a program multiple times simultaneously, where each of those program executions is related to a different security label derived from the security lattice. The input and output channels are also labeled with a security label, i.e., for each level exists a dedicated input/output channel. The level of the input channels is the same or lower than the level of the execution, but never higher. Referring to Figure 1 that is based on a security lattice with two levels (high and low), the general security or information flow control can be separated into the high input channel (private data channel) from the low input channel (public data channel) and similar for the output channel. If we turn this "classical" scenario for non-interference into a secure multi-execution of the program for non-interference between high and low, the result looks like Figure 2. SME creates multiple execution instances of the program, where the number of parallel executions is dependent on the number of security levels the security lattice is defined on. Here, for a lattice of two levels *high* (private) and *low* (public), two executions *high execution* (hi-ex) and *low execution* (lo-ex) are needed. As the security lattice defines that low data can become high data anytime, but not the other way around, hi-ex receives both high input and low input and low-ex receives only low input. Similarly the output of the executions are also controlled. Hi-ex is restrained from generating low output and lo-ex is barred from generating high output. In this scenario, non-interference can intuitively be established. If we compare Figures 1 and 2, keeping in mind that the attackers knowledge is gathered based on input and output channels, then SME does not change the classical setup: in both cases there exists only two input and only two output channels. However, in the non-SME setup, knowledge or instrumentation of the source/binary code of the program is needed to ensure non-interference of the execution, while in SME the program is purely controlled on its input and output channels through reliable scheduling of the right program execution and relaying inputs/outputs to the right channels and executions.

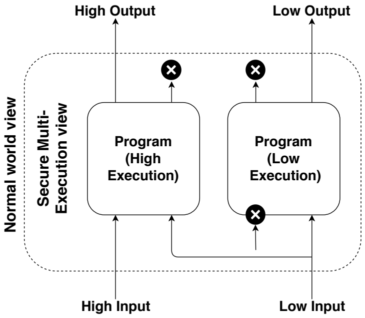

Figure 2: Secure multi-execution with non-interference for a security lattice with two levels.

Dummy inputs. By default, the SME setup requires presence of a *dummy* input for low execution in case only high input is available in order to ensure complete functionality of the program. By complete functionality, we mean, if in any case the unauthorized execution requires sensitive input, otherwise the execution will crash. To prevent the leak of high data as well as the crash of the unauthorized execution, *dummy* inputs are mandated for SME. But we can bend this rule in the specific case of Android, thanks to Android's non-dependency on different communication channels. For example, it is not mandatory to provide an Activity to the

Service component of an application to function properly, allowing us to skip such dummy inputs.

4.1 Precision and Soundness of SME

An important question for this kind of solution is the preservation of *soundness* and *precision* of the multi-executing application.

Soundness. Any program preserves non-interference under secure multi-execution. In secure multi-execution, any instance of execution is labeled with a security level. It only can produce output at that level. As a result, SME is sound, given correct scheduling of the executions and correct forwarding of inputs and outputs between the executions at different security levels.

Precision. If a program is non-interfering in normal execution, then its behavior under a normal execution and under SME are the same [11]. SME shows *per-channel transparency for a secure program* (used similar to precision) [37]. This means if the original program is secure then, from the view point of each channel, the sequence of input-output events in a given run of a program is the same in the original execution (Figure 1) and in SME (Figure 2).

Design considerations. Enforcing soundness and precision strictly also benefits the design of our secure multi-execution solution on Android. First, we do not want to modify application code. There exists no standard way to provide security guarantees for security modifications through injection. We want to ensure that if an application is non-interfering in normal execution then it is safe under SME execution also, thus our solution benefits from abstaining from code injections as done in prior works. Second, we do not want to provide an application-specific customized security lattice and I/O channeling policy. Providing customized policies is impossible without involving developers for millions of applications that can potentially run on a mobile phone. Thus, we want to provide a generalized lattice structure derived from Android's permission system. Re-using the permission system will help us not only to keep our lattice small and limited in structure, but also to keep it compatible with all applications. Third, every application is an ensemble of custom written program code. It is impossible to create a general framework that will work for every application by simply injecting some standard security code. I/O channels are provided and controlled by the operating system (including the middleware on Android) and it is easy to control the I/O channel with policies enforced by the operating system.

