# Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies

Sven Bugiel*
*bugiel@cs.uni-saarland.de*
*Saarland University, Germany*

Stephan Heuser
*stephan.heuser@sit.fraunhofer.de*
*Fraunhofer SIT, Germany*

Ahmad-Reza Sadeghi
*ahmad.sadeghi@trust.cased.de*
*Technische Universität Darmstadt / CASED, Germany*

## Abstract

In this paper we tackle the challenge of providing a generic security architecture for the Android OS that can serve as a flexible and effective ecosystem to instantiate different security solutions. In contrast to prior work our security architecture, termed *FlaskDroid*, provides mandatory access control simultaneously on both Android's middleware and kernel layers. The alignment of policy enforcement on these two layers is non-trivial due to their completely different semantics. We present an efficient policy language (inspired by SELinux) tailored to the specifics of Android's middleware semantics. We show the flexibility of our architecture by policy-driven instantiations of selected security models such as the existing work *Saint* as well as a new privacy-protecting, user-defined and fine-grained per-app access control model. Other possible instantiations include *phone booth mode*, or *dual persona* phone. Finally we evaluate our implementation on SE Android 4.0.4 illustrating its efficiency and effectiveness.

## 1 Introduction

Mobile devices such as smartphones and tablets have become very convenient companions in our daily lives and, not surprisingly, also appealing to be used for working purposes. On the down side, the increased complexity of these devices as well as the increasing amount of sensitive information (private or corporate) stored and processed on them, from user's location data to credentials for online banking and enterprise VPN, raise many security and privacy concerns. Today the most popular and widespread smartphone operating system is Google's Android [4].

**Android's vulnerabilities.** Android has been shown to be vulnerable to a number of different attacks such as malicious apps and libraries that misuse their privileges [57, 40, 25] or even utilize root-exploits [55, 40] to extract security and privacy sensitive information; taking advantage of unprotected interfaces [14, 12, 53, 32] and files [49]; confused deputy attacks [16]; and collusion attacks [46, 34].

**Solutions.** On the other hand, Android's open-source nature has made it very appealing to academic and industrial security research. Various extensions to Android's access control framework have been proposed to address particular problem sets such as protection of the users' privacy [19, 28, 15, 52, 7, 30]; application centric security such as *Saint* enabling developers to protect their application interfaces [39]; establishing isolated domains (usage of the phone in private and corporate context) [9]; mitigation of collusion attacks [8], and extending Android's Linux kernel with Mandatory Access Control [48].

**Observations.** Analyzing the large body of literature on Android security and privacy one can make the following observations: First, almost all proposals for security extensions to Android constitute mandatory access control (MAC) mechanisms that are tailored to the specific semantics of the addressed problem, for instance, establishing a fine-grained access control to user's private data or protecting the platform integrity. Moreover, these solutions fall short with regards to an important aspect, namely, that protection mechanisms operate only at a specific system abstraction layer, i.e., either at the middleware (and/or application) layer, or at the kernel-layer. Thus, they omit the peculiarity of the Android OS design that each of its two software layers (middleware and kernel) is important within its respective semantics for the desired overall security and privacy.

[*] Author was affiliated with Technische Universität Darmstadt/CASED at the time this work was conducted.

Only few solutions consider both layers [8, 9], but they support only a very static policy and lack the required flexibility to instantiate different security and privacy models.

The second observation concerns the distinguishing characteristic of application development for mobile platforms such as Android: The underlying operating systems provide app developers with clearly defined programming interfaces (APIs) to system resources and functionality – from network access over personal data like SMS/contacts to the onboard sensors. This clear API-oriented system design and convergence of functionality into designated service providers [54, 36] is well-suited for realizing a security architecture that enables fine-grained access control to the resources exposed by the API. As such, mobile systems in general and Android in particular provide better opportunities to more efficiently establish a higher security standard than possible on current commodity PC platforms [31].

**Challenges and Our Goal.** Based on the observations mentioned above, we aim to address the following challenges in this paper: 1) Can we design a generic and practical mandatory access control architecture for Android-based mobile devices, that operates on both kernel and middleware layer, and is flexible enough to instantiate various security and privacy protecting models just by configuring security policies? More concretely, we want to create a generic security architecture which supports the instantiation of already existing proposals such as *Saint* [39] or privacy-enhanced system components [58], or even new use-cases such as a *phone booth mode*. 2) To what extent would the API-oriented design of Android allow us to minimize the complexity of the desired policy? Note that policy complexity is an often criticized drawback of generic MAC solutions like SELinux [33] on desktop systems [54].

**Our Contribution.** In this paper, we present the design and implementation of a security architecture for the Android OS that addresses the above mentioned challenges. Our design is inspired by the concepts of the *Flask* architecture [50]: a modular design that decouples policy enforcement from the security policy itself, and thus provides a generic architecture where multiple and dynamic security policies can be supported by the system. In particular, our contributions are:

1. *System-wide security framework.* We present an Android security framework that operates on both the middleware and kernel layer. It addresses many problems of the stock Android permission framework and of related solutions which target either the middleware or the kernel layer. We base our implementation on SE Android [48], which has already been partially merged into the official Android source-code by Google[1].

2. *Security policy and type enforcement at middleware layer.* We extended Android's middleware layer with type enforcement and present our policy language, which is specifically designed for the rich semantics at this layer. The alignment of middleware and kernel layer policies in a system-wide security framework is non-trivial, particularly due to the different semantics of both layers.

3. *Use-cases.* We show how our security framework can instantiate selected use-cases. The first one is an attack-specific related work, the well-known application centric security solution Saint [39]. The second one is a privacy protecting solution that uses fine-grained and user-defined access control to personal data. We also mention other useful security models that can be instantiated with FlaskDroid.

4. *Efficiency and effectiveness.* We successfully evaluate the efficiency and effectiveness of our solution by testing it against a testbed of known attacks and by deriving a basic system policy which allows for the instantiation of further use-cases.

## 2 Background

In this section, we first present a short overview of the standard Android software stack, focusing on the relevant security and access control mechanisms in place. Afterwards, we elaborate on the SE Android Mandatory Access Control (MAC) implementation.

### 2.1 Android Software Stack

Android is an open-source software stack tailored to mobile devices, such as smartphones and tablets. It is based on a modified Linux kernel responsible for basic operating system services (e.g. memory management, file system support and network access).

Furthermore, Android consists of an application framework implementing (most of) the Android API. System Services and libraries, such as the radio interface layer, are implemented in C/C++. Higher-level services, such as System settings, the Location- and Audiomanager, are implemented in Java. Together, these components comprise the middleware layer.

[1] `http://www.osnews.com/story/26477/Android_4_2_alpha_contains_SELinux`

Android applications (apps) are implemented in Java and may contain native code. They are positioned at the top of the software stack (application layer) and use kernel and middleware Services. Android ships with standard apps completing the implementation of the Android API, such as a Contacts (database) Provider. The user can install additional apps from, for example, the Google Play store.

Android apps consist of certain components: Activities (user interfaces), Services (non user-interactive tasks), ContentProviders (SQL-like databases), and Broadcast Receivers (mailboxes for broadcast messages). Apps can communicate with each other on multiple layers: 1) Standard Linux Inter-Process Communication (IPC) using, e.g., domain sockets; 2) Internet sockets; 3) *Inter-Component Communication* (ICC) [21], a term abstractly describing a lightweight IPC mechanism between app components, called *Binder*. Furthermore, predefined actions (e.g., starting an Activity) can be triggered using an Intent, a unicast or broadcast message sent by an application and delivered using the Android ICC mechanism.

## 2.2 Security Mechanisms

**Sandboxing.** Android uses the Linux discretionary access control (DAC) mechanism for application sandboxing by assigning each app a unique user identifier (UID) during installation[2]. Every process belonging to the app is executed in the context of this UID, which determines access to low level resources (e.g. app-private files). Low-level IPC (e.g. using domain sockets) is also controlled using Linux DAC.