4.2 Classification and model selection

We have two different types of SME implementation available. One enforces totality of input environment where there is always an input present when the system needs it and the inputs are parsed by a cooperative scheduling [11]. But this behavior does not help us understand progress blocking interactive systems [36] (i.e., no progress in control flow until a new input enters the system). Android works in a non-total environment mode. The time that the user or an application provides input is by default totally random. To guarantee protection against attacks powered by varied input presence, Rafnsson and Sabelfeld [37] introduced a non-total environment of input. We followed this design as an obvious choice.

5 SYSTEM DESIGN OF ARIEL

Realizing secure multi-execution in Android requires 1) multi-execution of application sandboxes and 2) a new component to label the I/O channels to the multi-executing application processes.

5.1 Multi-Execution

Multi-executing Android apps requires modifications of the application launching process in Android. At this point, we explain the launching process in more technical details and elaborate on how we extend it to create a multi-executing environment for an application. Figure 3 depicts the application launch process upon receiving a request from the user, e.g., the user clicked the app's icon in the application menu. By default, starting an application is equal to starting an Activity of the application. The user's request first is caught by the Activity handler through *startActivityForResult()* (🅐 in Figure 3) and passed on to the Android *instrumentation* (🅑). The instrumentation class starts the Activity through *execStartExecute()* and passes it to *ActivityManagerService* through *ActivityManagerServiceProxy* (🅒 , 🅓 , and 🅔). Once the launch request enters the *ActivityManagerService* (AMS), *ActivityStackSupervisor* serves the request for launching a new Activity and locks the target application. Locking helps synchronize the application state, so no other process can interfere. Then AMS checks if there already exists any running process of the target application. If found, the same PID of the process will be returned to answer the incoming request (🅗). Otherwise, the request is passed to the *zygote* daemon through *Process* (🅕 and 🅖). Zygote forks itself and returns an empty application process into which the application code is loaded (see also Section 2). Once the new application process is available, AMS adds the PID to an application map along with the application's name that can be used to serve subsequent requests for the same app faster (🅗). As a consequence, if on vanilla Android, a multi-execution of an app is requested, simply the already existing application process is being used, but no additional application process is being started. After inspecting the application launch process, we identified the *startSpecificActivityLocked()* function of the AMS as the most suitable candidate to extend the default application launch process with support for multi-execution of applications. This function calls the *zygote* daemon to fork and span new processes. We augment this default control flow by adding an additional branch for multi-executing processes. The overall design is depicted in Figure 4. Steps ❶ through ❸ correspond to the steps 🅐 through 🅔 of the default control flow described above for Figure 3. In our extended control flow, before checking whether an application process already exists for the target application (❺), we check whether multi-execution is needed or not for that target application. This is done by adding a new option for multi-execution in the application settings. If multi-execution is not opted in, the normal flow of application launch will continue and follow the steps described earlier (❼ through ⓫ in Figure 3). If multi-execution is opted in (❻), then AMS will send the fork request to *zygote* (⓬ and ⓭). After a successful forking, this results in two (or more) different application processes loaded with the same application code (⓮ and ⓯).

While zygote does not stop us from starting multiple processes for the same application, the process handling for applications

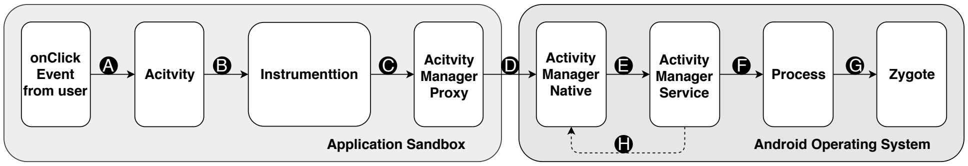

Figure 3: Default control flow to request launching a new Activity of an application, potentially involving starting a new application process.

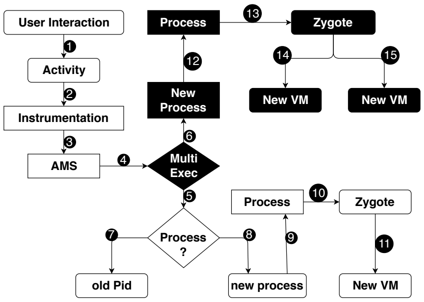

Figure 4: Control flow of ActivityManagerService to launch new application processes, extended with support for multi-execution of applications. Vanilla Android is marked in white boxes and Ariel extensions in black boxes.

within the AMS has to be further extended to support multi-execution of the same app. First, if one process for an application is available and a newly created one suddenly appears, Android will only consider the one with highest PID for dispatching events to this application or forwarding Intent messages. All other application processes of the same app with lower PIDs will be cleaned by the garbage-collector. To stop this functionality, we augmented the *mPidSelfLocked* list of AMS, that stores the relevant PID for every application, to accommodate more than one process PID of the same application, identified by *processName*, PID, and in Ariel also the security level of the process. This allows us to execute one application process for every defined security level of the security lattice. In the next section, we will talk about the policy lattice we are using. Second, among the two processes, one should be in foreground and the other one(s) should be in background (i.e., not visible to the user). The UI of an application in Android is always handled by the process with its highest PID. This behavior is deeply embedded in default Android (e.g., within Android's SurfaceFlinger responsible for processing events from the touchscreen) and can only be changed through invasive refactoring of Android's low-level code base. We utilize this constraint on the foreground app in our definition of the secure component (high) of an application (Section 5.4) and the lattice (Section 5.2).

5.2 Security Lattice

After having a multi-executing environment for Android apps, we need a policy lattice implementation for enforcing IFC. Different policies are imaginable that can dictate the form of the lattice, e.g., the common Bell-LaPadula that is also supported by SELinux' Multi-Level Security (MLS) extension in Android. However, plain Android, even with SELinux MLS, does not define a lattice, and for our Ariel implementation we derive our lattice from Android's permission system by mapping the permission protection levels to security levels (Figure 5). We focus on the normal and dangerous protection levels, which are used for the pre-defined permissions available to app developers and exclude signature or signatureOrSystem permissions that are used internally to apps and the system but not to protect user data. Thus, we enforce a non-interference between high executions with access to user-private data and low executions of untrusted code without such access. Our lattice has, hence, two security levels. We will call the execution with all privileges as *Hi-ex* and the execution with no privileges as *Lo-ex*. We need only two executions due to our two stage lattice settings. The high level of the lattice will cover both *dangerous* and *normal* permissions for high. The low level will have no permission or *Null*. As *dangerous permissions* ⊂ *normal permissions* ⊂ *null permission*, by severity and access hierarchy, our simple lattice design is partial in nature.

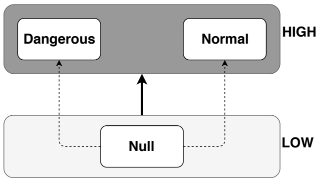

Figure 5: Security lattice derived from protection levels of Android permissions

We can easily achieve a process with null permission in Android by marking it as an *isolated process* [2] that causes the process to execute with no permissions at all. The process management in the AMS uses an internal data structure called *processRecord* to store such information about application processes. A *processRecord* involves all the data needed to manage running applications and to answer application requests for inter-app communication, including all Activities running, all Services running, or all ContentProvider connections this process is currently holding. We use this data structure for the Lo-ex process by turning the *isolated* flag to *TRUE*, effectively denying this process any permissions in the system, since any permission check for an isolated process will automatically be denied by the reference monitors in the middleware. It should be noted clearly, that this extension of Ariel affects only the enforcement in the middleware, e.g., within the AMS, location service, etc., but not the enforcement of filesystem-related permissions (e.g., Internet sockets, as explained in Section 2). Those permissions are enforced by the Linux kernel and are not affected by the *isolated* flag in the *processRecord* alone, but instead Android would by default execute an isolated component under a transient UID and GID that has no filesystem access at all. Covering those filesystem permissions in Ariel would require an integration of the lattice into the enforcement by the kernel, e.g., integration with SELinux MLS as proposed in Section 7.