**Permissions.** Access control is applied to ICC using *Permissions* [21]: Labels assigned to apps at install-time after being presented to and accepted by the user. These labels are checked by reference monitors at middleware- and application level when security-critical APIs are accessed. In addition to Android's default permissions, app developers can define their own permissions to protect their applications' interfaces. However, it should be noted that the permission model is *not* mandatory access control (MAC), since callees must discretely deploy or define the required permission check and, moreover, permissions can be freely delegated (e.g., URI permissions).

Permissions are also used to restrict access to some low level resources, such as the world read-/writeable external storage area (e.g. a MicroSD card) or network access. These permissions are mapped to Linux group identifiers (GIDs) assigned to an app's UID during installation and checked by reference monitors in the Linux kernel at runtime.

## 2.3 SELinux

Security Enhanced Linux (SELinux) [33] is an instantiation of the Flask security architecture [50] and implements a policy-driven mandatory access control (MAC) framework for the Linux kernel. In SELinux, policy decision making is decoupled from the policy enforcement logic. Various access control enforcement points for low-level resources, such as files, IPC, or memory protection enforce policy decisions requested from a *security server* in the kernel. This security server manages the policy rules and contains the access decision logic. To maintain the security server (e.g., reload the policy), SELinux provides a number of userspace tools.

**Access Control Model.** SELinux supports different access control models such as *Role-Based Access Control* and *Multilevel Security*. However, *Type Enforcement* is the primary mechanism: each object (e.g., files, IPC) and subject (i.e., processes) is labeled with a *security context* containing a *type* attribute that determines the access rights of the object/subject. By default, all access is denied and must be explicitly granted through policy rules—*allow rules* in SELinux terminology. Using the notation introduced in [26], each rule is of the form

$$allow\ T_{Sub}\ T_{Obj}\ :\ C_{Obj}\ O_C$$

where $T_{Sub}$ is a set of subject types, $T_{Obj}$ is a set of object types, $C_{Obj}$ is a set of object classes, and $O_C$ is a set of operations. The object classes determine which kind of objects this rule relates to and the operations contain specific functions supported by the object classes. If a subject whose type is in $T_{Sub}$ wants to perform an operation that is in $O_C$ on an object whose class is in $C_{Obj}$ and whose type is in $T_{Obj}$, this action is allowed. Otherwise, if no such rule exists, access is denied.

**Dynamic policies.** SELinux supports to some extent dynamic policies based on *boolean flags* which affect *conditional policy* decisions at runtime. These booleans and conditions have to be defined prior to policy deployment and new booleans/conditions can *not* be added after the policy has been loaded without recompiling and reloading the entire policy. The simplest example for such dynamic policies are booleans to switch between "enforcing mode" (i.e., access denials are enforced) and "permissive mode" (i.e., access denials are not enforced).

**Userspace Object Managers.** A powerful feature of SELinux is that its access control architecture can

[2] Developers may use the same UID (Shared UID, SUID) for their own apps. These apps will share the same sandbox.

be extended to security-relevant userspace daemons and services, which manage data (objects) independently from the kernel. Thus, such daemons and services are referred to as Userspace Object Managers (USOMs). They are responsible for assigning security contexts to the objects they manage, querying the SELinux security server for access control decisions, and enforcing these decisions. A prominent example for such USOMs on Linux systems is *GConf* [13].

## 2.4 SE Android

SE Android [48] prototypes SELinux for Android's Linux kernel and aims to demonstrate the value of SELinux in defending against various root exploits and application vulnerabilities. Specifically, it confines system Services and apps in different kernelspace security domains even isolating apps from one another by the use of the Multi-Level Security (MLS) feature of SELinux. To this end, the SE Android developers started writing an Android-specific policy from scratch. In addition, SE Android provides a few key security extensions tailored for the Android OS. For instance, it labels application processes with SELinux-specific security contexts which are later used in type enforcement. Moreover, since (in the majority of cases) it is a priori unknown during policy writing which apps will be installed on the system later, SE Android employs a mechanism to derive the security context of an app at install-time. Based on criteria, such as the requested permissions, apps are assigned a security type. This mapping from application meta-information to security types is defined in the SE Android policy.

Additionally, SE Android provides *limited* support for MAC policy enforcement at the Android middleware layer (MMAC) and we explain these particular features in Section 7.2 and provide a comparison to our FlaskDroid architecture.

# 3 Requirements Analysis for Android Security Architectures

## 3.1 Adversary Model

We consider a strong adversary with the goal to get access to sensitive data as well as to compromise system or third-party apps. Thus, we consider an adversary that is able to launch *software* attacks on different layers of the Android software stack.

### 3.1.1 Middleware Layer

Recently, different attacks operating at Android's middleware layer have been reported:
**Overprivileged 3$^{rd}$ party apps and libraries** threatening user privacy by adopting questionable privacy practices (e.g. WhatsApp [6] or Path [23]). Moreover, advertisement libraries, frequently included in 3$^{rd}$ party apps have been shown to exploit the permissions of their host app to collect information about the user [25].
**Malicious 3$^{rd}$ party apps** [22] leverage dangerous permissions to cause financial harm to the user (e.g., sending premium SMS) and exfiltrate user-private information [57, 40].
**Confused deputy attacks** concern malicious apps, which leverage unprotected interfaces of benign system [20, 41] and 3$^{rd}$ party [16, 56] apps (denoted deputies) to escalate their privileges.
**Collusion attacks** concern malicious apps that collude using covert or overt channels [8, 34] in order to gain a permission set which has not been approved by the user (e.g. the Soundcomber attack [46]).
**Sensory malware** leverages the information from onboard sensors, like accelerometer data, to derive privacy sensitive information, like user input [53, 12].

### 3.1.2 Root Exploits

Besides attacks at Android's middleware layer, various privilege escalation attacks on lower layers of the Android software stack have been reported [55, 40] which grant the attacker root (i.e., administrative) privileges and can be used to bypass the Android permission framework. For instance, he can bypass the ContactsProvider permission checks by accessing the contacts database file directly. Moreover, processes on Android executing with root privileges inherit all available permissions at middleware layer.

It should be noted that attacks targeting vulnerabilities of the Linux kernel are out of scope of this paper, since SE Android is a building block in our architecture (see Section 4) and as part of the kernel it is susceptible to kernel exploits.

## 3.2 Requirements

Based on our adversary model we derive the necessary requirements for an efficient and flexible access control architecture for mobile devices, focusing on the Android OS.
**Access Control on Multiple Layers.** Mandatory access control solutions at kernel level, such as SE Android [48] or Tomoyo [27], help to defend against or to constrain privilege escalation attacks on

the lower-levels of the OS [48]. However, kernel level MAC provides insufficient protection against security flaws in the middleware and application layers, and lacks the necessary high-level semantics to enable a fine-grained filtering at those layers [48, 47]. Access control solutions at middleware level [28, 15, 39, 9, 8] are able to address these shortcomings of kernel level MAC, but are, on the other hand, susceptible to low-level privilege escalation attacks.

Thus, a first requirement is to provide simultaneous MAC defenses at the two layers. Ideally, these two layers can be dynamically synchronized at run-time over mutual interfaces. At least, the kernel MAC is able to preserve security *invariants*, i.e., it enforces that any access to sensitive resources/functionality is always first mediated by the middleware MAC.

**Multiple stakeholders policies.** Mobile systems involve multiple stakeholders, such as the end-user, the device manufacturer, app developers, or other $3^{rd}$ parties (e.g., the end-user's employer). These stakeholders also store sensitive data on the device. Related work [39, 9] has proposed special purpose solutions to address the security requirements and specific problems of these parties. Naturally, the assets of different stakeholders are subject to different security requirements, which are not always aligned and might conflict. Thus, one objective for a generic MAC framework that requires handling policies of multiple stakeholders is to support (basic) policy reconciliation mechanisms [43, 35].