5.3 Input/Output channel classification

In addition to multi-executing and labeling application processes, SME requires I/O channeling and labeling of data. In Android, inter-component I/O channels are based on Intent based communication. For instance, starting an Activity, contacting a BroadcastReceiver, or starting a Service require Intent-based communication between sender and receiver application. All Intents are being routed between sender and receiver through the AMS, i.e., the AMS is a mandatory "man-in-the-middle" for Intents and responsible for resolving the receiver of Intents using its internal app management data and for delivering the Intent to the corresponding, authorized receivers.

Among the four app components, we consider Intents to an *Activity* as high input to the application of that Activity. This is motivated by the fact that Activities form a user interface that includes user interaction on screen and that the high execution of any multi-executed application is privileged to handle user-data, i.e., should also involve the user. In alternative policy settings, this design decision might change (e.g., if the security label of an execution depends on the code contained within the execution, the I/O channel label might depend on that). All other three channels, *Service, ContentProvider* and *BroadcastReceiver*, are considered low input to the multi-executing application. This is due to their hidden nature for the user. A *Service* request, a *ContentProvider* request, or a *Broadcast* does not need user interaction to communicate with another component from another application. We embed this I/O channeling inside the scheduler model of Ariel, as described in the following section.

5.4 Securing Multi-Execution with Lattice-based Security Policy

Through our extensions to the AMS, we achieved basic support for multi-execution of applications, adding all instances of an app to the AMS' process records to use them separately. We also marked the Lo-ex process as isolated and the Hi-ex process as Activities holding all granted permissions of the app, and we created a security lattice based on the permission protection levels. To actually *secure* the multi-execution of apps and enforce non-interference, the SME design demands a *scheduling* over the I/O channels between Hi-ex and Lo-ex. To this end, we implemented an I/O scheduling module embedded inside the AMS. The primary functionality of this module is to intercept all the Intent-based communication to and from an application that is routed through the AMS. Once there is an incoming message request, our scheduler first detects the destination of the request using the default receiver resolution of AMS. Then it queries the process records by *processName* (by default the application name) to get the associated PIDs. If only one PID is returned, the destination process is not multi-executing and the communication will take place as in default Android. But if two PIDs are returned, it will consider the PID with higher ID as the Hi-ex and forward all the incoming communication (both high and low). In Ariel, only the Activity response from the Hi-ex will be returned to the sender application. The process with lower PID will be considered as Lo-ex and all the low communications will be forwarded to the Lo-ex calling process, but none of the high communication. In summary, this scheduling implements the SME with non-interference as depicted in Figure 2.

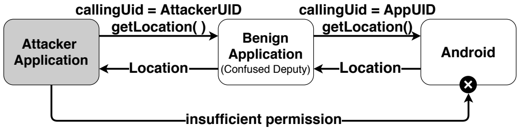

Figure 6: Permission re-delegation attack

6 CASE STUDIES

We have implemented Ariel in Android 6.0 (Marshmallow). To illustrate the benefits of Ariel for information flow control, we present two case studies for the protection of the user's privacy: mitigating permission re-delegation attacks [15] and mitigating data leaks from untrusted third party code [9, 14, 20, 45, 47].

6.1 Permission Re-delegation Attack

In our experimental setup we use a custom, privileged, benign application in combination with a custom attacker app as a running example for establishing SME. The benign app emulates a messaging app with an Activity to read messages and with a service API that leaks location data. Our attacker app retrieves the location data through service requests to benign app without holding the necessary permission to access location data itself (see Figure 6). This is an illustrative case for a permission re-delegation attack [15], a specialized case of the more general confused deputy problem, where a privileged application is tricked into misusing its privileges on behalf of another, unprivileged app. Here, the messaging app leaks its privileges for accessing the location data by leaking the corresponding data to the attacker app through an unprotected Service interface. Equivalent cases have been identified in the literature [1]. In past works, this problem has been tackled, for instance, by creating call-chains [7, 12] that allow the callee (here Android) to detect deputies, by reducing the privileges of callees [15], or by tainting data and enforcing policies on tainted data flows [13, 49]. As discussed in Section 3, those solutions come with their respective sets of drawbacks in practice (e.g., relying on code within the application sandboxes or involving the app developer).

Mitigation using SME. We show how to mitigate this problem by secure multi-executing the benign app. For the security lattice we use the two level security lattice introduced earlier (see Figure 5), where the Hi-ex holds both dangerous and normal permissions and Lo-ex has no permission at all. Location data is by default protected by a dangerous permission and hence labeled as high data. Our lattice hence implies that the input flowing to low execution cannot end up in any data protected by any kind of permissions.

When we run the benign application in SME mode the setup looks like Figure 7. As described in our system design in Section 5, the Ariel extension to the AMS realizes a scheduler that forwards

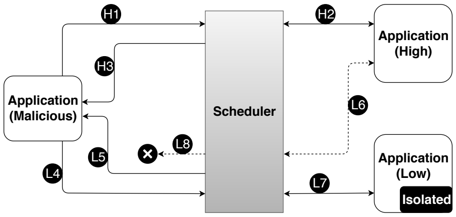

Figure 7: Mitigating confused deputy with SME

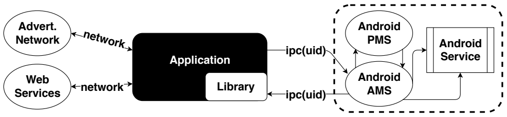

Figure 8: Advertisement library attack

inter-app communication between the attacker and benign app and is responsible for filtering input/output to and from the benign app according to the security lattice and labels of the execution instances. For our experiment we have considered only *Activity* and *Service* communication between apps. *ContentProvider* and *BroadcastReceiver* can be easily accommodated in the scheduler by intercepting the communication channels within the AMS.

In Figure 7, the attacker application sends an Activity request for reading messages (H1) and a service request to get location data (L4). Both requests are intercepted by the scheduler in the AMS, since the AMS has to be a mandatory relay in this communication, hence allowing our scheduler to operate as a non-bypassable reference monitor that is simultaneously behind a strong security boundary (separated process from the attacker app). The scheduler finds both the high and low instances of the multi-executing messaging app. The scheduler sends the activity request to the high execution of the messenger app (H2) and the service request to the low execution (L7) as well as the high execution (L6). The high execution answers back to the activity request with a result to the scheduler (via H2). The scheduler forwards the activity result to the attacker application as high output of a high input (H3), but it will drop the service response from the high execution considering it as low output of a high execution (high↛low in L8). Further, the low execution of the messaging app cannot answer the service request of get location due to its isolation and answers back with an error that is does not have enough permissions (via L7) and the scheduler forwards the error message to the attacker app (L5). The attacker application receives the messages from the messaging Activity as intended but it is unable to leak the location data though the messaging application Service. This way we can protect a benign application from acting as a confused deputy.

6.2 Malicious Libraries

A second attacker scenario we take as a case study are malicious advertisement libraries. Advertisement libraries are frequently included by app developers to monetize their applications, however, those libraries have repeatedly been shown [1, 9, 35, 42, 47] to be a hazard to the user's privacy. In Android's sandboxing model, those libraries execute under the UID of their host application (i.e., there is a lack of same origin policies as known from the web domain [43]) and hence they automatically inherit all permissions assigned to this UID. By actively using those privileges, libraries have been shown to leak user data to their servers without the user's explicit consent. This attack is depicted in Figure 8.

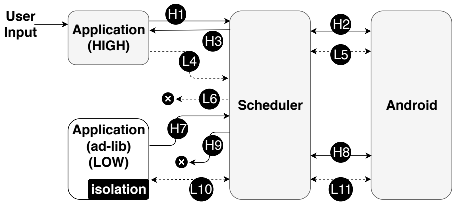

Figure 9: Mitigating malicious libraries using SME in combination with Android's *process* component attribute.