**Context-awareness.** The security requirements of different stakeholders may depend on the current context of the device. Thus, our architecture shall provide support for context-aware security policies.

**Support for different Use-Cases.** Our architecture shall serve as a basis for different security solutions applicable in a variety of use cases. For instance, by modifying the underlying policy our solution should be able to support different use cases (as shown in Section 5), such as the selective and fine-grained protection of app interfaces [39] or privacy-enhanced system Services and ContentProviders.

## 4 FlaskDroid Architecture

In this section, we provide an overview of our FlaskDroid architecture, elaborate in more detail on particular design decisions, and present the policy language employed in our system. Due to space constraints, we focus on the most important aspects and refer to our technical report [11] for more detailed information.

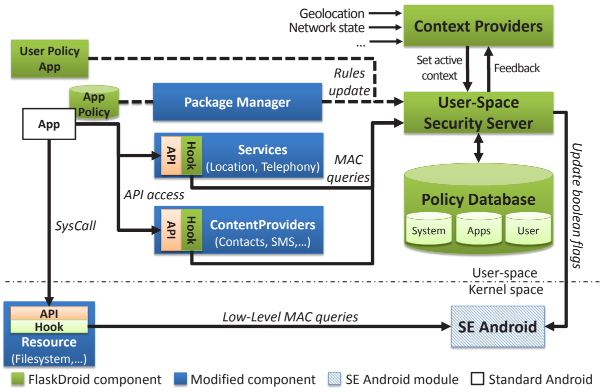

Figure 1: FlaskDroid Architecture

### 4.1 Overview

The high-level idea of FlaskDroid is inspired by the Flask security architecture [50], where various Object Managers at middleware and kernel-level are responsible for assigning security contexts to their objects. Objects can be, for instance, kernel resources such as Files or IPC and middleware resources such as Service interfaces, Intents, or ContentProvider data. On access to these objects by subjects (i.e., apps) to perform a particular operation, the managers enforce an access control decision that they request from a security server at their respective layer. Thus, our approach implements a *userspace security server*. Access control in FlaskDroid is implemented, as in SE Android (cf. Section 2), as *type enforcement*. However, in contrast to SE Android we extend our policy language with new features that are tailored to the Android middleware semantics (cf. Section 4.3). Moreover, to enable more dynamic policies, the policy checks in FlaskDroid depend also on the System State, which determines the actual security context of the objects and subjects at runtime.

Each security server is also responsible for the policy management for multiple stakeholders such as app developers, end-user, or $3^{rd}$ parties. A particular feature is that the policies on the two layers are synchronized at runtime, e.g., a change in enforcement in the middleware, must be supported/reflected at kernel-level. Thus, by decoupling the policy management and decision making from the enforcement points and consolidating the both layers, the goal of FlaskDroid's design is to provide fine-grained and highly flexible access control over operations on both middleware and kernel-level.

## 4.2 Architecture Components

Figure 1 provides an overview of our architecture. In the following, we will explain the individual components that comprise the FlaskDroid architecture.

### 4.2.1 SE Android Module

At the kernel-level, we employ stock SE Android [48] as a building block primarily for the following purposes: First, it is essential for hardening the Linux kernel [48] thereby preventing malicious apps from (easily) escalating their privileges by exploiting vulnerabilities in privileged (system) services. Even when an attack, usually with the intent of gaining *root* user privileges, is successful, SE Android can constrain the file-system privileges of the app by restricting the privileges of the root account itself. Second, it complements the policy enforcement at the middleware level by preventing apps from bypassing the middleware enforcement points (in Flask terminology defined as Userspace Object Managers (USOMs)), for example, accessing the contacts database file directly instead of going through the ContactsProvider app.

**Dynamic policies.** Using the dynamic policy support of SELinux (cf. Section 2.3) it is possible to reconfigure the access control rules at runtime depending on the current system state. Our Userspace Security Server (cf. Section 4.2.2) is hereby the trusted user space agent that controls the SELinux dynamic policies and can map system states and contexts to SELinux boolean variables (cf. Section 4.3). To this end, SE Android provides user space support (in particular *android.os.SELinux*).

### 4.2.2 Userspace Security Server

In our architecture, the Userspace Security Server is the central policy decision point for all userspace access control decisions, while the SE Android kernelspace security server is responsible for all kernelspace policy decisions. This approach provides a clear separation of security issues between the userspace and the kernelspace components. Furthermore, it enables at middleware level the use of a more dynamic policy schema (different from the more static SELinux policy language) which takes advantage of the rich semantics (e.g., contextual information) at that layer. Access control is implemented as type enforcement based on (1) the subject type (usually the type associated with the calling app), (2) the object type (e.g., *contacts_email* or the type associated with the callee app UID), (3) the object class (e.g., *contacts_data* or Intent), and (4) the operation on the object (e.g. *query*). The Userspace Security Server (USSS) is implemented as part of the Android system server (*com.android.server*) and comprises 3741 lines of Java code. It exposes an interface to the USOMs for requesting access control decisions over ICC (cf. Figure 1).

### 4.2.3 Userspace Object Managers

In FlaskDroid, middleware services and apps act as Userspace Object Managers (USOMs) for their respective objects. These services and apps can be distinguished into system components and $3^{rd}$ party components. The former, i.e., pre-installed services and apps, inevitably have to be USOMs to achieve the desired system security and privacy, while the latter can use interfaces provided by the Userspace Security Server to optionally act as USOMs.

Table 4 in Appendix B provides an overview of exemplary *system* USOMs in FlaskDroid and shows some typical operations each object manager controls. Currently, the USOMs implemented in FlaskDroid comprise 136 policy enforcement points. In the following, we explain how we instrumented selected components as Userspace Object Managers.

**PackageManagerService** is responsible for (un)installation of application packages. Furthermore, it is responsible for finding a preferred component for doing a task at runtime. For instance, if an app sends an Intent to display a PDF, the PackageManagerService looks for a preferred Activity able to perform the task.

As a Userspace Object Manager, we extend the PackageManagerService to assign consolidated middleware- and kernel-level app types to all apps during installation using criteria defined in the policy (cf. Section 4.3). This is motivated by the fact that at the time a policy is written, one cannot predict which $3^{rd}$ party apps will be installed in the future. Pre-installed apps are labeled during the phone's boot cycle based on the same criteria. More explicitly, we assign app types to the (shared) UIDs of apps, since (shared) UIDs are the smallest identifiable unit for application sandboxes. In addition, pre-defined UIDs in the system are reserved for particular system components[3] and we map these UIDs to pre-defined types (e.g., `aid_root_t` or `aid_audio_t`). Furthermore, we extend the logic for finding a preferred component to only consider apps which are allowed by the policy to perform the requested task.

**ActivityManagerService** is responsible for managing the stack of Activities of different apps, Activity life-cycle management, as well as providing the Intent

[3] These pre-defined UIDs on Android 4.0.4 are found in *system/core/include/private/android_filesystem_config.h*

broadcast system. As a USOM, the ActivityManagerService is responsible for labeling Activity and Intent objects and enforcing access control on them. Activities are labeled according to the apps they belong to, i.e., the UID of the application process that created the Activity. Subsequently, access control on the Activity objects is enforced in the ActivityStack subsystem of the ActivityManagerService. During operations that manipulate Activities, such as moving Activities to the foreground/background or destroying them, the ActivityStack queries the USSS in order to verify that the particular operations are permitted to proceed depending on the subject type (i.e., the calling app) and object type (i.e., the app owning the Activity being modified).

Similar to apps, Intents are labeled based on available meta-information, such as the action and category string or the sender app (cf. Section 4.3.1). To apply access control to Broadcast Intents, we followed a design pattern as proposed in [39, 9]. We modified the ActivityManagerService to filter out receivers which are not allowed to receive Intents of the previously assigned type (e.g., to prevent apps of lower security clearance from receiving Broadcasts by an app of a higher security clearance).