Mitigation using SME. Mitigation of malicious ad libraries with SME is not as simple as mitigating the permission re-delegation we presented in the last section. Unlike permission re-delegation mitigation, where the multi-executing application receives external input that can be unambiguously routed to and from the application instances, in the case of libraries a clear separation between calls from application code and library code to the Android services (e.g., AMS, location, etc.) is necessary. Several research works have shown good promise on separating library calls from application code. The research ranges from separating the library code into dedicated advertisement system services or apps [35, 42] to application byte-code rewriting [26, 30, 48]. A similar, more straight forward approach to achieve the same goal with SME can be based on Android's *process* attribute [3] for app components. A component tagged with this attribute will execute in a separate process with a different PID then the rest of the app components (i.e., separate entries in the *processRecord*s of AMS), but with the same UID. Hence, on default Android, the separate process would still have the same permissions as the app, however, when used in combination with SME we can control the data flows to and from those separate processes. With this setup in place, we can move library code into Activity components executing in a separate process and apply SME to prevent unwanted data flows to and from the library code.

Figure 9 presents this setup and the mitigation against data leaks by malicious libraries. The main application code, with which the user interacts, is labeled as high execution. When this code requests access to another Activity (H1), the scheduler will forward this request to AMS (H2) and return data on the high channel (H2 and H3). If the main application requests access to another Service component (L4), this is a low request and the scheduler will block the request due to violation of non-interference property (L5 and L6). In contrast, the ad library is executing in a low, isolated process, where all the service requests to and from the process and Android will be denied due to isolation (L10 and L11). Additionally, the low process could send an Intent to another Activity (low→high flow in H7 and H8), but any potential return value from that Activity would be dropped by the scheduler (violating high↛low flow in H9). Lastly, if the advertisement process requests access to another service component (L10 and L11), which could result in a data leak to the advertisement library (e.g., location, user contacts), the scheduler will forward this request, but the Android system services refuse cooperation with the isolated, low advertisement process.

7 DISCUSSION & FUTURE WORK

Ariel is the first realization of SME in Android and its elegantly simple implementation can establish well-known non-interference policies between application processes, but it naturally suffers some limitations like any other security extension. In this section, we discuss those limitations and also point out alternative policies and future work that we consider interesting.

Security levels vs. performance overhead. The current implementation of Ariel supports only two levels, namely, *high* and *low*. Accommodating more levels is fundamentally not difficult. Using a more fine-grained lattice than our lattice will help include more security levels, between which non-interference can be enforced by the scheduler. But this will increase the number of multi-executing instances, ultimately leading to higher performance overhead, since for every new security level another instance of the same app has to be spawned. Thus, as with SME outside the Android domain, this requires a careful trade-off between the granularity of the security policy and the acceptable overhead. It should be, however, noted that other solutions like poly-instantiations for DIFC (e.g., [32]) or IPC inspection [15] scale badly.

Support for declassification. Ariel does not support any kind of declassification policies. This makes the system more restricted, limiting the number of policies it can support. Rafnsson and Sabelfeld [37] showed how declassification policies can be integrated in presence of a fine-grained security lattice and extending Ariel with such support is left for future work.

More channels and different policies. Our current implementation does not consider the *filesystem* as I/O channel between application processes, since in that case we need to embed the SME security lattice policy in the kernel, which is managing this communication channel. This can be a potential future work by extending the underlying Linux kernel to become aware of multi-execution of apps. A potential alley to this end could be an integration with SElinux MLS to enforce a fine-grained security lattice that governs which app instances can communicate with which other app instances via the filesystem (i.e., the security levels of the lattice are reflected in the SELinux MLS security levels) or even Binder IPC.

Further, our current lattice is derived from the protection levels of permissions and hence this governs what we consider as high and low security levels. For instance, Activities and communication between Activities are considered high and hence any data can flow between Activities. Different policies and strategies to derive the lattice could be explored. As a concrete example, in Aquifer [33] data flows are controlled along UI-flows, e.g., an email app Activity sends an attachment to a reader app Activity for displaying, and Aquifer reduces the privileges of the reader app to prevent data leakage. In Ariel, using a different policy for the security lattice could establish the same protection by ensuring that the attachment is scheduled to the low execution of the reader app (e.g., based on the type of the forwarded data payload).

8 CONCLUSION

The plethora and extent of private information that is concentrated on today's mobile phones necessitates rigid and reliable privacy controls. Over the last decade, a broad spectrum of research works has proposed valuable extensions to Android's security architecture to enhance the control over user-data, such as location, contacts, media files, etc. While those works made valuable contributions to improve Android's security, the majority of the solutions was concerned with controlling the *access* to data. However, equally important is the control of how data, once released to an app, can flow between different apps or even the components of the same app with different trust levels. Neglecting control of the information flow opens the door for attacks, such as permission re-delegation or malicious advertisement libraries.

Information flow control is a long-standing problem in the security community and very often the proposed solutions face intricate problems, such as requiring developer support (which hinders backwards compatibility or developer-independent deployment) or inlined security-critical code (which is notoriously hard to protect against a maliciously acting app developer), to name two practical concerns. One of the newer tools in the toolbox for information flow control is secure multi-execution, which treats processes as blackboxes and enforces non-interference policies through secure scheduling between different instances of the same program at different security levels of a lattice.

In this paper, we presented Ariel, the first implementation of secure multi-execution for Android. We solved the technical challenge of multi-executing Android applications and integrating a scheduler into Android's ActivityManagerService to enforce non-interference of the data flows between the instances of applications at different security levels. We showed how SME with Ariel can help mitigate two well-known attacks on Android, permission re-delegation and malicious advertisement libraries, and pointed out future directions of SME on Android to support other use-cases and further inter-app communication channels.

Acknowledgments. 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) (AutSec/FKZ: 16KIS0753).

REFERENCES

[1] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. SoK: Lessons Learned From Android Security Research For Appified Software Platforms. In *Proc. 37th IEEE Symposium on Security and Privacy (SP '16)*. IEEE Computer Society.

[2] Android Developer Docs. 2017. Android Manifest File: Service. https://developer.android.com/guide/topics/manifest/service-element. Last visisted: 06/12/2018.

[3] Android Developer Docs. 2018. Android Manifest File: Activity. https://developer.android.com/guide/topics/manifest/activity-element. Last visisted: 06/13/2018.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM.

[5] Thomas H. Austin, Thomas Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst.* 39, 3 (2017), 10:1–10:56.

[6] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. 2016. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM.

[7] Michael Backes, Sven Bugiel, and Sebastian Gerling. 2014. Scippa: system-centric IPC provenance on Android. In *Proc. 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM.

[8] Nataliia Bielova and Tamara Rezk. 2016. Spot the Difference: Secure Multi-execution and Multiple Facets. In *Proc. 21st European Symposium on Research in Computer Security (ESORICS 2016)*. Springer.

[9] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. *CoRR* abs/1303.0857 (2013). http://dblp.uni-trier.de/db/journals/corr/corr1303.html#abs-1303-0857

[10] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243.

[11] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *Proc. 31st IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society.

[12] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proc. 20th USENIX Security Symposium (SEC '11)*. USENIX Association.

[13] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association.

[14] William Enck, Damien Octeau, Patrick McDaniel, and Chaudhuri Swarat. 2011. A Study of Android Application Security. In *Proc. 20th USENIX Security Symposium (SEC '11)*. USENIX Association.

[15] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *Proc. 20th USENIX Security Symposium (SEC '11)*. USENIX Association.

[16] J. S. Fenton. 1974. Memoryless Subsystems. *Comput. J.* 17, 2 (1974), 143–147.

[17] Joseph A. Goguen and JosÃľ Meseguer. 1982. Security Policies and Security Models. In *Proc. 3rd IEEE Symposium on Security and Privacy (SP '82)*. IEEE Computer Society.

[18] Joseph A. Goguen and JosÃľ Meseguer. 1984. Unwinding and Inference Control. In *Proc. 5th IEEE Symposium on Security and Privacy (SP '84)*. IEEE Computer Society.

[19] Google. 2018. Put Android to Work. https://www.android.com/enterprise/employees/. Last visisted: 06/13/2018.