**Content Providers** are the primary means for apps to share data. This data can be accessed over a well-defined, SQL-like interface. As Userspace Object Managers, ContentProviders are responsible for assigning labels to the data entries they manage during insertion/creation of data and for performing access control on update, query, or deletion of entries. Two approaches for access control are supported: 1) at the API level by controlling access to the provider as a whole or 2) integrating it into the storage back-end (e.g., SQLite database) for more fine-grained per-data access control.

For approach 2), we implemented a design pattern for SQLite-based ContentProviders. Upon insertion or update of entries, we verify that the subject type of the calling app is permitted to perform this operation on the particular object type. To filter queries to the database we create one SQL View for each subject type and redirect the query of each calling app to the respective View for its type. Each View implements a filtering of data based on an access control table managed by the USSS which represents the access control matrix for subject/object types. This approach is well-suited for any SQLite-based ContentProvider and scales well to multiple stakeholders by using nested Views.

**Service** components of an app provide a particular functionality to other (possibly remote) components, which access the Service interface via ICC. To instantiate a Service as a Userspace Object Manager, we add access control checks when a (remote) component connects to the Service and on each call to Service functions exposed by the Service API. The developer of the Service can set the types of the service and its functions by adding type-tags to their definitions.

Service interfaces are exposed as Binder IPC objects that are generated based on an interface specification described in the Android Interface Definition Language (AIDL). We extended the lexer and parser of Android's AIDL tool to recognize (developer-defined) type tags on Service interfaces and function declarations. The AIDL code generator was extended to automatically insert policy checks for these types in the auto-generated Service code. Since the AIDL tool is used during build of the system as well as part of the SDK for app development, this solution applies to both system Services and $3^{rd}$ party app Services in the same way.

### 4.2.4 Context Providers

A context is an abstract term that represents the current security requirements of the device. It can be derived from different criteria, such as physical criteria (e.g., the location of the device) or the state of apps and the system (e.g., the app being currently shown on the screen). To allow for flexible control of contexts and their definitions, our design employs Context Providers. These providers come in form of plugins to our Userspace Security Server (see Figure 1) and can be arbitrarily complex (e.g., use machine learning) and leverage available information such as the network state or geolocation of the device to determine which contexts apply. Context Providers register Listener threads in the system to detect context changes similar to the approach taken in [15]. Each Context Provider is responsible for a distinct set of contexts, which it activates/deactivates in the USSS. Decoupling the context monitoring and definition from our policy provides that context definitions do not affect our policy language except for very simple declarations (as we will show in Section 4.3.1).

Moreover, the USSS provides feedback to Context Providers about the performed access control decisions. This is particularly useful when instantiating security models like [8, 15] in which access control decisions depend on previous decisions.

## 4.3 Policy

### 4.3.1 Policy Language and Extensions

We extend SELinux's policy semantics for *type enforcement* (cf. Section 2.3) with new default classes

Listing 1: Assigning types to apps and Intents

```
1 defaultAppType untrustedApp_t;
2 defaultIntentType untrustedIntent_t;
3
4 appType app_telephony_t {
        Package:package_name=com.android.phone; };
5
6 intentType intentLaunchHome_t {
7       Action:action_string=android.intent.action.MAIN;
8       Categories:category=android.intent.category.HOME;};
```

Listing 2: Linking booleans with contexts

```
1 bool phoneBooth_b = false;
2 kbool allowIPTablesExec_b = true;
3
4 switchBoolean {
5       context=phoneBooth_con;
6       auto_reverse=true;
7       phoneBooth_b=true;};
```

and constructs for expressing policies on both middleware and kernel-level. A recapitulation of the SELinux policy language is out of scope of this paper and we focus here on our extensions.

**New default classes.** Similar to classes at the kernel-level, like *file* or *socket*, we introduce new default classes and their corresponding operations to represent common objects at middleware level, such as Activity, Service, ContentProvider, and Intent. Operations for these classes are, for example, *query* a ContentProvider or *receive* an Intent.

**Application and Intent Types.** A further extension is the possibility to define criteria by which applications and Intents are labeled with a security type (cf. Listing 1). The criteria for apps can be, for instance, the application package name, the requested permissions or the developer signature. Criteria for assigning a type to Intent objects can be the Intent action string, category or receiving component. If no criteria matched, a default type is assigned to apps (line 1) and Intents (line 2), respectively.

**Context definitions and awareness.** We extend the policy language with an option to declare *contexts* to enable context-aware policies. Each declared context can be either *activated* or *deactivated* by a dedicated Context Provider (cf. Section 4.2).

To actually enable context-aware policies, we introduce in our policy language *switchBoolean* statements which map contexts to booleans, which in turn provide dynamic policies. Listing 2 presents the definition of booleans and *switchBoolean* statements. For instance, the switchBoolean statement in lines 4-9 defines that as soon as the context *phoneBooth_con* is active, the boolean *phoneBooth_b* has to be set to *true*. As soon as the *phoneBooth_con* context is deactivated, the *phoneBooth_b* boolean should be reset to its initial value (line 6). To map contexts to the kernel-level, we introduce *kbool* definitions (line 2), which point to a boolean at kernel level instead of adding a new boolean at middleware level. Changes to such kernel-mapped boolean values by *switchBoolean* statements trigger a call to the SELinux kernel module to update the corresponding SELinux boolean.

### 4.3.2 Support for Multiple Stakeholders

A particular requirement for the design of FlaskDroid is the protection of interests of different stakeholders. This requires that policy decisions consider the policies of all involved stakeholders. These policies can be pre-installed (i.e., system policy), delivered with apps (i.e., app developer policies), or configured by the user (e.g., *User Policy App* in Figure 1).

In FlaskDroid, $3^{rd}$ party app developers may optionally ship app-specific policies with their application packages and additionally choose to instrument their app components as Userspace Object Managers for their own data objects. FlaskDroid provides the necessary interfaces to query the Userspace Security Server for policy decisions as part of the SDK. These decisions are based on the app-specific $3^{rd}$ party policy, which defines custom appType statements to label subjects (e.g., other apps) and declares app-specific object types. To register app-specific policies, the PackageManagerService is instrumented such that it extracts policy files during app installation and injects them into the USSS.

A particular challenge when supporting multiple stakeholders is the reconciliation of the various stakeholders' policies. Different strategies for reconciliation are possible [43, 35] and generally supported by our architecture, based on namespaces and global/local type definitions. For instance, as discussed in [43], *all-allow* (i.e., all stakeholder policies must allow access), *any-allow* (i.e., only one stakeholder policy must allow access), *priority* (i.e., higher ranked stakeholder policies override lower ranked ones), or *consensus* (i.e., at least one stakeholder policy allows and none denies or vice versa). However, choosing the right strategy strongly depends on the use-case. For example, on a pure business smartphone without a user-private domain, the system (i.e., company) policy always has the highest priority, while on a private device a consensus strategy may be preferable.

We opted for a consensus approach, in which the *system* policy check is mandatory and must always consent for an operation to succeed.

## 5 Use-cases / Instantiations

In the following we will show how FlaskDroid can instantiate certain privacy and security protecting use-cases. More use-cases and concrete examples are provided in our technical report [11].

### 5.1 Privacy Enhanced System Services and Content Providers

System Services and ContentProviders are an integral part of the Android application framework. Prominent Services are, for instance, the LocationManager or the Audio Services and prominent ContentProviders are the contacts app and SMS/MMS app. By default, Android enforces permission checks on access to the interfaces of these Services and Providers.

**Problem description:** The default permissions are non-revocable and too coarse-grained and protect access only to the entire Service/Provider but not to specific functions or data. Thus, the user cannot control in a fine-grained fashion which sensitive data can be accessed how, when and by whom. Apps such as Facebook and WhatsApp have access to the entire contacts database although only a subset of the data (i.e., email addresses, phone numbers and names) is required for their correct functioning. On the other hand, recent attacks demonstrated how even presumably privacy-unrelated and thus unprotected data (e.g. accelerometer readings) can be misused against user's security and privacy [53, 12].

**Solution:** Our modified AIDL tool automatically generates policy checks for each Service interface and function in the system. We tagged selected query functions of the system AudioService, LocationManager, and SensorManager with specific security contexts (e.g., fineGrainedLocation_t as *object_type*, locationService_c as *object_class*, and getLastKnownLocation as *operation*) to achieve fine-grained access control on this information. Our policy states that calling functions of this object type is prohibited while the phone is in a security sensitive state. Thus, retrieving accelerometer information or recording audio is not possible when, e.g., the virtual keyboard/PIN pad is in the foreground or a phone call is in progress.

In Section 4.2.3 we explained how ContentProviders (e.g. the ContactsProvider) can act as User-space Object Managers. As an example, users can refine the system policy to further restrict access to their contacts' data. A user can, for instance, grant the Facebook app read access to their "friends" and "family" contacts' email addresses and names, while prohibiting it from reading their postal addresses and any data of other groups such as "work".

### 5.2 App Developer Policies (Saint)

Ongtang et al. present in [39] an access control framework, called *Saint*, that allows app developers to ship their apps with policies that regulate access to their apps' components.

**Problem description:** The concrete example used to illustrate this mechanism consists of a shopping app whose developer wants to restrict the interaction with other $3^{rd}$ party apps to only specific payment, password vault, or service apps. For instance, the developer specifies that that the password vault app must be at least version 1.2 or that a personal ledger app must not hold the Internet permission.

The policy rules for the runtime enforcement of Saint on Inter-Component communication (ICC) are defined as the tuple (Source, Destination, Conditions, State). Source defines the source app component of the ICC and optional parameters for an Intent object (e.g., action string). Destination describes similarly the destination app component of the ICC. Conditions are optional conjunctional conditions (e.g., permissions or signature key of the destination app) and State describes the system state (e.g., geolocation or bluetooth adapter state).

**Solution:** Instantiating Saint's runtime access control on FlaskDroid is achieved by mapping Saint's parameters to the type enforcement implemented by FlaskDroid. Thus, Source, Destination, and Conditions are combined into security types for the subject (i.e., source app) and object (i.e., destination app or Intent object). For instance, a specific type is assigned to an app with a particular signature and permission. If this app is source in the Saint policy, it is used as subject type in FlaskDroid policy rules; and if it is used as destination, it is used as object type. The object class and operation are directly derived from the destination app. The *system state* can be directly expressed by *booleans* and *switchBoolean* statements in the policy and an according Context Provider. Appendix A provides a concrete policy example for the instantiation of the above shopping app example.

## 6 Evaluation and Discussion

In this section we evaluate and discuss our architecture in terms of policy design, effectiveness, and performance overhead.

### 6.1 Policy

To evaluate our FlaskDroid architecture, we derived a basic policy that covers the pre-installed system

USOMs that we introduced in Section 4.2.3.

**Policy Assessment.** For FlaskDroid we are for now foremost interested in generating a *basic policy* to estimate the access control complexity that is inherent to our design, i.e., the number of new types, classes, and rules required for the *system* Userspace Object Managers. This basic policy is intended to lay the foundation for the development of a *good* policy, i.e., a policy that covers *safety*, *completeness*, and *effectiveness* properties. However, the development of a security policy that fulfills these properties is a highly complex process. For instance, on SELinux enabled systems the policies were incrementally developed and improved after the SELinux module had been introduced, even inducing research on verification of these properties [24]. A similar development can be currently observed for the SE Android policies which are written from scratch [48] and we envision inducing a similar research on development and verification of FlaskDroid policies.

**Basic Policy Generation.** To generate our basic policy, we opted for an approach that follows the concepts of TOMOYO Linux' learning phase[4] and other semi-automatic methods [42]. The underlying idea is to derive policy rules directly from observed application behavior. To generate a log of system application behavior, we leveraged FlaskDroid's audit mode, where policy checks are logged but not enforced. Under the assumption, that the system contained in this auditing phase only trusted apps, this trace can be used to derive policy rules.

To achieve a high coverage of app functionality and thus log all required access rights, we opted for testing with human user trials for the following reasons: First, automated testing has been shown to exhibit a potentially very low code coverage [24] and, second, Android's extremely event-driven and concurrent execution model complicates static analysis of the Android system [56, 24]. However, in the future, static analysis based (or aided) generation of access control rules is more preferable in order to cover also corner-cases of applications' control-flows.

The users' task was to thoroughly use the pre-installed system apps by performing various everyday tasks (e.g., maintaining contacts, writing SMS, browsing the Internet, or using location-based services). To analyze interaction between apps, a particular focus of the user tasks was to leverage inter-app functionality like sharing data (e.g., copying notes from a website into an SMS). For testing, the users were handed out Galaxy Nexus devices running FlaskDroid with a *No-allow-rule* policy. This is a manually crafted policy containing only the required subject/object types, classes and operations for the USOMs in our architecture, but no allow rules. The devices were also pre-configured with test accounts (e.g., EMail) and test data (e.g., fake contacts).

Using the logged access control checks from these trials, we derived 109 access control rules required for the correct operation of the system components (as observed during testing), which we learned to be partially operationally dependent on each other. Our pre-installed middleware policy contained 111 types and 18 classes for a fine-granular access control to the major system Services and ContentProviders (e.g., ContactsProvider, LocationManager, PackageManagerService, or SensorManager). These rules (together with the above stated type and object definitions) constitute our *basic policy.* Although SELinux policies cannot be directly compared to our policy, since they target desktop operating systems, the difference in policy complexity (which is in the order of several magnitudes [11]) underlines that the design of mobile operating systems facilitates a clearer mandatory access control architecture (e.g., separation of duties). This profits an easier policy design (as supported by the experiences from [54, 36]).

**$3^{rd}$ Party Policies.** The derived basic policy can act as the basis on top of which additional user, $3^{rd}$ party, and use-case specific policies can be deployed (cf. Section 5). In particular, we are currently working on extending the basic policy with types, classes and allow rules for popular apps, such as WhatsApp or Facebook, which we further evaluated w.r.t. user's privacy protection (cf. Section 6.2). A particular challenge is to derive policies which on the one hand protect the user's privacy but on the other hand preserve the intended functionality of the apps. Since the user privacy protection strongly depends on the subjective security objectives of the user, this approach requires further investigation on how the user can be involved in the policy configuration [58].

However, as discussed in Sections 3 and 4.2.2, multiple policies by different stakeholders with potentially conflicting security objectives require a reconciliation strategy. Devising a general strategy applicable to all use-cases and satisfying all stakeholders is very difficult, but use-case specific strategies are feasible [44, 29]. In our implementation, we opted for a consensus approach, which we successfully applied during implementation of our use-cases (cf. Section 5). We explained further strategies in Section 4.3.2.

[4] http://tomoyo.sourceforge.jp/2.2/learning.html.en

| Attack | Test |
|---|---|
| Root Exploit | mempodroid Exploit |
| App executed by root | Synthetic Test App |
| Over-privileged and Information-Stealing Apps | Known malware<br>Synthetic Test App<br>WhatsApp v2.8.4313<br>Facebook v1.9.1 |
| Sensory Malware | Synthetic Test App [53, 12, 46] |
| Confused Deputy | Synthetic Test App |
| Collusion Attack | Synthetic Test Apps [46] |

Table 1: List of attacks considered in our testbed

## 6.2 Effectiveness

We decided to evaluate the effectiveness of FlaskDroid based on empirical testing using the security models presented in Section 5 as well as a testbed of known malware retrieved from [55, 3] and synthetic attacks (cf. Table 1). Alternative approaches like static analysis [18] would benefit our evaluation but are out of scope of this paper and will be addressed separately in future work.

**Root exploits.** SE Android successfully mitigates the effect of the *mempodroid* attack. While the exploit still succeeds in elevating its process to root privileges, the process is still constrained by the underlying SE Android policy to the limited privileges granted to the root user [48].

**Malicious apps executed by root.** While SE Android constrains the file-system privileges of an app process executed with root UID, this process still inherits all Permissions at middleware level. In FlaskDroid, the privileges of apps running with this omnipotent UID are restricted to the ones granted by the system policy to root (cf. `aid_root_t` in Section 4.2.3). During our user tests, we had to define only one allow rule for the `aid_root_t` type on the middleware layer, which is not surprising, since usually Android system or third-party apps are not executed by the root user. Thus, a malicious app gaining root privileges despite SE Android, e.g., using the *mempodroid* exploit [48], is in FlaskDroid restricted at both kernel and middleware level.

**Over-privileged and information stealing apps.** We verified the effectiveness of FlaskDroid against over-privileged apps using a) a synthetic test app which uses its permissions to access the ContactsProvider, the LocationManager and the SensorManager as $3^{rd}$ party apps would do; b) malware such as Android.Loozfon [2] and Android.Enesoluty [1] which steal user private information; and c) unmodified apps from Google Play, including the popular WhatsApp messenger and Facebook apps. In all cases, a corresponding policy on FlaskDroid successfully and gracefully prevented the apps and malware from accessing privacy critical information from sources such as the ContactsProvider or LocationManager.

**Sensory malware.** To mitigate sensory malware like *TapLogger* [53] and *TouchLogger* [12], we deployed a context-aware FlaskDroid policy which causes the SensorManager USOM to filter acceleration sensor information delivered to registered *SensorListeners* while the on-screen keyboard is active. Similarly, a second policy prevents the *SoundComber* attack [46] by denying any access to the audio record functionality implemented in the MediaRecorderClient USOM while a call is in progress.

**Confused deputy and collusion attacks.** Attacks targeting confused deputies in system components (e.g. `SettingsAppWidgetProvider` [41]) are addressed by fine-grained access control rules on ICC. Our policy restricts which app types may send (broadcast) Intents reserved for system apps.

Collusion attacks are in general more challenging to handle, especially when covert channels are used for communication. Similar to the mitigation of confused deputies, a FlaskDroid policy was used to prohibit ICC between colluding apps based on specifically assigned app types. However, to address collusion attacks *efficiently*, more flexible policies are required. We already discussed in Section 4.2.4 a possible approach to instantiate *XManDroid* [8] based on our Context Providers and we elaborate in the subsequent Section 6.3 on particular challenges for improving the mitigation of collusion attacks.

## 6.3 Open Challenges and TCB

**Information flows within apps.** Like any other access control system, e.g., SELinux, exceptions for which enforcement falls short concern attacks which are licit within the policy rules. Such shortcomings may lead to unwanted information leakage. A particular challenge for addressing this problem and controlling access and separation (*non-interference*) of security relevant information are information flows within apps. Access control frameworks like FlaskDroid usually operate at the granularity of application inputs/outputs but do not cover the information flow within apps. For Android security, this control can be crucial when considering attacks such as collusion attacks and confused deputy attacks. Specifically for Android, taint tracking based approaches [19, 28, 45] and extensions to Android's IPC mechanism [17] have been proposed. To which extend these approaches could augment the coverage and hence effectiveness of FlaskDroid has to be explored in future work.

**User-centric and scalable policies.** While

FlaskDroid is a sophisticated access control framework for enforcing security policies and is already now valuable in specific scenarios with fixed policies like business phones or locked-down devices [11], a particular challenge of the forthcoming policy engineering are user-centric and scalable policies for off-the-shelf end-user devices. Although expert-knowledge can be used to engineer policies for the static components of the system, similar to common SELinux-enabled distributions like Fedora, an orthogonal, open research problem is how to efficiently determine the individual end-users' security and privacy requirements and how to map these requirements scalable to FlaskDroid policy rules w.r.t. the plethora of different apps available. To this end, we started exploring approaches to provide the end-user with tools that abstract the underlying policies [10]. Furthermore, the policy-based classification of apps at install-time applied in FlaskDroid could in the future be augmented by different or novel techniques from related fields, e.g., role-mining for RBAC systems [51], to assist the end-user in his decision processes.

**Trusted Computing Base.** Moreover, while SE Android as part of the kernel is susceptible to kernel-exploits, our middleware extensions might be compromised by attacks against the process in which they execute. Currently our SecurityServer executes within the scope of the rather large Android system server process. Separating the SecurityServer as a distinct system process with a smaller attack surface (smaller TCB) can be efficiently accomplished, since there is no strong functional inter-dependency between the system server and FlaskDroid's SecurityServer.

## 6.4 Performance Overhead

**Middleware layer.** We evaluated the performance overhead of our architecture based on the *No-allow-rule* policy and the basic policy presented in Section 6.1 using a Samsung Galaxy Nexus device running FlaskDroid. Table 2 presents the mean execution time $\mu$ and standard deviation $\sigma$ for performing a policy check at the middleware layer in both policy configurations (measured in $\mu s$) as well as the average memory consumption (measured in $MB$) of the process in which our Userspace Security Server executes (i.e., the system server). Mean execution time and standard deviation are the amortized values for both cached and non-cached policy decisions.

In comparison to permission checks on a vanilla Android 4.0.4 both the imposed runtime and memory overhead are acceptable. The high standard deviation is explained by varying system loads, however, Figure 2 presents the cumulative frequency distribution for our policy checks. The shaded area represents the 99.33% confidence interval for our basic policy with a maximum overhead of $2ms$.

| | $\mu$ (in $\mu s$) | $\sigma$ (in $\mu s$) | memory (in MB) |
|---|---|---|---|
| **FlaskDroid** | | | |
| No-allow-rule | 329.505 | 780.563 | 15.673 |
| Basic policy | 452.916 | 4887.24 | 16.184 |
| **Vanilla Android 4.0.4** | | | |
| Permission check | 330.800 | 8291.805 | 15.985 |

Table 2: Runtime and memory overhead

| | $\mu$ (in $ms$) | $\sigma$ (in $ms$) |
|---|---|---|
| FlaskDroid (Basic policy) | 0.452 | 4.887 |
| XManDroid [8] (Amortized) | 0.532 | 2.150 |
| TrustDroid [9] | 0.170 | 1.910 |

Table 3: Performance comparison to related works

In comparison to closest related work [8, 9] (cf. Section 7), FlaskDroid achieves a very similar performance. Table 3 provides an overview of the average performance overhead of the different solutions. *TrustDroid* [9] profits from the very static policies it enforces, while FlaskDroid slightly outperforms *XManDroid* [8]. However, it is hard to provide a completely fair comparison, since both TrustDroid and XManDroid are based on Android 2.2 and thus have a different baseline measurement. Both [8, 9] report a baseline of approximately $0.18ms$ for the default permission check, which differs from the $0.33ms$ we observed in Android 4.0.4 (cf. Table 2).

**Kernel layer.** The impact of SE Android on Android system performance has been evaluated previously by its developers [48]. Since we only minimally add/modify the default SE Android policy to cater for our use-cases (e.g., new booleans), the negligible performance overhead presented in [48] still applies to our current implementation.

# 7 Related Work

## 7.1 Mandatory Access Control

The most prominent MAC solution is SELinux [33] and we elaborated on it in detail in our Background and Requirements Sections 2 and 3. Specifically for mobile platforms, related work [54, 36] has investigated the placement of SELinux enforcement hooks in the operating system and user-space services on OpenMoko [36] and the LiMo (Linux Mobile) platform [54]. Our approach follows along these ideas but for the Android middleware.

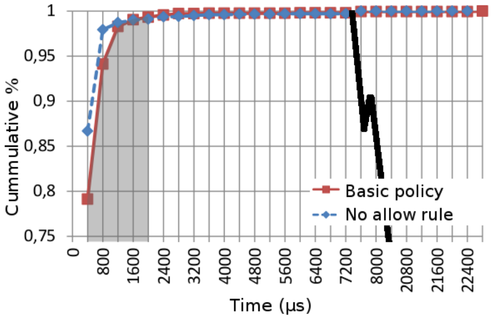

Figure 2: CDF of the performance overhead. Shaded area represents the 99.33 confidence interval for checks with *Basic policy*.

Also TOMOYO Linux [27], a path-based MAC framework, has been leveraged in Android security extensions [8][9]. Although TOMOYO supports more easily policy updates at runtime and does not require extended file system attributes, SELinux is more sophisticated, supports richer policies, and covers more object classes [5].

However, as we state in Section 3, low-level MAC alone is insufficient. In this paper we show how to extend the SE Android security architecture into the Android middleware layer for policy enforcement.

### 7.2 SE Android MMAC

The SE Android project was recently extended by different mechanisms for mandatory access control at Android's middleware layer [47], denoted as MMAC:
**Permission revocation** is a simple mechanism to dynamically revoke permissions by augmenting the default Android permission check with a policy driven check. When necessary, this additional check overrules and negates the result of the default check.

However, this permission revocation is in almost all cases unexpected for app developers, which rely on the fact that if their app has been installed, it has been granted all requested permissions. Thus, developers very often omit error handling code for permission denials and hence unexpectedly revoking permissions easily leads to application crashes.

In FlaskDroid, policy enforcement also effectively revokes permissions. However, we use USOMs which integrate the policy enforcement into the components which manage the security and privacy sensitive data. Thus, our USOMs apply enforcement mechanisms that are *graceful*, i.e., they do not cause unexpected behavior that can cause application crashes. Related work (cf. Section 7.3) introduced some of these graceful enforcement mechanisms, e.g., filtering table rows and columns from ContentProvider responses [58, 15, 28, 8, 9].
**Intent MAC** protects with a white-listing enforcement the delivery of Intents to Activities, Broadcast Receivers, and Services. Technically, this approach is similar to prior work like [58, 8, 9]. The white-listing is based on attributes of the Intent objects (e.g., the value of the action string) and the security type assigned to the Intent sender and receiver apps.

In FlaskDroid, we apply a very similar mechanism by assigning Intent objects a security type, which we use for type enforcement on Intents. While we acknowledge, that access control on Intents is important for the overall coverage of the access control, Intent MAC alone is insufficient for policy enforcement on inter-app communications. A complete system has to consider also other middleware communications channels, such as Remote Procedure Calls (RPC) to Service components and to ContentProviders. By instrumenting these components as USOMs and by extending the AIDL compiler (cf. Section 4.2) to insert policy enforcement points, we address these channels in FlaskDroid and provide a non-trivial complementary access control to Intent MAC.
**Install-time MAC** performs, similar to *Kirin* [20], an install-time check of new apps and denies installation when an app requests a defined combination of permissions. The adverse permission combinations are defined in the SE Android policy.

While FlaskDroid does not provide an install-time MAC, we consider this mechanism orthogonal to the access control that FlaskDroid already provides and further argue that it could be easily integrated into existing mechanisms of FlaskDroid (e.g., by extending the install-time labeling of new apps with a blacklist-based rejection of prohibited app types).

### 7.3 Android Security Extensions

In the recent years, a number of security extensions to the Android OS have been proposed.

Different approaches [38, 37, 15, 39] add mandatory access control mechanisms to Android, tailored for specific problem sets such as providing a DRM mechanism (*Porscha* [38]), providing the user with the means to selectively choose the permissions and runtime constraints each app has (*APEX* [37] and *CRePE* [15]), or fine-grained, context-aware access control to enable developers to install policies to protect the interfaces of their apps (*Saint* [39]). Essentially all these solutions extend Android with MAC at the middleware layer. The explicit design goal of our architecture was to provide an ecosystem that is flexible enough to instantiate those related works based on policies (as demonstrated in Section 5 at the example of *Saint*) and additionally providing the benefit of a consolidated kernel-level MAC.

The pioneering framework *TaintDroid* [19] introduced the tracking of tainted data from sensible sources on Android and successfully detected unauthorized information leakage. The subsequent *AppFence* architecture [28] extended TaintDroid with checks that not only detect but also prevent such unauthorized leakage. However, both TaintDroid and AppFence do not provide a generic access control framework. Nevertheless, future work could investigate their applicability in our architecture, e.g., propagating the security context of data objects. The general feasibility of such "context propagation" has been shown in the *MOSES* [45] architecture.

*Inlined Reference Monitors* (IRM) [52, 7, 30] place policy enforcement code for access control directly in $3^{rd}$ party apps instead of relying on a system centric solution. An unsolved problem of *inlined* monitoring in contrast to a system-centric solution is that the reference monitor and the potentially malicious code share the same sandbox and that the monitor is *not* more privileged than the malicious code and thus prone to compromise.

The closest related work to FlaskDroid with respect to a two layer access control are the *XManDroid* [8] and *TrustDroid* [9] architectures. Both leverage TOMOYO Linux as kernel-level MAC to establish a separate security domain for business apps [9], or to mitigate collusion attacks via kernel-level resources [8]. Although they cover MAC enforcement at both middleware and kernel level, both systems support only a very static policy tailored to their specific purposes and do not support the instantiation of different use-cases. In contrast, FlaskDroid can instantiate the XManDroid and TrustDroid security models by adjusting policies. For instance, different security types for business and private apps could be assigned at installation time, and boolean flags can be used to dynamically prevent two apps from communicating if this would form a collusion attack.

## 8 Conclusion

In this paper, we present the design and implementation of FlaskDroid, a policy-driven generic two-layer MAC framework on Android-based platforms. We introduce our efficient policy language that is tailored for Android's middleware semantics. We show the flexibility of our architecture by policy-driven instantiations of selected security models, including related work (Saint) and privacy-enhanced system components. We demonstrate the applicability of our design by prototyping it on Android 4.0.4. Our evaluation shows that the clear API-oriented design of Android benefits the effective and efficient implementation of a generic mandatory access control framework like FlaskDroid.

## Availability

The source code for FlaskDroid is available online at `http://www.flaskdroid.org`.

## References

[1] Android.Enesoluty | Symantec. `http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99`.

[2] Android.Loozfon | Symantec. `http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99`.

[3] Contagio Mobile. `http://contagiominidump.blogspot.de/`.

[4] Gartner Says Worldwide Mobile Phone Sales Declined 1.7 Percent in 2012. `http://www.gartner.com/newsroom/id/2335616`.

[5] TOMOYO Linux Wiki: How is TOMOYO Linux different from SELinux and AppArmor? `http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison`.

[6] WhatsApp reads your phone contacts and is breaking privacy laws. `http://www.digitaltrends.com/mobile/whatsapp-breaks-privacy-laws/`.

[7] Backes, M., Gerling, S., Hammer, C., and von Styp-Rekowsky, P. Appguard - enforcing user requirements on android apps. In *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2013), Springer-Verlag.

[8] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards taming privilege-escalation attacks on android. In *NDSS* (2012), The Internet Society.

[9] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and lightweight domain isolation on android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM)* (2011), ACM.

[10] Bugiel, S., Heuser, S., and Sadeghi, A.-R. myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. Tech. Rep. TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt, Nov. 2012.

[11] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware. Tech. Rep. TUD-CS-2012-0231, Center for Advanced Security Research Darmstadt, December 2012.

[12] Cai, L., and Chen, H. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *6th USENIX conference on Hot topics in security (HotSec)* (2011), USENIX Association.

[13] Carter, J. Using gconf as an example of how to create an userspace object manager, 2007.

[14] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In *MobiSys* (2011), ACM.

[15] Conti, M., Nguyen, V. T. N., and Crispo, B. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC)* (2010), Springer-Verlag.

[16] Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC)* (2010), Springer-Verlag.

[17] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. Quire: Lightweight provenance for smartphone operating systems. In *USENIX Security* (2011), USENIX Association.

[18] Edwards, A., Jaeger, T., and Zhang, X. Runtime verification of authorization hook placement for the Linux security modules framework. In *CCS* (2002), ACM.

[19] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010), USENIX Association.

[20] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *CCS* (2009), ACM.

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

[22] F-Secure Labs. Mobile Threat Report: Q3 2012, 2012.

[23] Federal Trade Commission. Path social networking app settles FTC charges it deceived consumers and improperly collected personal information from users' mobile address books. `http://www.ftc.gov/opa/2013/02/path.shtm`, Jan. 2013.

[24] Gilbert, P., Chun, B.-G., Cox, L. P., and Jung, J. Vision: automated security validation of mobile apps at app markets. In *2nd international workshop on Mobile cloud computing and services (MCS)* (2011), ACM.

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

[26] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. Verifying information flow goals in security-enhanced linux. *Journal on Computer Security 13*, 1 (Jan. 2005), 115–134.

[27] Harada, T., Horie, T., and Tanaka, K. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference* (2004).

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

[29] Hu, H., Ahn, G.-J., and Kulkarni, K. Detecting and resolving firewall policy anomalies. *IEEE Transactions on Dependable and Secure Computing 9*, 3 (2012), 318–331.

[30] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM)* (2012), ACM.

[31] Kostiainen, K., Reshetova, E., Ekberg, J.-E., and Asokan, N. Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In *CODASPY* (2011), ACM.

[32] Lineberry, A., Richardson, D. L., and Wyatt, T. These aren't the permissions you're looking for. BlackHat USA 2010. `http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf`, 2010.

[33] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *FREENIX Track: USENIX Annual Technical Conference* (2001), USENIX Association.

[34] Marforio, C., Ritzdorf, H., Francillon, A., and Capkun, S. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC* (2012), ACM.

[35] McDaniel, P., and Prakash, A. Methods and limitations of security policy reconciliation. In *S&P* (2002), IEEE Computer Society.

[36] Muthukumaran, D., Schiffman, J., Hassan, M., Sawani, A., Rao, V., and Jaeger, T. Protecting the integrity of trusted applications in mobile phone systems. *Security and Communication Networks 4*, 6 (2011), 633–650.

[37] Nauman, M., Khan, S., and Zhang, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ASIA CCS* (2010), ACM.

[38] Ongtang, M., Butler, K., and McDaniel, P. Porscha: Policy oriented secure content handling in Android. In *ACSAC* (2010), ACM.

[39] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in Android. In *ACSAC* (2009), IEEE Computer Society.

[40] Porter Felt, A., Finifter, M., Chin, E., Hanna, S., and Wagner, D. A survey of mobile malware in the wild. In *1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2011), ACM.

[41] Porter Felt, A., Wang, H., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *USENIX Security* (2011), USENIX Association.

[42] Provos, N. Improving host security with system call policies. In *USENIX Security* (2003), USENIX Association.

[43] Rao, V., and Jaeger, T. Dynamic mandatory access control for multiple stakeholders. In *SACMAT* (2009), ACM.

[44] Reeder, R. W., Bauer, L., Cranor, L. F., Reiter, M. K., and Vaniea, K. More than skin deep: measuring effects of the underlying model on access-control system usability. In *International Conference on Human Factors in Computing Systems (CHI)* (2011), ACM.

[45] Russello, G., Conti, M., Crispo, B., and Fernandes, E. MOSES: supporting operation modes on smartphones. In *SACMAT* (2012), ACM.

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

[47] Smalley, S. Middleware MAC for android. `http://kernsec.org/files/LSS2012-MiddlewareMAC.pdf`, Aug. 2012.

[48] Smalley, S., and Craig, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS* (2013), The Internet Society.

[49] Smith, C. Privacy flaw in skype android app exposed. `http://www.t3.com/news/privacy-flaw-in-skype-android-app-exposed/`.

[50] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. The Flask security architecture: System support for diverse security policies. In *USENIX Security* (1999), USENIX Association.

[51] Vaidya, J., Atluri, V., and Warner, J. RoleMiner: mining roles using subset enumeration. In *CCS* (2006), ACM.

[52] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for android applications. In *USENIX Security* (2012), USENIX Association.

[53] Xu, Z., Bai, K., and Zhu, S. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *WiSec* (2012), ACM.

[54] Zhang, X., Seifert, J.-P., and Aciiçmez, O. SEIP: simple and efficient integrity protection for open mobile platforms. In *International conference on Information and communications security (ICICS)* (2010), Springer-Verlag.

[55] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *S&P* (2012), IEEE Computer Society.

[56] Zhou, Y., and Jiang, X. Detecting passive content leaks and pollution in android applications. In *NDSS* (2013), The Internet Society.

[57] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS* (2012), The Internet Society.

[58] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. Taming information-stealing smartphone applications (on android). In *TRUST* (2011), Springer-Verlag.

# A Concrete Instantiation of Saint policies with FlaskDroid

Listing 3 shows an instantiation of the developer policy in [39] on FlaskDroid. This policy is deployed by the shopping app and thus `self_t` refers to the shopping app. We define types `app_trustedPayApp_t`, `app_trustedPayApp_t`, `app_noInternetPerm_t` (lines 1-3 and lines 8-16) for the specific apps the shopping app is allowed to interact with and describe some of the allowed interactions by means of `Intent` types `intent_actionPay_t` and `intent_recordExpense_t` (lines 5-6 and lines 18-24). Afterwards, we declare access control rules that reflect the policy described in [39] (lines 26-28). For instance, the rule in line 26 defines that the shopping app is allowed to send an `Intent` with action string `ACTION_PAY` only to an app with type `app_trustedPayApp_t` (line 20), which in turn is only assigned to apps with the developer signature `308201...` (line 9).

Listing 3: Policy deployed by the shopping app, showing an instantiation of the Saint [39] runtime policy example.

```
1  type app_trustedPayApp_t;
2  type app_trustedPWVault_t;
3  type app_noInternetPerm_t;
4
5  type intent_actionPay_t;
6  type intent_recordExpense_t;
7
8  appType app_trustedPayApp_t {
9   Developer:signature=308201...; };
10
11 appType app_trustedPWVault_t {
12  Package:package_name=com.secure.passwordvault;
13  Package:min_version=1.2; };
14
15 appType app_noInternetPerm_t {
16 Package:permission=~android.permission.INTERNET; };
17
18 intentType intent_actionPay_t {
19  Action:action_string=ACTION_PAY;
20  Components:receiver_type=app_trustedPayApp_t; };
21
22 intentType intent_recordExpense_t {
23  Action:action_string=RECORD_EXPENSE;
24  Components:receiver_type=app_noInternetPerm_t; };
25
26 allow self_t intent_actionPay_t: intent_c { send };
27 allow self_t app_trustedPWVault_t: any { any };
28 allow self_t intent_recordExpense_t: intent_c { send };
```

# B Userspace Object Managers

| USOM | Example operations |
|---|---|
| **Service USOMs** | |
| PackageManagerService | getPackageInfo findPreferredActivity getInstalledApplications installPackage |
| ActivityManagerService | startActivity moveTask grantURIPermission sendBroadcast registerBroadcastReceiver |
| AudioService | setStreamVolume setVibrateSetting |
| PowerManagerService | acquireWakeLock isScreenOn reboot preventScreenOn |
| SensorManager | getSensorList getDefaultSensor |
| LocationManagerService | requestLocationUpdates addProximityAlert getLastKnownLocation |
| SMSManager | copyMessageToIcc deleteMessageFromIcc sendTextMessage |
| TelephonyManager | getCellLocation getDeviceId getCellLocation |
| **ContentProvider USOMs** | |
| ContactsProvider2 | query insert update delete writeAccess readAccess |
| MMSSMSProvider | query insert update delete |
| TelephonyProvider | query insert update delete |
| SettingsProvider | query insert update delete |

Table 4: Exemplary System USOMs