[20] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '12)*. ACM.

[21] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)*. ACM.

[22] Gurvan Le Guernic. 2007. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* Ph.D. Dissertation. Kansas State University, United States of America.

[23] Hao Hao, Vicky Singh, and Wenliang Du. 2013. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS '13)*. ACM.

[24] Daniel Hedin and Andrei Sabelfeld. 2012. Information-Flow Security for a Core of JavaScript. In *Proc. 25th Computer Security Foundations Symposium (CSF '12)*. IEEE Computer Society.

[25] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart E. Schechter, and David Wetherall. 2011. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)*. ACM.

[26] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 2017. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In *Proc. 24th ACM Conference on Computer and Communication Security (CCS'17)*. ACM.

[27] Mauro Jaskelioff and Alejandro Russo. 2011. Secure Multi-execution in Haskell. In *Ershov Memorial Conference (Lecture Notes in Computer Science)*, Vol. 7162. Springer.

[28] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android - (Extended Abstract). In *Proc. 18th European Symposium on Research in Computer Security (ESORICS 2013)*. Springer.

[29] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press.

[30] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proc. 13th International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*. ACM.

[31] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM.

[32] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *Proc. 25th USENIX Security Symposium (SEC' 16)*. USENIX Association.

[33] Adwait Nadkarni and William Enck. 2013. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)*. ACM.

[34] Minh Ngo, Fabio Massacci, Dimiter Milushev, and Frank Piessens. 2015. Runtime Enforcement of Security Policies on Black Box Reactive Programs. In *Proc. 42nd Symposium on Principles of Programming Languages (POPL '15)*. ACM.

[35] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: privilege separation for applications and advertisers in Android. In *Proc. 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS '12)*. ACM.

[36] Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. 2012. Securing Interactive Programs. In *Proc. 25th Computer Security Foundations Symposium (CSF '12)*. IEEE Computer Society.

[37] Willard Rafnsson and Andrei Sabelfeld. 2016. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security* 24, 1 (2016), 39–90.

[38] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. 2012. MOSES: supporting operation modes on smartphones. In *Proc. 17th Symposium on Access Control Models and Technologies (SACMAT '12)*. ACM.

[39] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. 23rd Computer Security Foundations Symposium (CSF '10)*. IEEE Computer Society.

[40] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[41] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Transactions on Information and System Security* 3, 1 (Feburary 2000), 30–50.

[42] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *Proc. 21st USENIX Security Symposium (SEC '12)*. USENIX Association.

[43] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. 2010. On the Incoherencies in Web Browser Access Control Policies. In *Proc. 31st IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society.

[44] S. Smalley and R. Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS '13)*. The Internet Society.

[45] Sooel Son, Google Daehyeok, Kim Kaist, and Vitaly Shmatikov. 2015. What Mobile Ads Know about Mobile Users. In *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS '16)*. The Internet Society.

[46] Statista. 2018. Mobile OS market share. https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/

[47] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*. IEEE Computer Society.

[48] Mengtao Sun and Gang Tan. 2014. NativeGuard: protecting android applications from third-party native libraries. In *Proc. 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '14)*. ACM.

[49] Mingshen Sun, Tao Wei, and John C. S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proc. 23rd ACM Conference on Computer and Communication Security (CCS'16)*. ACM.

[50] Eran Tromer and Roei Schuster. 2016. DroidDisintegrator: Intra-Application Information Flow Control in Android Apps. In *Proc. 13th ACM Symposium on Information, Computer and Communication Security (ASIACCS '16)*. ACM.

[51] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188.

[52] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM.

[53] Yuanzhong Xu and Emmett Witchel. 2015. Maxoid: transparently confining mobile applications with custom views of state. In *Proc. 10th European Conference on Computer Systems (EuroSys '15')*. ACM.

[54] Steve Zdancewic and Andrew C. Myers. 2003. Observational Determinism for Concurrent Program Security. In *Proc. 16th IEEE Computer Security Foundations Workshop (CSFW '13)*. IEEE Computer Society.

[55] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: Isolating Advertisements from Mobile Applications in Android. In *Proc. 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM.