Technical Report

Nr. TUD-CS-2012-0231
December 5, 2012
(Revised February 20, 2013)

CASED

Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware

Authors
Sven Bugiel, Stephan Heuser, Ahmad-Reza Sadeghi

Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware

Sven Bugiel[*], Stephan Heuser[†], Ahmad-Reza Sadeghi[*†‡],

[*]System Security Lab / CASED, Technische Universität Darmstadt, Germany

[†]Fraunhofer SIT, Darmstadt, Germany

[‡]Intel Collaborative Research Institute for Secure Computing at Technische Universität Darmstadt, Germany

ABSTRACT

Smartphones and tablets have become an integral part of our daily life. They increasingly store and process security and privacy sensitive data which makes them attractive targets for attackers. In particular for the popular Android OS, a number of security extensions have been proposed that target specific security and privacy problems caused by Android's lack of a fine-grained, dynamic and system-wide mandatory access control.

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 in a consolidated manner. The synchronization of policy enforcement between the 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: one is from the existing work (*Saint*) while the other one is 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.

On the downside, the increased usage of smart devices in security and privacy critical contexts (e.g., mobile banking) as well as the storage and processing of sensitive data on them introduce new security and privacy risks. Android has been shown to be vulnerable to a number of different attacks such as malicious apps and libraries that misuse their privileges [89, 62, 34] or even utilize root-exploits [87, 62] to extract security and privacy sensitive information; taking advantage of unprotected interfaces [18, 14, 85, 48] and files [76]; confused deputy attacks [20]; and collusion attacks [68, 51].

Solutions.

On the other hand, Android's open-source nature has made it very appealing to academic and industrial security research. Its system architecture and security mechanisms have been thoroughly scrutinized, and various extensions to Android's access control framework have been proposed to address particular problem sets such as protection of the users' privacy [23, 40, 19, 84, 8, 44]; application centric security such as *Saint* enabling developers to protect their application interfaces [59]; establishing isolated domains (usage of the phone in private and corporate context) [12]; mitigation of collusion attacks [11], and extending Android's Linux kernel with Mandatory Access Control [55].

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. Only few solutions consider both layers [11, 12], 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 [86, 54] 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 [46].

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* [59] or privacy-enhanced system components [90], 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 [50] on desktop systems [86].

Our Contribution.

In this paper, we present a security architecture for the Android OS, that addresses these challenges by designing a security framework that can serve as an appropriate ecosystem for different security and privacy protecting models. Our design is inspired by the concepts of the Flask architecture [78]: 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. We aim to demonstrate that such complex and at the same time flexible security frameworks can be more efficiently integrated into the design of today's smart device operating systems compared to traditional non-mobile systems. In particular, our contributions are:

1. *System-wide security framework.* We present the design and implementation of a security framework that extends the Android OS and 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 level. We base our implementation on SE Android [55], which has already been partially merged into the official Android source-code by Google and is expected to become a standard feature of future Android versions [1].

2. *Security policy.* We developed and integrated type enforcement at Android's middleware layer, which has completely different semantic than the kernel level, and its synchronization with the kernel enforcement at runtime. We present our policy language that leverages the API-oriented design of Android. In particular, as we will discuss in detail, privacy and security critical functionalities on Android are concentrated in a small number of system components forming single access points to the corresponding functionality. This is an important design aspect that significantly simplifies policy authoring and achieving a high code coverage.

3. *Use-cases.* By adjusting the policies, our security framework can instantiate attack-specific related work and use-cases that go beyond this related work. We demonstrate with example use-cases how the different semantics of each layer (middleware/kernel) can be used to provide more powerful and efficient security policies that are jointly enforced on those layers.

4. *Efficiency and effectiveness.* We successfully evaluate the efficiency end effectiveness of our solution by testing it against a testbed of known attacks, and by deriving an example system policy that can function as a baseline policy which allows for instantiating further use-cases.

The remainder of this paper is structured as follows. In Section 2 we provide technical background information. We discuss our adversary model, requirements analysis and challenges in Section 3. In Section 4 we present the FlaskDroid architecture and policy language and show its implementation in Section 5. We demonstrate in Section 6 use-cases for FlaskDroid's applicability and evaluate our architecture in Section 7. We discuss related work in Section 8 and conclude this paper in Section 9.

2. BACKGROUND AND CHALLENGES

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

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, which is responsible for basic operating system services, such as process scheduling, memory management, file system support and network access. Certain subsystems have been modified by Google to be compatible with the scarce resources of mobile devices, e.g. available power and memory [10].

Furthermore, Android consists of an application framework implementing (most of) the Android API. System Services and libraries, such as the radio interface layer and native code libraries, are implemented in C/C++. Android also provides a Java virtual machine executing the Dalvik bytecode format, which has been tailored to the specific needs of resource-constrained devices. Higher-level services, such as System settings, the Clipboard, the Wifi-, Location- and Audiomanager, are implemented in Java. Together, these components comprise the Android middleware layer.

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 rely on the previously described kernel and middleware Services. Android ships with standard applications completing the implementation of the Android API, such as a Contacts (database) Provider. Additional apps can be installed by the user from, for example, Google Play [2].

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

[2]`https://play.google.com/store`

Android applications consist of certain components: Activities (user interfaces), Services (background processes), ContentProviders (SQL-like databases), and Broadcast Receivers (mailboxes for broadcast messages). Applications 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), a term abstractly describing IPC between application components using a light-weight implementation of OpenBinder [60, 26], simply denoted as *Binder*.

ICC on Android is usually implemented using a domain specific language, the *Android Interface Definition Language* (AIDL), which is compiled into Java stub and skeleton code using the AIDL compiler. 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 (ab)uses the Linux discretionary access control (DAC) mechanisms for application sandboxing: Every application installed on the phone is assigned a unique user identifier (UID) during installation[3]. Every process belonging to the application is executed in the context of this UID, which effectively mediates access to low level resources (e.g. files). Based on this mechanism, applications are assigned a private storage area on the device's internal flash memory. Low-level IPC (e.g. using pipes or domain sockets) is also controlled using Linux DAC.

Permissions. In addition, access control is applied to ICC by using *Permissions* [33]: Labels assigned to applications at installation 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 the default permissions defined by the Android OS itself, application 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 [32]).

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.

Application signing. X.509 certificates are used to ensure application integrity and authenticity. Additionally, the developer signatures are used to enforce a *same origin policy* for application updates. However, there is no mandatory hierarchical public-key infrastructure in place, and as such most developers use self-signed certificates. The trust model of the market is based on reputation.

2.3 SELinux

Security Enhanced Linux (SELinux) is an instantiation of the Flask security architecture [78] and implements a policy-driven mandatory access control (MAC) framework for the Linux kernel. An essential design decision of its architecture is that policy decision making is decoupled from the policy enforcement logic. SELinux uses the Linux Security Module (LSM) [83] architecture, which provides various access control enforcement points for low-level resources, such as files, local IPC, or memory protections. When an LSM hook is triggered (e.g., a file is opened), the SELinux LSM module requests a policy decision from a *security server* in the kernel which manages the policy rules and contains the access decision logic. Depending on the security server's decision, the SELinux security module denies or allows the operation to proceed. To maintain the security server (e.g., update the policy), SELinux provides a number of userspace tools.

Access Control Model. In an SELinux enabled system, each object (e.g., files, IPC channels, etc.) and subject (i.e., processes) is labeled with a *security context*, which consists of the attributes triplet (*user, role, type*). These attributes determine to which objects a subject has access based on *Type Enforcement* and *Role-Based Access Control* access control mechanisms.

Type Enforcement is the primary mechanism for access control in SELinux and is based on the context's *type* attribute. By default, all access is denied and must be explicitly granted through policy rules—*allow rules* in SELinux terminology. Using the notation introduced in [35], 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 subject types and object types are the types set in the security context of the subject and object, respectively. 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. For instance, the rule

$$allow\ useradd\_t\ passwd\_t\ :\ file\ write$$

defines that a process (subject) with type *useradd_t* is allowed to *write* an object with class *file* and type *passwd_t*. This rule is important on multi-user desktop systems, where the `/etc/passwd` file contains essential user information and thus should be protected. The `useradd` tool adds a new user to the system by adding the new user's information to the `passwd` file and thus requires write access. A typical SELinux policy on Fedora Linux 17 currently defines more than 600 types, almost 100 classes, and more than 100,000 allow rules. We evaluate policy complexities of different SELinux versions and of FlaskDroid in more detail in Section 7.1.

The *user* and *role* attribute form the basis for SELinux *Role-Bases Access Control*, which builds upon type enforcement by defining which type and role combinations are valid for each user in the policy.

Optionally, SELinux supports *Multilevel Security* (MLS), which extends the fundamental type enforcement. When MLS is enabled, the security context is extended with *low security level* and *high security level* attributes, where the low level represents the current security level of a subject/object and the high level represents the clearance level of the subject/object. Each security level is described by a *sensitivity* and a set of *categories*. Sensitivities are strictly hierarchical and reflect an ordered data sensitivity model (e.g., TopSe-

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

cret, Secret, Unclassified) [52]. Categories are unordered and reflect data compartmentalization (e.g., Research, Human Resources, Contracts). With MLS enabled, access to objects is only allowed of the subject holds a high enough security clearance (sensitivity) and the correct category for the object.

Dynamic policies. SELinux supports to some extent dynamic policies based on *boolean flags* which affect *conditional policy* decisions at runtime. Nevertheless, 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 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, but at most logged). Other booleans can, for instance, refine the coverage of the access control, e.g., by enabling/disabling access control on certain object classes like files or sockets.

Technically, this mechanism is implemented in the form of *if* statements for *allow* rules in the policy. Thus, only when the *if* condition evaluates to *True*, the rules in the block of the *if* statement are considered during access control decisions.

Userspace Object Managers. A powerful feature of SELinux is that its access control architecture can be extended to security-relevant userspace daemons and services, which manage data (objects) independently from the kernel [81]. 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. Prominent examples for such USOMs include the *X Window System server* [82] (Linux' display manager), *GConf* [16] (the GNOME settings manager), *SE-PostgreSQL* [45] (a security-enhanced object-relational database system), or *D-BUS* (a message bus system for inter-process communication).

Alternatively to querying the kernelspace security server, USOMs could query a userspace security server for access control decisions.[4] However, this approach is not any longer pursuit by the SELinux developers.

2.4 SE Android.

SE Android [74, 73, 71] prototypes SELinux for Android's Linux kernel. It aims to demonstrate the value of SELinux in defending against various root exploits and application vulnerabilities on the Android platform [74, 73, 71]. 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. Although SE Android is a prototype of SELinux for Android, there are a few key security extensions tailored for Android that SE Android brings about. First, SE Android introduces new hooks for Android's Binder driver making the latter a Kernelspace Object Manager. This ensures that all Binder IPC is subject to SE Android policy enforcement. Second, it labels application processes with SELinux-specific security contexts which are later used in type enforcement. In contrast to traditional desktop platforms, where new application processes are spanned when executing a binary, on Android new app processes are forked from a system process, denoted *Zygote*, which is pre-initialized with all important shared libraries and thus enables fast starts of new app processes. Thus, security labeling had to be integrated into this mechanism in order to label the newly created app processes accordingly. Moreover, the SE Android developers had to start writing an SE Android specific policy from scratch. This new policy contains at the time of writing more than 200 types, about 80 object classes, and roughly 1,400 allow rules, which is magnitudes smaller than previous SELinux policies (see also Section 7.1). Thirdly, 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 applications at install-time. Based on criteria, such as the permissions the app requests or its developer signature, apps are assigned a security type. This mapping from application meta-information to security types is defined in the SE Android policy.

Middleware MAC. While the above listed security mechanisms are directly derived from SELinux and address the lower level of the Android software stack (e.g., files, sockets, and IPC), SE Android additionally provides *simple* support for MAC policy enforcement at the middleware layer[5] (MMAC) inspired by various related work [72]. In particular, MMAC consists of three distinct mechanisms: 1) Install-time MAC, which, similar to Kirin [25], performs a policy-driven install-time check of new applications and denies installation when the application requests a defined combination of permissions; 2) Permission revocation, which is realized similar to existing implementations found in custom roms[6], commercial products[7], or related work [90, 12, 11]. This mechanism overrules the default Android permission check with a policy-based decision to allow/deny an application to leverage a granted permission; 3) Intent MAC, which protects with a white-listing enforcement the delivery of Intents to Activities, Broadcast Receivers, and Services. Similar mechanisms are employed, for instance, in [90, 12, 11]. However, in SE Android, the white-listing rules are based on the security type of the sender and receiver of the Intent message as well as Intent data such as the Action string.

3. REQUIREMENT ANALYSIS FOR ANDROID SECURITY ARCHITECTURES

3.1 Adversary Model

We consider a strong adversary with the goal to get access to sensitive assets. Typical examples of attacks are stealing confidential data, violating the user's privacy, as well as compromising system or third-party applications. That means we consider a strong adversary model that is able to launch *software* attacks targeting different layers of the Android software stack.

Middleware Layer.

Recently, different attacks operating at Android's middleware layer have been reported. Prominent examples are:

[4]`http://oss.tresys.com/projects/policy-server`

[5]See also `http://selinuxproject.org/page/SEAndroid#Middleware_MAC`

[6]CyanogenMod - `http://www.cyanogenmod.com`)

[7]3LM - `http://www.3lm.com`

Overprivileged 3rd party apps and libraries These attacks consist of privacy violations that range from questionable privacy practices of 3rd party app developers to spyware-like behavior. For instance, popular apps like WhatsApp [7, 6] Path [5, 28], or Facebook [3] have been publicly debated to overstep the necessary boundaries of their access to user's private data, e.g., by uploading the entire contacts database of the ContactsProvider instead of only the subset of contacts information required for correct app functionality.

Moreover, advertisement libraries, frequently included in 3rd party apps on Android, have been shown to exploit the permissions of their host app to collect information about the user [34], including privacy sensitive data such as contacts or location.

Malicious 3rd party apps In the recent past the number of mobile malware has steadily increased [79, 27]. The predominant observed malicious behavior consists of leveraging dangerous permissions to cause financial harm to the user (e.g., sending premium SMS). Additionally, as supported by academic studies, the exfiltration of user-private information is in fact also a prevalent characteristic of mobile malware [89, 62] such as the Geinimi Trojan [49].

Confused deputies Confused deputy attacks concern malicious applications, which leverage unprotected interfaces of benign applications (denoted deputies) to indirectly access protected functionality or data and thus escalate their privileges at runtime. Recent research has identified confused deputies in system apps such as Phone [24] or Settings [63] (e.g., to trigger phone calls), as well as in 3rd party applications [20, 88] (e.g., to retrieve contacts information).

Collusion attacks Collusion attacks concerns malicious applications that collude in order to merge their permission sets and gain a permission set which has not been approved by the user. A prominent example for a collusion attack is Soundcomber [36], where one application has the permission to record audio and monitor the call activity, while a second one owns the Internet permission. When both applications collude, they can capture the credit card number (spoken by the user during a call) and leak it to remote adversary. Collusion attacks can be further subclassified according to the channel over which they communicate [11, 51], e.g., overt channels like sockets versus covert channel like file locks or volume level.

Sensory malware Sensory malware leverages the information from onboard sensors in order derive user's privacy sensitive information. For instance, the accelerometer provides information about the movement of the phone, which can be used to infer the user input to the virtual keyboard, e.g., passwords [85, 14].

Root Exploits.

Besides attacks at Android's middleware layer, various privilege escalation attacks on lower layers of the Android software stack have been reported [87, 9] which grant the attacker root (i.e., administrative) privileges on the system. Leveraging these privileges, an attacker can bypass the Android permission framework. For instance, instead of querying contacts information from the ContactsProvider, he can directly access the contacts database on the file-system. Moreover, processes on Android executing with root privileges automatically inherit all available permissions at middleware layer and are hence omnipotent at both layers.

It should be noted that attacks targeting vulnerabilities of the Linux kernel are out of scope of this paper. This is motivated by the fact that 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 now derive the necessary requirements for an efficient and flexible access control architecture for mobile devices. Essentially, these requirements are valid for various mobile operating systems. In this paper we focus on the popular and open-source Android OS.

Access Control on Multiple Layers..

To mitigate the attacks at middleware layer, a large body of literature has been established that aims at extending Android's middleware with attack-specific access control [23, 40, 19, 84, 8, 44, 59, 12, 11].

Kernel-level MAC. However, any security extension to the middleware can be circumvented by privilege escalation attacks at the lower level, e.g., by using root exploits, as explained in our Adversary Model (Section 3.1). Mandatory access control solutions at kernel level, such as SE Android [55] or Tomoyo [37], help to defend against or to constrain these low-level privilege escalation attacks [74, 73, 71].

Middleware MAC. 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 [74, 72]. For instance, by design the inter-process communication (IPC) between two applications A and B is very often interposed by a middleware system component C that mediates this communication. A low-level MAC, e.g., on Binder IPC, lacks the required semantics to handle access control on such indirect communication channels between the two applications: It could prohibit the IPC $A \rightarrow C$ or $C \rightarrow B$, but its semantics do not allow to recognize the logical communication channel $A \rightarrow B$. General prevention of the communication between A and C or B and C would not make sense, because the correct and stable functionality of applications like A and B depends on the capability to communicate with system components like C. Moreover, prohibiting Binder IPC at kernel level easily causes unexpected security exceptions that cause application crashes. Furthermore, the extensive permissions a process executing with root UID holds at Android's middleware layer effectively circumvent the default permission-based security mechanism. A middleware MAC is able to address these problems of kernel level MAC, since it is based on middleware semantics.

Thus, a first challenge 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 and no (low-level) privilege escalation attack, such as a root exploit, bypasses these middleware access control.

Multiple stakeholders policies..

Mobile systems involve different parties 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, such as *Saint* [59], *TrustDroid* [12], or *TISSA* [90], propose special purpose solutions to address the security requirements and specific problems of app developers, 3^{rd} parties (here companies), or the end-user, respectively. 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 of multiple stakeholders is to support (basic) policy reconciliation mechanisms for dynamic policies from these stakeholders [65, 53]. For instance, [65] discusses different strategies for reconciliation, such as *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).

Context-awareness..

The security requirements of different stakeholders may depend on the context the device is currently used in. For instance, the security policy of an enterprise might dictate that certain assets, such as apps, may only be accessed during working hours or while the device is located on enterprise premises. 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 privacy-preserving use cases (as shown in Section 6), such as a privacy-preserving phone-booth mode, the selective and fine-grained protection of app interfaces [59], or multiple isolated security domains, e.g., dual-persona smartphones that isolate enterprise and private interests from each other [12].

Advantage of mobile OSes for policy complexity..

At first glance it may seem very challenging to realize a fine-grained and flexible access control on both the middleware and Kernel-level. In particular, SELinux [50] and similar Mandatory Access Control solutions on desktop and server systems are notorious for their extremely complex policies (cf. Section 7.1) and a comparable complexity could be expected for our solution. However, the design of mobile operating system differs in one important point from the design of traditional platforms: Security and privacy critical functionality is concentrated in a small number of privileged system components, which expose these functionality through clearly defined interfaces to applications. This is a distinct advantage for reducing the policy complexity in FlaskDroid: 1) The privileged system components form a single point of access to their functionality and instrumenting them as MAC enforcement point achieves inherently a high degree of coverage; 2) The extend of the policy primitives (e.g., object classes and operations) and rules is significantly reduced, since only a small number of system components has to be instrumented as MAC enforcement points.

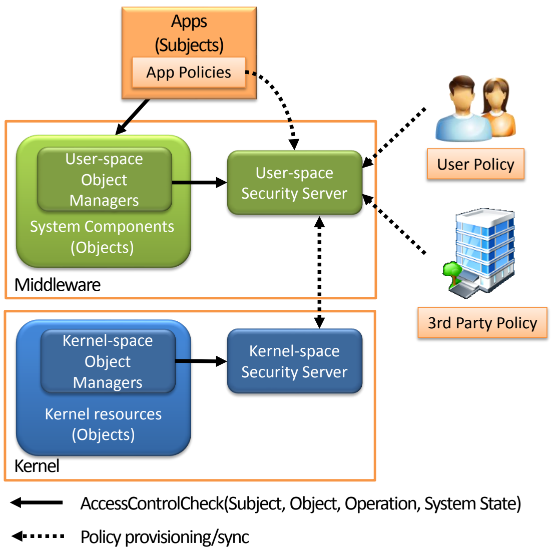
Apps
(Subjects)
App Policies
User Policy
User-space Object Managers
System Components (Objects)
User-space Security Server
Middleware
3rd Party Policy
Kernel-space Object Managers
Kernel resources (Objects)
Kernel-space Security Server
Kernel
AccessControlCheck(Subject, Object, Operation, System State)
Policy provisioning/sync

Figure 1: *FlaskDroid concept*

4. FlaskDroid ARCHITECTURE

In this section, we explain the high-level concept of FlaskDroid and its architecture. We present the policy language employed in our system, provide an overview of our FlaskDroid architecture, and elaborate in more detail on particular design decisions made therein.

4.1 Overview

In this paper, we introduce our *FlaskDroid* architecture, our enhancement to the Android operating system that is inspired by the Flask security architecture [78]. The high-level idea is depicted in Figure 1: various Object Managers at middleware and kernel-level are responsible for assigning their objects security contexts. Objects can be, for instance, kernel resources such as files or IPC and middleware resources such as Service interfaces, Intents, or ContentProvider data. As shown in Figure 1, 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 (AccessControlCheck). Thus, our approach follows the concept of a userspace policy manager (cf. Section 2.3). Moreover, to enable more dynamic policies, the policy check in FlaskDroid depends not only on the Subject, Object, and Operation performed, but also the System State, which determines the actual security context of the objects and subjects at runtime. For instance, while the dialer app is allowed to access the user's contacts data in one state, it may not be allowed to do so in another state (cf. Section 6.6).

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 at runtime, 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 Access Control

Access control can be abstractly described as a function

$$\mathcal{AC}(p_1, p_2, \ldots, p_n) \mapsto 0/1$$

which maps a number of input parameters to a DENY (0) or ALLOW (1) decision. The access control policy consists of a set of access control rules $\mathcal{R}$ and a default decision $\bot \in \{0, 1\}$. Each access control rule $r \in \mathcal{R}$ is defined as a tuple

$$r \approx (p_1, p_2, \ldots, p_n)$$

where $p_1, p_2, \ldots, p_n$ denote the parameters of the rule. An access control decision is then defined as

$$\exists r \in \mathcal{R} : r \approx (p_1, p_2, \ldots, p_n) \Rightarrow \mathcal{AC}(p_1, p_2, \ldots, p_n) \mapsto \overline{\bot}$$

$$\nexists\, r \in \mathcal{R} : r \approx (p_1, p_2, \ldots, p_n) \Rightarrow \mathcal{AC}(p_1, p_2, \ldots, p_n) \mapsto \bot$$

This means that if there exists a rule $r \in \mathcal{R}$ which matches the input parameters $p_1, p_2, \ldots, p_n$, $\mathcal{AC}$ returns the inverse of the default decision, or if no such rule exists, it returns the default decision. An access control system implements *whitelisting*, if $\bot = 0$, or *blacklisting*, if $\bot = 1$.

The access control rules for type enforcement in SELinux (cf. Section 2.3) consists of a quadruplet (p_1, p_2, p_3, p_4) where, according to the notion in Section 2.3

$$\begin{aligned} p_1 &= T_{Subj} \\ p_2 &= T_{Obj} \\ p_3 &= C_{Obj} \\ p_4 &= O_C \end{aligned}$$

and we will explain this in more detail in Section 4.3 when we introduce the access control policy in FlaskDroid.

However, at this point we shortly illustrate the above described access control at the example of the default Android permission check. This permission check implements a *simple* whitelisting access control $\mathcal{AC}(p_1, p_2)$, where parameter $p_1 = \mathsf{UID}_{\mathsf{Caller}}$ denotes the UID of the calling app represented as an Integer and parameter $p_2 = \mathsf{Permission}$ denotes the permission the calling app must hold represented as a String. Each rule $r \in \mathcal{R}^{Perm}$ is a tuple of the form $(\mathsf{UID}, \mathsf{Permission})$, defining that a UID holds a permission Permission. To illustrate this more concretely, consider an application with $\mathsf{UID} = 10010$ which only holds the CONTACTS permission, i.e., there exists a rule $r \in \mathcal{R}^{Perm}$ with $r = (10010, \mathsf{CONTACTS})$. Thus, when this app accesses the ContactsProvider to query contacts information, the permission check

$$\mathcal{AC}(10010, \mathsf{CONTACTS}) \mapsto 1$$

would allow this access. However, if the same application would try to open an Internet socket, the corresponding permission check

$$\mathcal{AC}(10010, \mathsf{INTERNET}) \mapsto 0$$

would deny this operation since no rule $r = (10010, \mathsf{INTERNET})$ exists for this UID.

Similarly, different related works which extend Android's security architecture at the middleware layer [58, 57, 19, 59] or kernel-level [55, 12, 11] can be expressed as MAC-enhancements to the Android OS and thus be represented by policy rules of the form $(p_1, p_2, \ldots, p_n)$. We will show this in more detail in Sections 6 and 8).

With FlaskDroid we aim at a security architecture that provides the enforcement of access control rules that are generic enough to allow for the instantiation of different security models.

4.3 Policy

In this section we explain the policy language used in FlaskDroid to epxress the access control rules.

4.3.1 Policy Language and Extensions

In FlaskDroid, we employ SELinux's policy semantics for expressing policies on both middleware and kernel-level. A detailed description of the SELinux policy language and all its features is out of scope of this paper—the reader is referred to [50] for a more elaborate explanation and to Section 2.3 for a summary—and here we outline only the subset of features important for our architecture.

The fundamental building block for SELinux' policies and thus for our middleware MAC is type enforcement. Each subject and object in the system has been assigned a *type*. Objects are further distinguished by their *class*, e.g., file or socket, and *operations* the object class supports, e.g., read, write, open. Thus, each policy rule $r \in \mathcal{R}$ consists of the quadruplet $p_1 = subject\_type$, $p_2 = object\_type$, $p_3 = object\_class$, and $p_4 = operation$, which express that a particular subject type is allowed/denied to perform a certain operation on an object of a particular class and type. Listing 17 in Appendix A shows the definition of some types as well as their grouping in *attributes* – synonyms representing a set of types within policy definitions. To allow for more dynamic policies, SELinux introduces so called *booleans* which allow conditional policy rules, i.e., the validity of a rule r can depend on the current value of an assigned boolean b_r. Thus, also the result of the access control check $\mathcal{AC}$ depends on the current state of all defined booleans. For readability, we refer to the conditional boolean of a rule r as parameter p_5 in r. Similarly, we introduce a new parameter p_5 for $\mathcal{AC}$ which represents the current state of all booleans present in the access control system and hence might affect the access control decision.

To make use of the middleware's rich semantics and the available contextual information – both of which allow for more powerful policy rules – we extend the middleware policy language with new default classes, constructs and transition definitions which we will present in the subsequent sections.

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. The most basic classes are directly derived from the fundamental building blocks of Android applications and communication channels available, namely Activity, Service, ContentProvider, Broadcast, and Intent. Operations for these classes are derived in the same manner, for example, *startActivity*, *query* a ContentProvider, or *receive* a Broadcast. Classes can also be derived from another class, thus inheriting all operations of the parent class while new operations for the child class can be optionally added. Listing 18 in Appendix B presents examples for the definition of basic classes such as *activity_c* or *service_c* and illustrates inheritance at the example of *intentService_c* which inherits from *service_c* as well as several ContentProviders like *contactsProvider_c* which inherit from the *contentProvider_c* class.

Application Types.

A further extension is the possibility to define criteria by which applications are labeled with a security type. This is motivated by the fact that at the time a policy is written, one cannot predict which apps will be installed in the future. Thus, metrics are required to label apps upon installation. This is consistent with the extensions of SE Android (cf. Section 2.4), where a similar mechanism is used to label newly spawned application processes based on pre-defined criteria.

Listing 1 illustrates an example policy snippet defining the criteria for assigning a type to applications (i.e., labeling apps). The criteria at middleware level can be, for instance, the application package name, the requested permissions, the developer signature, or potential, non-standard meta-information such as an external signature (lines 10-50). To illustrate this more concretely, consider a newly installed application whose package name equals *com.android.apps.tag*. According to the lines 30-33 in the policy in Listing 1, this app would be assigned the type *app_tag_t*, since this the criteria for type *app_tag_t* match the new app. If no criteria matched a specific app, a default type is assigned (line 4). In FlaskDroid, labels are assigned to the sandbox of applications, i.e., to their UID or in case of shared UIDs to their shared UID. The latter decision is motivated by the fact that UID is the smallest identifiable unit provided by Binder.

Listing 1: *Example policy snippet illustrating the definition of criteria for assigning an app a specific type.*

```
 1 /*
 2 Default type
 3 */
 4 defaultAppType untrustedApp_t;
 5
 6 /*
 7 Define criteria to assign types to apps
 8 */
 9
10 appType app_cased_t
11 {
12     Developer:signature=0xFEF9...;
13     Package:package_name=de.cased.trust.app;
14     ExternalSignature:keyFileLocation=/etc/key.file;
15     ExternalSignature:signatureFileLocation=assets/sig.file;
16 };
17
18 appType android_t
19 {
20   /* All of packages under this UID */
21   Package:package_name=android;
22   Package:package_name=com.android.keychain;
23   Package:package_name=com.android.settings;
24   Package:package_name=com.android.seandroid_manager;
25   Package:package_name=com.android.providers.settings;
26   Package:package_name=com.android.systemui;
27   Package:package_name=com.android.vpndialogs;
28 };
29
30 appType app_tag_t
31 {
32   Package:package_name=com.android.apps.tag;
33 };
34
35 appType app_backupconfirm_t
36 {
37   Package:package_name=com.android.backupconfirm;
38 };
39
40 appType app_telephony_t
41 {
42   Package:package_name=com.android.phone;
43   Package:package_name=com.android.providers.telephony;
44 };
45
46 appType app_bluetooth_t
47 {
48   Package:package_name=com.android.bluetooth;
49 };
50
51 [...]
```

Intent types.

Similarly to apps, also Intents may be the object of access control enforcement, e.g., if a particular app is allowed to send or receive an Intent of a particular type. Thus, also for Intents we require criteria to label Intent objects. Listing 2 illustrates the definition of such criteria for assigning a type to Intent objects. It leverages the information available about Intents such as their action string or category, or receiving component (lines 7-11). Again, if no criteria matched an Intent, a default type is used (line 4). For instance, the example in Listing 2 would assign an Intent with action string *android.intent.action.MAIN*, category *android.intent.category.HOME*, and type *app_launcher_t* of the receiving component the type *intentLaunchHome_t*. In contrast to apps, which are labeled once during installation, Intents are labeled on-demand at runtime during policy checks which involve Intent objects and, naturally, the assigned type is bound to the life-time of the corresponding Intent object.

Listing 2: *Policy snippet showing definition of criteria for assigning an Intent a specific type.*

```
 1 /*
 2 Default type
 3 */
 4 defaultIntentType untrustedIntent_t;
 5
 6
 7 intentType intentLaunchHome_t
 8 {
 9     Action:action_string=android.intent.action.MAIN;
10     Categories:category=android.intent.category.HOME;
11     Components:receiver_type=app_launcher_t;
12 };
```

Context definitions and awareness.

We extend the policy language with an option to define device *contexts*. 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). This extension is aimed towards enabling context-aware policies. For instance, one can dynamically revoke the permission to query the contacts database. Listing 3 shows the declaration of three contexts, `work_con`, `phoneBooth_con`, and `iptablesExecForbidden_con`. Each declared context can be either *active* or *inactive* and we explain in the subsequent Section 4.4 how contexts are activated/deactivated by dedicated Context Provider components.

Listing 3: *Policy snippet showing declaration of contexts.*

```
context work_con;
context phoneBooth_con;
context iptablesExecForbidden_con;
[...]
```

To actually make use of context-aware policies, access control rules defined in our policy language may depend on optional boolean parameters, which in turn depend on the currently active contexts. This booleans exist only at the middleware and affect only the policy decision making in the user-space security server for the middleware. To similarly map contexts to the kernel-level, we introduce special boolean definitions which point to a boolean at kernel level instead of adding a new boolean at middleware. Changes to such kernel-mapped boolean values triggers a call to the SELinux kernel module to update the corresponding SELinux boolean (cf. Section 4.4.1). On a context switch reported by the system, all boolean variables that relate to the new context are set to their respective values. This enables or disables the policy rules which depend on these boolean values. When a context is unset, all related booleans can be optionally auto-reverted to their original value or, alternatively, one can define other contexts which trigger a change of those booleans.

Listing 4 presents the definition of a boolean *phoneBooth_b* at middleware level (line 4) and references to a boolean *allowIPTablesExec_b* defined in the underlying SE Android policy (line 9). Both, contexts (cf. Listing 3) and booleans, are used in switchBoolean statements (lines 15-26), which define which booleans are (un)set depending on which context is (in-)active. To illustrate this, consider the switchBoolean statement in lines 15-20 in Listing 4, which defines that as soon as the context *phoneBooth_con* is active, the middleware 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 original value, i.e., *False* (line 4). The switchBoolean in lines 22-27 works identical, however, for the context *iptablesExecForbidden* and the boolean *allowIPTablesExec_b* which is an SE Android boolean and hence the change in value is mapped to the kernel-level.

Listing 4: *Policy snippet showing how booleans are linked with contexts.*

```
 1  /*
 2  Middleware boolean definitions
 3  */
 4  bool phoneBooth_b = false;
 5
 6  /*
 7  Kernel boolean defintion used for sync with SE Android
 8  */
 9  kbool allowIPTablesExec_b = true;
10
11  /*
12  Dynamic policies
13  */
14
15  switchBoolean
16  {
17      context=phoneBooth_con;
18      auto_reverse=true;
19      phoneBooth_b=true;
20  };
21
22  switchBoolean
23  {
24      context=iptablesExecForbidden_con;
25      autoReverse=true;
26      allowIPTablesExec_b=false;
27  }
28
29  [...]
```

Policy rules.

Listing 5 shows the definition of some example access control rules. For defining access control rules as described in Section 4.2, we leverage the default SELinux *allow rule* syntax. Thus, each rule is of the form

$$r = (p_1, p_2, p_3, p_4)$$

where

$$\begin{aligned} p_1 &= subject\_type \\ p_2 &= object\_type \\ p_3 &= object\_class \\ p_4 &= operation \end{aligned}$$

To ease writing policies, all parameters can be *sets* of logically disjunct parameters, i.e., the parameter matches if any of the set members matches. For instance, the rule in line 5 of Listing Listing 5 has as *subject_type* parameter the set of types *{app_system_t app_contacts_t app_launcher_t}*, as *object_type* the type *allContactsData_t*, as *object_class* parameter the class *contactsProvider_c*, and as *operation* the function *query*. Optionally, these rules can also depend on boolean parameters which enable or disable policy rules. This dependency is noted in form of *if* statements in the rules. For instance, the two rules in lines 17 and 18 in Listing 5 depend on the boolean *phoneBooth_b* and are only valid if *phoneBooth_b* is *True* or are invalid if is *False*, respectively.

To further ease writing access control policies, a rule can be defined as

$$r = (\mathsf{self}, p_3, p_4)$$

where

$$\begin{aligned} p_3 &= object\_class \\ p_4 &= operation \end{aligned}$$

if $p_1 = p_2$, meaning that this rule applies always when subject type and object type are identical. Lines 1-3 of Listing 5 show examples of such rules. For instance, the rule in line 3 states, that any app can send and receive broadcasts to and from apps with the identical type.

Additionally, we add the keyword *any*, which can be used as a wildcard for the subject type, object type, object class, and operations and matches any argument. For example, the rule

$$r = (any, app\_untrustedApp\_t, \mathsf{contentProvider\_c}, any)$$

would allow any operation by any subject type on a ContentProvider component of an app with type *app_untrustedApp_t*.

Listing 5: *Policy snippet showing definition of access control rules (optionally depending on boolean parameters).*

```
1  self: app_c {checkPermission};
2  self: activity_c {finish moveTask};
3  self: broadcast_c {receive send};
4
```

```
allow {app_system_t app_contacts_t app_launcher_t}
      allContactsData_t: contactsProvider_c {query};

allow {app_system_t app_telephony_t app_contacts_t
      app_launcher_t} {app_system_t app_telephony_t
      app_contacts_t app_launcher_t}: package_c
      {getPackageInfo getPackageInfoWithUninstalled
      getPackageUID getPackageGIDs getPackagesForUid
      getNameForUid getUidForSharedUser
      findPreferredActivity queryIntentActivities
      getInstalledApplications
      getInstalledApplicationsWithUninstalled
      getInstalledPackages
      getInstalledPackagesWithUninstalled};

allow {app_system_t app_telephony_t app_contacts_t
      app_launcher_t} {app_system_t app_telephony_t
      app_contacts_t app_launcher_t}: app_c
      {checkPermission};

allow {app_system_t app_telephony_t app_contacts_t
      app_launcher_t} {app_telephony_t app_contacts_t}:
      activity_c {start};

allow {app_system_t app_telephony_t app_contacts_t
      app_launcher_t} {app_system_t app_telephony_t
      app_contacts_t app_launcher_t}: activity_c {moveTask
      finish};

if(~phoneBooth_b)
{
    allow {app_system_t app_telephony_t app_contacts_t
          app_launcher_t} {app_system_t app_telephony_t
          app_contacts_t app_launcher_t}: activity_c {start
          moveTask finish};
    allow app_telephony_t allContactsData_t:
          contactsProvider_c {query};
};

[...]
```

4.3.2 Dynamic Policies and Multiple Stakeholders

A particular challenge for the design of FlaskDroid is the support for protecting the interests of different stakeholders on the mobile device. This requires, that policy decisions consider the policies of all involved stakeholders. These policies can be, for instance, pre-installed (i.e., system policy or enterprise policy), delivered with application packages (i.e., app developer policies), or configured manually via a GUI (i.e., user policies). Thus, in FlaskDroid 3rd party developers have the choice to deploy custom policies for their applications. Note, that this is completely voluntary on their part and they may choose to opt in and rely on our security framework to enforce their policies. In case they opt out from using our framework (or they are unaware of it), their applications are still subject to the system and user policy enforcement.

Moreover, developers who opt in can instrument their app as a userspace object manager for their own data objects and FlaskDroid supports them by providing the necessary interfaces to the system security server as part of the SDK. For instance, a 3rd party application could contain a ContentProvider component to manage sensitive data, e.g., user account information like email addresses, full names, or telephone number. Further, the developer of this application wants to provide access to this data to other 3rd party applications which have been signed by another developer that is in a set of endorsed developers. Additionally, this access should be fine-grained on a per data basis, i.e., some endorsed developers should be able to retrieve only email addresses from the account information, while others should be able to only retrieve the phone number. To implement this system with FlaskDroid, the 3rd party developer would deploy a policy in which he defines criteria to label other 3rd party applications on the system with a type according to their signature (i.e., endorsed for email address, endorsed for phone numbers, not endorsed). To enforce this policy, he would instrument the ContentProvider component such that it queries the Userspace Security Server via the SDK-provided function when another application access the ContentProvider. Based on the access decision, the ContentProvider returns only the data for which the calling app has been endorsed. A concrete design for such enforcement is provided, e.g., in our technical report [13].

A particular challenge when supporting multiple stakeholders is the consolidation of the various stakeholders' policies. Different strategies for reconciliation are possible [65, 53] and supported by our architecture, based on namespaces and global/local type definitions. For instance, as discussed in [65], *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 which the access control policy is developed. 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.

4.4 Architecture Components

Figure 2 provides an overview of our architecture. In the following paragraphs we will explain the individual components that comprise the FlaskDroid architecture.

4.4.1 SE Android Module

At the kernel-level, we employ the SE Android module provided in the SE Android revision of the Android sources [55]. As explained in Sections 2.3 and 2.4, SELinux/SE Android provides a powerful basis for enforcing a fine-grained policy on kernel-level objects and classes, such as files or IPC.

In our design, we leverage SE Android primarily for the following purposes (cf. Figure 3): First, it is an essential building block for hardening the Linux kernel [74, 71] thereby preventing malicious applications from (easily) escalating their privileges by exploiting vulnerabilities in privileged (system) services/daemons. Even when an attack, usually with the intent of gaining *root* user privileges, is successful, SE Android can constrain the privileges of the application by restricting the privileges of the root account itself. Second, it complements the policy enforcement at the middleware level. Kernel-level MAC prevents applications from bypassing the middleware enforcement points (in Flask terminology defined as Userspace Object Managers (USOMs)) by enforcing that any privileged operation must go through the Android software stack in a top-down fashion and hence pass all policy enforcement points.

Figure 4 provides two examples for enforcing Android's security model with SE Android and preventing apps from bypassing middleware enforcement points: direct access to low-level system resources such as the radio daemon or the

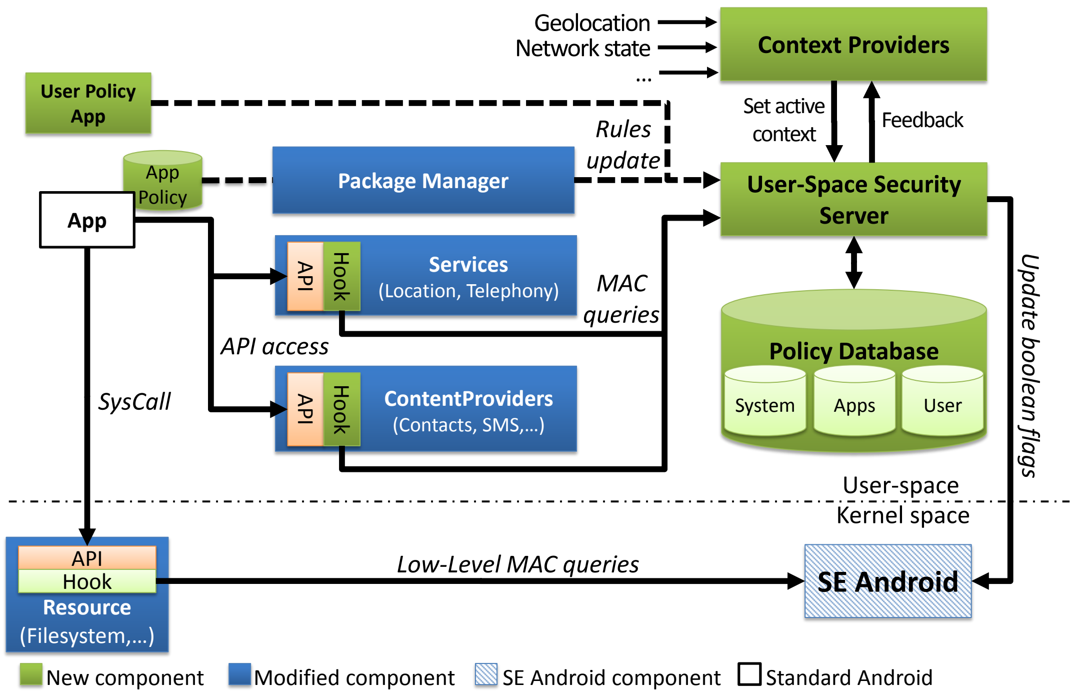

Figure 2: *FlaskDroid Architecture*

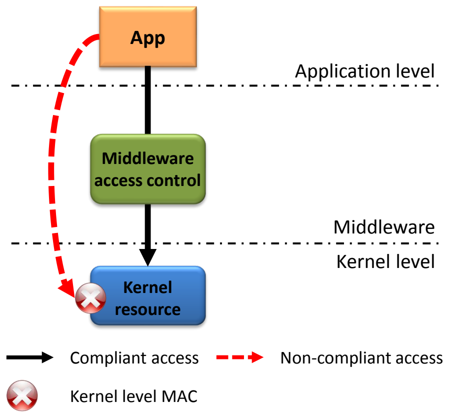

Figure 3: *SE Android as building block to 1) protect high-value low-level resources and 2) ensure middleware access control mechanisms cannot be bypassed.*

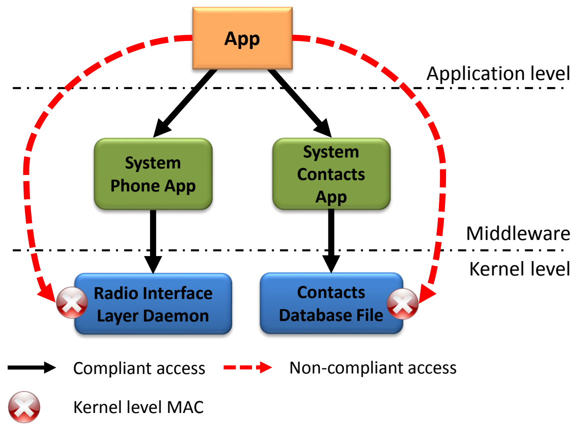

Figure 4: *Concrete examples for enforcing Android's security model with SE Android.*

contacts database file can be restricted to be only permissible if coming from the system phone app or system contacts management app. These apps in turn implement access control at middleware level on their public interfaces over which they expose radio or contacts management functionality to other apps.

Similarly, the SELinux concept of *domain-transitions*—whereby the type of a subject (i.e., process) changes to a new type[8]—can be leveraged to prevent applications from performing low-level operations: For example, a SELinux policy could dictate that the privilege to configure the kernel netfilter at runtime is limited to specific (system) apps. Using the dynamic policy support of SELinux (cf. boolean flags/conditional policies in Section 2.3) it is possible to reconfigure these access control rules at runtime depending on the current system state. For example, as explained in more detail in Section 6.5, a boolean flag could be used to allow 3^{rd} party firewall apps to access the kernel netfilter configuration when the device is not connected to the company network. When connected to the company network, access to the kernel netfilter configuration is restricted so that sensitive network traffic cannot be redirected easily.

[8]It is worth noting that *process transition* is a capability that a subject needs to have in order to perform a domain transition. Thus, domain transitions are forbidden unless explicitly allowed by the security policy.

USOM	Example operation
Service USOMs	
PackageManagerService	getPackageInfo getPackageUID findPreferredActivity getInstalledApplications installPackage uninstallPackage
ActivityManagerService	startActivity moveTask grantURIPermission sendBroadcast receiveBroadcast registerBroadcastReceiver
AudioService	adjustStreamVolume getSreamVolume setStreamVolume setRingerMode setVibrateSetting
PowerManagerService	acquireWakeLock isScreenOn reboot preventScreenOn
SensorManager	getSensorList getDefaultSensor unregisterListener registerListener
LocationManagerService	getAllProviders requestLocationUpdates addGpsStatusListener addProximityAlert getLastKnownLocation
ClipboardService	getPrimaryClip setPrimaryClip getPrimaryClipDescription addPrimaryClipChangedListener
SMSManager	copyMessageToIcc deleteMessageFromIcc disableCellBroadcast sendTextMessage
TelephonyManager	getCellLocation getDeviceId listen getNetworkType getCellLocation getMsisdn
AccountManagerService	getAccounts addAccount clearPassword getPassword grantAppPermission
ContentProvider USOMs	
ContactsProvider2	query insert update delete writeAccess readAccess bulkInsert
MMSSMSProvider	query insert update delete
TelephonyProvider	query insert update delete
SettingsProvider	query insert update delete bulkInsert
CalendarProvider2	query insert delete

Table 1: *List of system Userspace Object Managers in FlaskDroid with example operations controlled by each manager. Currently, the USOMs implemented in FlaskDroid comprise 136 policy enforcement points.*

Our middleware extension 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).

4.4.2 Userspace Object Managers

In our architecture, 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 mandatory 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 Userspace Object Managers (cf. Section 4.4.1).

Table 1 provides an overview of the *system* USOMs in FlaskDroid and shows some typical operations each USOM controls. Altogether, the USOMs implemented in FlaskDroid currently comprise 136 policy enforcement points.

In the following, we explain how we instrument the Services and ContentProviders as Userspace Object Managers. In particular, we highlight the roles of the PackageManagerService and ActivityManagerService in the design of FlaskDroid.

PackageManagerService.

The PackageManagerService is responsible for (un)installation of applications. It also maintains a global state of package meta information, such as a mapping between package names and their UID, install directories on the filesystem etc. More importantly, it maintains a global list of all available Activities, Services, Broadcast Receivers, and ContentProviders contained in the installed applications in order to find a preferred component for doing a task at runtime. For instance, if an email client sends an Intent to view a PDF document available as an attachment, the PackageManagerService looks for a preferred Activity able to perform the task; if there is no preferred Activity, it prompts the user with a selection of applications that can read a pdf file and waits for the user's choice.

As a Userspace Object Manager, we extend the functionality of the PackageManagerService to assign consolidated middleware- and kernel-level app types to pre-installed system applications at boot time and newly installed applications at install time using the criteria defined in the policy (cf. Section 4.3). More explicitly, we assign app types to the UID of the application, also considering shared UIDs between different apps. For instance, considering the *appType* definition in line 30 of Listing 1 in Section 4.3.1, the PackageManagerService would assign the security type *app_tag_t* to the UID under which the package *com.android.apps.tag* executes. Thus, whenever a policy check for access control $\mathcal{AC}(p_1, p_2, p_3, p_4)$ is performed for an operation of this app, then $p_1 = app\_tag\_t$ (if the app is subject) or $p_2 = app\_tag\_t$ (if the app is object) or both $p_1 = p_2 = app\_tag\_t$ if two components of this same app interact. Object-specific types and classes, on the other hand, are specific to the middleware or kernel layer and are used to define policy rules specific to the semantics of their respective layer. For instance, the object classes *contactsProvider_c* or *locationService_c* are middleware specific classes.

Similarly, we extend the logic for finding a preferred component for performing a task at runtime. This implicitly affects the interaction between the system and the user. By default, if several components are suitable to perform a given task (e.g., in the above example several PDF viewers are installed) but no component is set as preferred, the user is prompted to chose one of the potential receivers for performing the task. By filtering this receiver list based on policies, the user-prompt contains only options which would be allowed by the policy, while impossible choices are omitted. The filtering is based on the type of the application that triggers the task, the type of the Intent that describes the task, and the type of the potential receiving apps. Thus, let Recv_t to be the type of a potential receiver, Intent_t the type assigned to the Intent, and Sender_t the type of the app that triggers the task. Then, each receiving app is filtered for which either no allow rules exists that allows type Recv_t to receive an Intent of type Intent_t or that allows type Recv_t to communicate with type Sender_t.

Moreover, PackageManagerService is responsible for discovering developer provided policies in the application installation packages and forwarding them to the Userspace Security Server. The USSS parses them and considers them during access control decisions that involve the corresponding application (see Section 4.4.3).

ActivityManagerService.

The ActivityManagerService as part of the Android system server process is responsible for managing the stack of Activities of different applications, Activity life-cycle management, as well as providing the Intent broadcast system. As a USOM, the ActivityManagerService is responsible for labeling Activity, Intent and Broadcast 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 during operations that manipulate Activities, such as moving Activities to the foreground/background or destroying them. Listing 18 in Appendix B shows some examples of these object specific operations for these object classes.

In Section 4.3.1, we explained how Intents are labeled depending on available meta-information, such as the action and category string or the sending app process (UID). Broadcasts are simply Intents which are delivered to a list of registered receiver apps. To apply access control to the Broadcasts, the ActivityManagerService filters out all 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). Thus, for each receiver a policy check $\mathcal{AC}(p_1, p_2, p_3, p_4)$ is performed, where p_1 is the security type of the receiver, p_2 is the security type of the Intent, p_3 is the object class such as intent_c or *broadcast_c*, and p_4 is the operation such as receive. send, or receiveSticky (cf. Listing 18 in Appendix B). Moreover, the sender of the Broadcast can assign a custom label to his Intent in order to further classify or also to declassify this Intent if in accordance with the policies.

Content Providers.

As described in Section 2.1, ContentProviders 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 entries they manage during insertion/creation of data and for performing access control on update, query, or deletion of entries. The exact mechanism how they realize this can be based on different approaches: 1) more generic, but rather coarse-grained, based on the URI that identifies particular data entries within a ContentProvider; or 2) more fine-grained by integrating it into the storage back-end (e.g., SQLite Database), but losing generality. Currently, almost all ContentProviders are implemented as an SQLite database. Therefore, object labeling and access control can be achieved at the level of rows, columns or even cells within the database.

Services.

A Service is a component of an application which provides a particular functionality to other (possibly remote) components and exposes its interface as a Binder object that is generated based on an interface specification described in the Android Interface Definition Language (AIDL, cf. Section 2). To instantiate a Service component as a Userspace Object Manager, we opted for an semi-automated approach. In FlaskDroid, we provide an extension of the AIDL language and its corresponding code generator. Interfaces generated from an AIDL definition automatically contain a policy-based access control check on calls to the Service interface and each offered function. Additionally, the developer of the Service can add type-tags to the function definition in AIDL to further restrict access to this function and our code generator automatically adds this check to the produced interface code. Modifying the AIDL tool is inspired by the approach presented in [21]. 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 apps in the same way.

4.4.3 Userspace Security Server

To implement the policy decision point for userspace types and objects, two approaches are feasible: (1) Moving the policy decision point into the Linux kernel (e.g., the SE Android security server), which is queried by the USOMs directly, or (2) implementing a Userspace Security Server (USSS) which is responsible for all userspace policy decisions.

For FlaskDroid, we opted for the second approach by providing a clear separation of security issues between the userspace and the kernelspace components. Our design not only avoids overloading the kernel security server with access control rules which only make sense for middleware entities (subjects and objects) but, as described in Section 4.3.1, also enables the use of a more dynamic policy schema (different from the more static SELinux policy language). This enables us to take advantage of the rich semantics (e.g., contextual information) in the middleware layer.

Our Userspace Security Server exposes an interface to the USOMs which allows querying for access control decisions (cf. Figure 2). This is similar to the default permission check of Android, which is part of the SDK and can be called by any application through public interfaces. As explained in Section 4.3, our policy check is based on (1) the subject type (usually the type associated with the calling app UID), (2) the object type (e.g., *contacts_email* or the type associated with the called app UID), (3) the object class of the object (e.g., *contacts_data* or Intent), and (4) the operation on the object (e.g. *query*). This policy check implements the access control decision $\mathcal{AC}$ introduced in Section 4.2.

Regarding policies from multiple stakeholders (cf. Section 3.2), we take a conservative approach and require 3^{rd} party developers to define their custom policies including types, object classes, and operations. This policy is then included in the application package file and parsed by the PackageManagerService during the installation process. However, since we allow developers to define their own types, etc., we use namespaces to avoid name collisions between different policies. Thus, we allow parallel sets of policy rules, e.g., $\mathcal{R}^{System}$ for a pre-installed system policy, $\mathcal{R}^{10010}$ for the policies deployed by the application with UID = 10010 or $\mathcal{R}^{User}$ for policies defined by the end-user. Upon policy check, we identify the corresponding policies upon which the policy decision is based. For example, if $\mathcal{AC}(p_1, p_2, p_3, p_4)$ involves a system app and a third party app with UID = 10010, this check has to consider both $\mathcal{R}^{System}$ and $\mathcal{R}^{10010}$. Naturally, the different policies might conflict (e.g., $\mathcal{R}^{System}$ allows access and $\mathcal{R}^{10010}$ denies) and thus a reconciliation of the access control is required. As described in Section 4.3.2, our architecture allows us to define different consolidation strategies for the userspace policies, which the USSS then applies during access control decisions.

4.4.4 *Context Providers*

In Section 4.3.1, we presented our current mechanism for defining context-aware policies. By mapping contexts to booleans, policy rules are dynamically enabled/disabled at runtime in dependence on the contexts.

To allow for flexible control of contexts and their definitions, our design leverages Context Providers. This providers come in form of plugins to our Userspace Security Server (see Figure 2) and can be arbitrarily complex (e.g., uses machine learning) and leverage available information such as the network state or geolocation of the device to determine which declared contexts they activated/deactivate. Decoupling the context monitoring and definition from our policy provides that context definitions do not affect our policy language except for very simple declarations (see Listing 3). Thus, instead of defining the exact parameters for a context in the policy, e.g., longitude and latitude for geolocation, we only declare contexts and the Context Provider inform the USSS which contexts are active.

Moreover, the Userspace Security Server provides feedback to Context Providers, for instance, the executed access control decisions. Thus, this feedback also contributes to the current system context. A good example for the usefulness of such a feedback channel is an architecture like *XManDroid* [11], which aims at mitigating collusion attacks between different apps. In XManDroid, the set of allowed IPC channels of an app is directly dependent on its past IPC behavior. This approach can be instantiated with FlaskDroid with a plugin that models the currently established IPC channels (e.g., as a graph [11]) based on the granted access control decision. The plugin internally evaluates this graph and sets the contexts such that each application's IPC channels are restrained such that no collusion attack is feasible.

5. IMPLEMENTATION

We implemented a prototype of FlaskDroid based on SE Android in revision 4.0.4 [55] on a Samsung Galaxy Nexus phone. In the remainder of this section, we provide technical details on how we realized our prototype.

5.1 Policy Implementation

Listing 6 presents the full grammar of our policy language as implemented and used in the Listings throughout this paper. The grammar is noted in *Extended Backus-Naur Form*. We implemented a Python-based compiler for this language, using the *pyparsing* and *ElementTree* libraries, which produces an XML representation of the policy. For sanity checks, the compiler tool verifies the XML output against an XML Schema of the policy language as supported by our extensions on the device.

Listing 6: *Grammar of our policy language presented in Extended Backus Naur Form*

```
Policy ::= DefaultDecision, DefaultAppType, ↙
      DefaultIntentType, PolicyElements ;

(* Default definitions *)
DefaultDecision ::= “defaultDecision”, DecisionValue ;
DefaultAppType ::= “defaultAppType”, Identifier ;
DefaultIntentType ::= “defaultIntentType”, Identifier ;

(* Policy construct *)
PolicyElements ::= {Attribute}, {Type}, {Boolean}, ↙
      {KBoolean}, {Class}, {Rulestatement}, {Context}, ↙
      {SwitchBoolean}, {AppType}, {IntentType} ;

(* Policy element definitions *)
Attribute ::= “attribute”, Identifier, “;” ;
Type ::= “type”, #(Identifier), “;” ;
Boolean ::= “boolean”, Identifier, “=”, BooleanValue, “;” ;
KBoolean ::= “kboolean”, Identifier, “=”, BooleanValue, “;” ;
Class ::= “class”, Identifier, [ “inherits”, Identifier ], [ ↙
      BraceIdentifierlist ] “;” ;
Rulestatement ::= “if”, “(”, Identifier, “)”, “{”, 1*(Rule), ↙
      “}”, “;” | 1*(Rule) ;
Rule ::= “allow”, SubjectType, ObjectType, “:”, ↙
      ObjectClass, Permission “;”
      | “self”, “:”, ObjectClass, Permission “;” ;
Context ::= “context”, Identifier, “;” ;
SwitchBoolean ::= “switchBoolean”, sbBody, “;” ;
sbBody ::= “{”, contextAssignment, autoReverse, ↙
      1*(BoolAssignment), “}” ;
contextAssignment ::= “context”, “=”, Identifier, “;” ;
autoReverse ::= “auto_reverse”, “=”, BooleanValue, “;” ;
AppType ::= “appType”, Identifier, DomainAssignmentlist, ↙
      “;” ;
IntentType ::= “intentType”, Identifier, ↙
      DomainAssignmentlist, “;” ;

(* Basic constructs *)
KVAssignment ::= Identifier, “=”, Value, “;” ;
BoolAssignment ::= Identifier, “=”, BooleanValue, “;” ;
Assignmentlist ::= “{”, 1*(KVAssignment), “}” ;
DomainAssignmentlist ::= “{”, 1*(Identifier, “:”, ↙
      KVAssignment), “}” ;
Identifierlist ::= Identifier, *(whitespace, Identifier) ;
BracketIdentifierlist ::= “[”, Identifierlist, “]” ;
BraceIdentifierlist ::= “{”, Identifierlist, “}” ;
SubjectType ::= (Identifier | BraceIdentifierlist) ;
ObjectType ::= (Identifier | BraceIdentifierlist) ;
ObjectClass ::= (Identifier | BraceIdentifierlist) ;
Permission ::= BraceIdentifierlist ;

(* Basic definitions *)
DecisionValue ::= “deny” | “allow” ;
BooleanValue ::= “true” | “false” ;
Identifier ::= 1*(alphanum | “_”) ;
Value ::= [ “~” ], 1*(alphanum | “_” | “:” | “.” | “/”) ;

(* Basic terminals *)
alphanum ::= { alpha | digit } ;
alpha ::= “A” | ... | “Z” | “a” | ... | “z” ;
digit ::= “0” | ... | “9” ;
whitespace ::= ? white space characters ? ;
```

While the policy language is very close to the SELinux policy language and more suitable for human readers, the XML output is contains performance-oriented optimizations which decrease human readability. For instance, it unrolls attributes and type sets in allow rules, such that lookup of policy rules during policy checking performs more efficiently. For instance, Listing 7 presents in line 2 a compact example rule to allow the Exchange app and the eMail app to bind to services of the Exchange app and the eMail app. The compiler would unroll this rule and produce output containing four separate rules expressing the same security objectives, as shown in lines 4-7 of Listing 7.

Listing 7: *Example policy in compact form and set of rules expressing the same security objectives.*

```
1 /* Compact form for allow rule */
```

```
allow { app_exchange_t app_email_t } { app_exchange_t
      app_email_t }: service_c {bind};
/* Equivalent set of rules for the same security objectives */
allow app_exchange_t app_exchange_t: service_c {bind};
allow app_exchange_t app_email_t: service_c {bind};
allow app_email_t app_exchange_t: service_c {bind};
allow app_email_t app_email_t: service_c {bind};
```

5.2 Userspace Security Server

The Userspace Security Server acts as the central policy decision point for access control on objects in userspace and is implemented as part of the Android system server (*com.android.server*). It currently comprises 3741 lines of Java code. At system initialization, it loads and parses the XML-based policies and creates an in-memory representation. Access control rules are represented using a dedicated *AccessControlPolicy* class. To increase performance, we implemented a central Access Vector Cache (AVC) which stores policy decisions by mapping the tuple *(SubjectType, ObjectType, ObjectClass, Permission, Boolean)* to the corresponding policy decision using a Java HashMap data structure.

Interfaces.

Since the Userspace Object Managers (USOMs) may be scattered throughout the middleware- and application layers, the USSS provides an API for policy checks that is exposed to applications and other system components through the *Context* class. However, some middleware USOMs which account for the bulk of the policy checks, e.g., the ActivityManagerService and the PackageManagerService, are executed inside the system server as well, so no ICC is required to query the USSS.

Figure 5 presents the policy check mechanism and in particular the interface exposed to USOMs. The interface consists of a function call *checkPolicy*, which is triggered by USOMs when, e.g., an app accesses a managed object through an operation (step 1). The parameters to *checkPolicy* (step 2) are the UID of the calling app, the security type of the object, the class of the object, and the operation. While the object manager defines the object type, object class, and the operations it supports, it is oblivious of the security type of the subject. Instead, the USSS resolves the provided UID to its security type using the criteria defined in the *appType* statements the policy (cf. Section 4.3). Alternatively, if the access control check is triggered system centric (e.g., on Inter Component Communication between two apps), *checkPolicy* can be called with the UID of the object manager instead of the object type and the USSS will map both subject and object UID to their security types. Mapping the UIDs to security types within the USSS allows for a more flexible type assignment and further eases the implementation of multiple, simultaneous security policies for different stakeholders (see below and Section 4.3.2). After resolving the security types, the USSS performs the access control decision $\mathcal{AC}(subject\_type, object\_type, object\_class, operation)$ as defined in Section 4.3.1 and returns the decision to the USOM for enforcement.

Multiple stakeholders.

As explained in Section 4.3.2, multiple stakeholders may have interest in deploying access control rules on the device to protect their respective assets. In our implementation, we opted for supporting policies included in application packages (APKs) of 3^{rd} party apps in addition to the pre-installed system policy. The PackageManagerService is instrumented such that it extracts policy files included in APKs during app installation and injects them into the USSS. To differentiate between the different policies, we leverage namespaces. In each namespace, security types and objects classes can be defined that are independent of the other namespaces. Thus, for instance, app developers can define app-specific policies with custom app and Intent types which are valid only within their own namespace.

Figure 5 illustrates policy checks with namespaces, here *System*, *Object*, and *Subject*. Since 3^{rd} party app policies should not interfere with other 3^{rd} party app functionality, except when the other apps interact with the app, the Object and Subject policies during the check are the ones deployed by the Object and Subject app. If the Subject or Object app is a *system* app, their policies are covered by the mandatory *System* policy. These Subject and Object policies are optional, i.e., they are only available if the 3^{rd} party developer opted in to use FlaskDroid and deployed a policy in the APK of his app. If either Object or Subject app did not deploy a policy, the respective check always yields *Allow*.

Since the Subject and Object policies are specific to the app that deployed it and are only used when the app is involved in an access control decision, it would be inefficient if we would require authors of app-specific policies to define extra app type definitions for their own app. Thus, we allow in these policies a special, reserved type *self_t* for the subject or object type of an allow rule. This special type represents the app that deployed this rule. For instance, the rule

$$r = (\mathsf{trustedApp\_t}, \mathsf{self\_t}, \mathsf{contentProvider\_c}, \mathsf{query})$$

would define, that any app with security type trustedApp is allowed to query a ContentProvider component belonging to the app that deployed that rule. When querying an app-specific policy, our USSS automatically adjusts the subject or object argument of the query depending on whether the app owning this policy is acting as subject or object. In Figure 5, for instance, the object type parameter would be automatically mapped to the *self_t* type for the second check based on the *Object* policy policy and similarly the subject type parameter in the third check to the *Subject* policy would be automatically set to *self_t*.

Figure 5 also shows that in our implementation we opted for a conservative consensus approach for policy reconciliation, i.e., at least one policy checks allow access and none denies. Since the *system* policy check is mandatory, the system policy must always consent into an access.

Context-awareness.

To implement context-aware policies, we implement different Context Providers, which register Listener threads to be notified about context changes similar to the approach taken in [19]. In our current implementation, the context is derived from the location information provided by the GPS sensor, the user presence detected by the ActivityManagerService, the ActivityStack, and a notification Intent by the telephony app. These information are matched against defined contexts within the Context Providers. For instance, the notification Intent is send by the telephony app, when the user activated/deactivated the *Phone Booth Mode* (see Section 6.6) and the geolocation is used to determine if the user is currently at his workplace. When a Context Provider

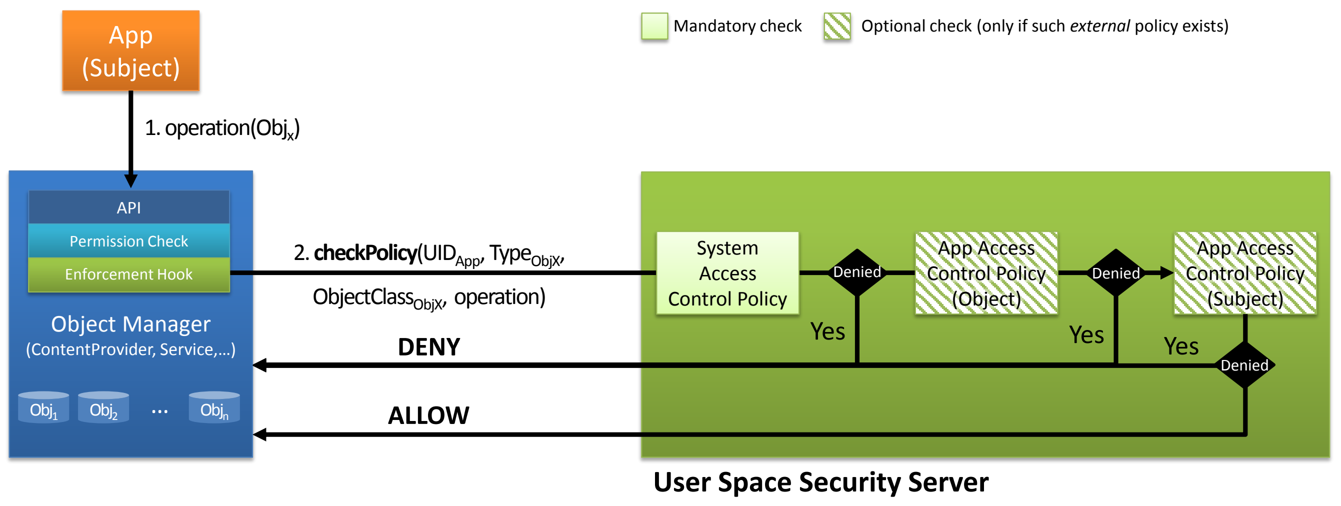
Mandatory check
Optional check (only if such *external* policy exists)
App (Subject)
1. operation(Obj_x)
API
Permission Check
Enforcement Hook
Object Manager
(ContentProvider, Service,...)
Obj_1
Obj_2
...
Obj_n
2. checkPolicy(UID_App, Type_ObjX, ObjectClass_ObjX, operation)
DENY
ALLOW
System Access Control Policy
Denied
Yes
App Access Control Policy (Object)
Denied
Yes
App Access Control Policy (Subject)
Yes
Denied
User Space Security Server

Figure 5: *Interface between Userspace Object Managers and Userspace Security Server. Additionally, shows the access control decisions in the Userspace Security Server based on policies of different stakeholders system, subject app developer, and object app developer. Here, a conservative consensus policy reconciliation of the policies. Subject and object policies are optional, since external 3^{rd} parties may not deploy policies.*

matches these criteria to a defined context, it informs the USSS that the corresponding context declared in the policy (e.g., `phoneBooth_con` or `work_con` in Listing 3) is active/inactive. The USSS than maps the context to boolean values as defined by the *switchContext* statements in the policy (cf. Section 4.3). Abstractly, this equals setting the parameter p_5 as defined in Section 4.3.1.

Synchronisation with Kernel MAC.

As explained in Section 4.4.1, our middleware extensions take the role of a trusted user space agent which manages SE Android's booleans (represented by *kbool* variable in the policy and used in *switchBoolean* statements, cf. Section 4.3). SE Android provides to this end user space support (in particular *android.os.SELinux*), which enable runtime configuration of the kernel MAC booleans. Self-contained system and kernel MAC policies ensure that only the system server is allowed to use this mechanism.

5.3 Userspace Object Managers

We instrumented essential Android subsystems like the PackageManagerService and ActivityManagerService, as well as System Applications like the ContactsProvider, the MMS-SMSProvider, or the LocationManager as Userspace Object Managers in order to enforce fine-grained policy-driven access control on *objects* they manage. After analyzing the security-critical operations regarding the objects these components manage, we integrated hooks to query the USSS for policy decisions at strategical valuable locations in the control flow from public APIs of the respective USOMs.

PackageManagerService.

As explained in Section 4.4, the PackageManagerService is responsible for (un)installation of application packages and for maintaining a global list of application components that could be queried at runtime in order to perform a specific task.

The PackageManagerService is instrumented as a USOM by the following extensions: (1) introducing a mechanism to tag newly installed and pre-installed applications with security labels and maintaining a list of labels assigned to all installed applications, and (2) introducing policy enforcement points in APIs that query information about other packages or access the PackageManagerService's global list of application components to obtain a suitable component that can perform a task on another application's behalf e.g., an Email client calling a PDF reader to read an email attachment.

Security labeling of application packages is achieved by extending the relevant Java class to include a security label field for each package (Shared UID), and by hooking the PackageManagerService's function calls responsible for installing apps with calls to label them during the installation process; pre-installed apps are labeled during the phone's boot cycle. It is important to note that the criteria for labeling apps with a particular security label is based on meta information in the application package e.g., granted permissions, developer's signature etc. The exact meta information chosen to label the app is stipulated by the system policy.

In addition, pre-defined UIDs in the system are reserved for particular system components, for instance daemons. Listing 8 shows the pre-defined UIDs on Android 4.0.4 as found in *system/core/include/private/android_filesystem_config.h*. With the exception of the system UID (`AID_SYSTEM`) and NFC UID (`AID_NFC`) these UIDs are not assigned to an application package managed by the PackageManagerService. Thus, in our implementation we map these UIDs by default to pre-defined types (e.g., `aid_root_t` or `aid_audio_t`) which are hence mandatory types in our system policy.

Secondly, policy enforcement points are introduced in those function calls of PackageManagerService that query a preferred component to perform a specific task at runtime. For instance, when a third-party application with a security label S_A wants to read a document, the PackageManagerService is queried to look for suitable document readers. After the PackageManagerService *resolves* a suitable document reader to perform the required task, our policy enforcement points

Listing 8: *Pre-defined, reserved UIDs for Android system components*

```
AID_ROOT 0 /* traditional unix root user */
AID_SYSTEM 1000 /* system server */

AID_RADIO 1001 /* telephony subsystem, RIL */
AID_BLUETOOTH 1002 /* bluetooth subsystem */
AID_GRAPHICS 1003 /* graphics devices */
AID_INPUT 1004 /* input devices */
AID_AUDIO 1005 /* audio devices */
AID_CAMERA 1006 /* camera devices */
AID_LOG 1007 /* log devices */
AID_COMPASS 1008 /* compass device */
AID_MOUNT 1009 /* mountd socket */
AID_WIFI 1010 /* wifi subsystem */
AID_ADB 1011 /* android debug bridge (adbd) */
AID_INSTALL 1012 /* group for installing packages */
AID_MEDIA 1013 /* mediaserver process */
AID_DHCP 1014 /* dhcp client */
AID_SDCARD_RW 1015 /* external storage write access */
AID_VPN 1016 /* vpn system */
AID_KEYSTORE 1017 /* keystore subsystem */
AID_USB 1018 /* USB devices */
AID_DRM 1019 /* DRM server */
AID_AVAILABLE 1020 /* available for use */
AID_GPS 1021 /* GPS daemon */
AID_UNUSED1 1022 /* deprecated, DO NOT USE */
AID_MEDIA_RW 1023 /* internal media storage write
        access */
AID_MTP 1024 /* MTP USB driver access */
AID_UNUSED2 1025 /* deprecated, DO NOT USE */
AID_DRMRPC 1026 /* group for drm rpc */
AID_NFC 1027 /* nfc subsystem */

AID_SHELL 2000 /* adb and debug shell user */
AID_CACHE 2001 /* cache access */
AID_DIAG 2002 /* access to diagnostic resources */
```

query the USSS to check if the resolved document reader can in fact be used by the caller. To facilitate the policy check, PackageManagerService passes security labels of the caller (S_A), the callee (say S_B) and the operation to be performed ("package query") to the USSS. Thus, our policy enforcement points ensure that runtime component queries adhere to our userspace system policy.

Activities and Broadcasts.

As explained in Section 4.4, the ActivityManagerService is responsible for the life-cycle management of Activities as well as providing the Intent broadcasting subsystem. More explicitly, the ActivityStack subsystem of the ActivityManagerService is responsible for starting/destroying Activities, maintaining which application is shown in the foreground on the screen, or reordering the stack of Activities.

We extended the relevant management functions of ActivityManagerService for Activities (i.e., GUI objects, cf. Section 2.1) with calls to the USSS in order to verify that the particular functions are permitted to proceed depending on the subject type (i.e., the calling app), object type (i.e., the app owning the Activity being modified), and the current phone state. Thus, it provides fine-grained access control over the Activity operations such as moving another Activity into the foreground on the screen.

For access control on Intent Broadcasts, we followed a design pattern as proposed in other works [59, 12]. Similar to these works, we verify for each sender-receiver pair of a Broadcast that this communication is policy compliant and remove non-compliant receivers before executing the broadcast operation. Again, the check is based on the subject type (i.e., the app receiving the Broadcast), the object type (i.e., the app broadcasting), and the phone state. Moreover, additional checks based on the Intent type have been added to verify that the broadcasting app is allowed to send this type of Intent, that the receiver is allowed to receive this type of Intent, and if the sender declassified the Intent.

Services.

We extended the code generator of Android's AIDL tool to automatically include a policy check within each function (declared in the applications AIDL file) of the *onTransact* method responsible for multiplexing incoming calls to the Service. We further extended the lexer and parser of Android's AIDL tool to recognize if an interface function declaration is tagged with an object type and thus add a second policy check to the tagged functions' code with respect to the object type declared by the developer. Modifying the AIDL tool is inspired by the approach presented in [21]. Since the AIDL tool is also used during build of the system, these extensions affect all system Services as well and provide a very convenient way to enforce fine-grained policy-driven access control to system Services' interfaces.

Content Providers.

As argued in Section 4.4, how a ContentProvider can work as a Userspace Object Manager depends on the storage backend of the Provider. However, we implemented one particular design pattern for SQLite database-based ContentProviders. For a detailed technical description of this mechanism, we refer to our related technical report [13]and provide here a summary at the example of the ContactsProvider. In the system ContactsProvider, we leverage the contacts' groups (e.g., "friend", "work", "family", etc.) and the mimetype of contacts' data (e.g., "email address", "postal address", etc.) as object types. 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. Filtering queries to the database is technically more involved. Our solution is based on SQL Views[9] by creating one View for each subject type and redirecting the query of each calling app to the View for its type. Each View implements a filtering of contacts/contacts' data based on an access control table that represents the access control matrix for subject/object types. The values of the access control matrix are retrieved from the USSS upon database creation or when a new application with permission to access this provider is installed. The same design pattern can be leveraged on any SQLite database-based ContentProvider and similarly the retrievable access control matrix can be leveraged by any kind of ContentProvider. Further, this technique scales well to multiple stakeholders by using nested views, i.e., each level of the nesting corresponds to one stakeholder.

[9]A virtual SQL table, which represents a subset of the original table's content based on an inherent select statement. Views can be used instead of Tables in SQL commands.

6. USE-CASES / INSTANTIATIONS

As mentioned in our threat model (cf. Section 3.1), the recent incidents on smartphone privacy and security breaches have led to a demand for privacy protecting as well as (enterprise) security solutions in practice. In the following we will introduce some of them and show how FlaskDroid can instantiate them. Moreover, we will discuss how FlaskDroid can go beyond these specific solutions, for instance, by securely integrating virus scanner or firewall management apps into the overall system.

6.1 Privacy Enhanced System Services and Content Providers

System Services and ContentProviders are an integral part of the Android application framework and implement the API exposed to 3^{rd} party apps. 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: However, it is known that the default permissions are too coarse-grained and protect access only to the entire Service/Provider but not to specific functions or data. Moreover, they are static and cannot be selectively revoked. Thus, the user cannot control in a fine-grained fashion which sensitive data can be accessed how, when and by whom. For instance, the popular WhatsApp service has recently been shown to upload user's contacts data [7]. Also other apps such as Facebook have access to the entire contacts database although only a subset of the data (i.e., email addresses) is required for their correct functioning. On the other hand, recent attacks demonstrated how even presumably privacy-unrelated and thus unprotected data such as the accelerometer readings can be misused against user's security and privacy [85, 14].

Solution: Our modified AIDL tool automatically generates policy checks for each Service interface and function in the system during build of the system, thus enabling policy-based access control to each system Service interface and function. We tagged selected query functions of the system AudioService, LocationManager, and SensorManager with specific security contexts (e.g., *object_type* asfineGrainedLocation_t, *object_class* as locationService_c, and *operation* as getLastKnownLocation) to achieve fine-grained access control on this information. Our policy could state, 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 would not be possible when, e.g., the virtual keyboard/PIN pad is in the foreground or a phone call is in progress (both definable phone states).

In Section 5.3 we explained how ContentProviders (e.g. the ContactsProvider) can act as Userspace Object Managers. As an example, users can refine the system policy access control to their contacts' data. A user can, for instance, grant the Facebook app read access to their "friends" and "family" contact's email addresses and names, while prohibiting it from reading their postal addresses and any data of other groups such as "work". For technical on how this is implemented and on the policies, we refer to our technical report [13].

6.2 Privacy-Enhanced Image Media Store

Modern Android smartphones are equipped with cameras. Photos a user takes may contain sensitive information, where the sensitivity is defined by the current usage context of the device. For example, in case a device is used for business and private purposes, photos taken while on company premises or during working hours should not be accessible by private apps. In addition, a user may want to protect his private images from being accessed by the employer.

Problem description: When a photo taken by the user is stored on the device, meta-information about the photo (filename, location etc.) and the photo file itself are accessible by all apps which can access the external storage area. While meta-information is stored in the Mediaprovider ContentProvider, the photo itself is stored on the external storage area of the device, which is implemented using either an embedded flash module or a removable SD card. Android uses the VFAT file system for this storage area, which does not provide fine-grained access control. While recent Android versions use a file system which supports access control, Android emulates a VFAT file system on top of it by means of a *File System In Userspace (FUSE)*. A reference monitor in the kernel checks Linux GID based permissions (`READ_EXTERNAL_STORAGE`/`WRITE_EXTERNAL_STORAGE`) whenever apps try to access this storage area. Thus, all apps which hold these permission can access all photos.

Solution: In FlaskDroid, the sensitivity of a photo is defined by the usage context a photo was taken in. This information can either be derived from user input by asking the user whenever a photo is taken, or by context deriviation from sensor information (cf. Section 4.4.4). We assign a security context to the photo metadata stored by the camera app in the Mediaprovider, which acts as a Userspace Object Manager.

A technical challenge for access control on the photo files themselves is the fact that the VFAT file system does not support extended file system attributes, a prerequisite for storing the SELinux security context for file objects. When the VFAT file system is emulated by the previously described FUSE module, this module can be instrumented to act as a Userspace Object Manager and mediate access based on the type of the accessing app. Alternatively, a file system with support for extended attributes for the external storage area, as proposed in [23], can be used.

It should be noted that when using removable media, FlaskDroid's access control mechanisms can be circumvented by removing the SDCard and accessing it from a secondary device. This challenge can be addressed by encrypting the contents of the SDCard with a key bound to the mobile device.

6.3 Multi-level Security

Smartphones are increasingly applied in scenarios where different security domains are desired. The most prevalent example today are corporate smartphones which are simultaneously used for private purposes or vice versa (i.e., based on *"Bring your own device"* philosophy). Previous work [12] has acknowledged this need for domain isolation on dual used smartphones and our FlaskDroid architecture can efficiently and effectively instantiate this use-case.

Problem description: Android does not provide support for declaring different security domains and strongly isolating them from each other. Thus, malicious apps in or attacks against the private domain on the phone can compromise the corporate domain.

Solution: Our architecture supports multi-level security (MLS) and thus with FlaskDroid business applications can be clearly distinguished and labeled with a corresponding type during installation (cf. Section 4). At runtime, the non-business and business domain are securely isolated from each other at middleware level and kernel level. For instance, only apps from the business domain are permitted to contact the Services of another business domain app. Moreover, contacts created for the business domain (i.e., of group "work") could only be read and written by application from the business domain (see also the use-case in Section 6.1). Similarly, our consolidated kernel- and userspace policies can ensure that despite DAC, files created by a business app could only be accessed by other business apps.

6.4 Secure Logs

A further use-case of our interest concerns the log facility of Android. Android applications can write entries to the *log* facility by either writing directly to the world-writeable */dev/log/** device files, by using the *log* and *logwrapper* tools or the Android Log API. By modifying the Android-specific *logger* kernel driver, the UID of the application writing to the logfile is saved upon write. On the middleware level, the UID of the application is associated with the userspace subject type.

Problem description: Applications holding the *READ_LOGS* permission effectively become member of the log group which has read access to */dev/log/** (enforced by the Linux DAC, similar to the Internet permission; cf. Section 2.1)). Applications by default use the *logcat* tool available on the phone to read and parse the entries in */dev/log/**[10]. However, the capability to read the logs has been shown to be a security and privacy threat. For instance, sensitive location information are frequently logged by apps and thus could be retrieved from the log [48] and even Facebook credentials were discovered in log entries [61].

Solution: SELinux is used to enforce that only the Android *logcat* tool is allowed to read from */dev/log/**. With *logcat* now being the only access point to log entries, it can be extended as a Userspace Object Manager: Upon read access, the log facility filters all entries from the result, for which the reader does not have the required security clearance. Policy checks from the logcat tool to our middleware USSS are performed via a dedicated socket provided by the USSS. This strongly resembles the exemplary use-case for the SELinux access control on the */etc/passwd* file on Linux systems. Only the *passwd* tool is allowed to modify this security sensitive file and is inherently trusted to enforce that users can only modify their own entries. However, in contrast to this default use-case, our secure logs require the joint operation of both middleware and kernel-level. Listing 9 presents a simple policy for enforcing access control on log entries. In this case, every application has read access only to his own logging information in the default log, but no other (line 1). Only apps belonging to the system application attribute are allowed to read all log entries from both the default log and the system log (line 2).

It should be noted that since Android version 4.1, the `READ_LOGS` permission is not available to 3rd party apps anymore[11]. In comparison, our approach allows 3rd apps to access their own log entries, which is a benefit for application developers and preserves legacy-compliance.

Listing 9: *Example policy snippet illustrating implementing different security types on log entries*

```
1 self: log_c { read };
2 allow systemApp_a any: { log_c syslog_c } { read };
```

6.5 Firewall and Anti-Virus Apps

In default Android, certain permissions like Internet or Bluetooth are mapped to Linux groups (cf. Section 2.1). Similarly, our architecture is used to introduce new capabilities, which, in contrast to permissions, can be revoked at runtime using booleans. As example, we introduced the capabilities that a 3rd party app can act as a manager for the Linux kernel packet filter, i.e., it can execute the *iptables* tool with sufficient privileges, and the capability to inspect other apps' APK packages (both, the public and the private portion, cf. *forward locking*) in order to perform anti-virus scans. While this could be achieved by adding new permissions to the system, using FlaskDroid is much more flexible and allows for efficient upgrades of the policy as well as more fine-grained access control.

Problem description: Default Android does not support this use-case and requires the user to root his phone to enable contemporary available firewall and anti-virus apps with the above described functionality. Naturally, this severely impedes the platform security and opens the door for other malware that leverages the root privileges.

Solution: We install the *iptables* binary with setuid bit and user root. Additionally, we assign this binary an SELinux security label with *fwapp* as role, so that only apps that have been assigned this role (or that inherited this role) can execute the binary. Similarly, we introduce a role *avapp* which is allowed to inspect the private files of other apps.[12] These SELinux roles are assigned to apps during installation (i.e., to their UID) and in future work we plan to extend this install process such that apps can requests roles in their manifest and depending on user consensus or their app type (cf. Section 4.3) the role is granted or denied to the application.

6.6 Phone Booth Mode

Users may lend their mobile device to an acquaintance (or even stranger) to make a phone call. The goal of this use-case is to temporarily lock the device in a secure state in which it can be handed out to another person for the sake of making a phone call. In this mode, the phone is configured to grant access only to telephony-related features and apps (c.f. Figure 6), and access to privacy-sensitive data, such as contacts and call log entries, is denied.

Problem description: The user loses or gives up physical control over his device and has to trust the other person to not violate his privacy by, for example, inspecting the entire call history and contacts database visible within the

[10] While it is possible to read */dev/log/** directly, in practice, all applications we analyzed use the logcat tool.

[11] `http://code.google.com/p/android/issues/detail?id=34792`

[12] This includes, that the DAC permissions are relaxed, since the access must be allowed by DAC **and** MAC

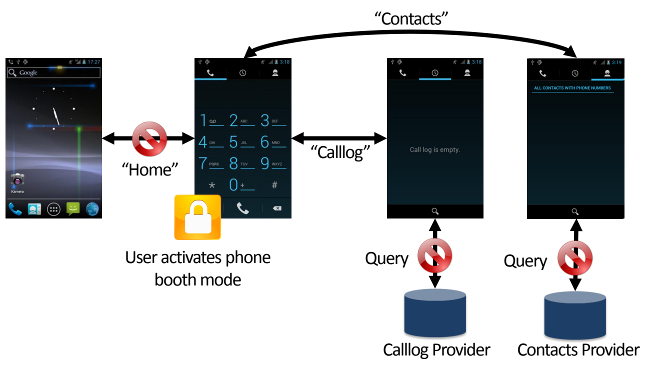

Figure 6: *Phone Booth Mode*

dialer app or even exploring other apps and data on the phone.

Solution: Our fine-grained access control within the ActivityManagerService and ActivityStack allows us to control if another application can be moved into or off the foreground on the screen (cf. Figure 6). The user can trigger a transition into a special phone-context "phone booth mode" by pressing a button, in which the policy dictates that the dialer app is the only app permitted to be in the foreground. Simultaneously, the policy defines that all access to the CallLogProvider and ContactsProvider must be denied, meaning they return empty results to all queries, thus preventing the phone app from auto-completing entered telephone numbers or from showing the call log/contacts data. In this mode, the user can safely lend his phone to another person for the sake of making a phone call.

To exit this specific phone state, the phone app informs the Userspace Security Server about the changed state and thus resets the corresponding Boolean value to its original state. To authenticate this operation, different options exist. For instance, the user is prompted to enter his PIN or configured password (or on more recent hardware, using biometrics) and only if the authentication succeeded, the phone app changes the state.

Moreover, the same approach could be used to provide a generic "kiosk mode" as well, which locks the user into a single application, such as a game when handing the phone to a child.

The policy for implementing the phone booth mode was already used in listings in Section 4.3, but listing 10 presents summary of the relevant parts. The policy defines a separate boolean `phoneBooth_b` (line 1) to represent the phone state, i.e., either the phone booth mode is activated or not. To manage this boolean, we use a dedicated context `phoneBooth_con` (line 3) and switchBoolean (lines 5-10), which defines that on activating this switch the `phoneBooth_b` boolean is set to true and automatically reset to false if the `phoneBooth_con` context is deactivated. The snippet shows the rules for normal operation of the phone (lines 12-30), including querying contacts, switching Activities, etc. However, part of these operations are only allowed if the phone is *not* in phone booth mode (lines 26-30), meaning that this rules are disabled if the phone booth mode is active (i.e., the `phoneBooth_b` boolean is true). The affected rules address the switching of Activities (line 28) and the querying of the contacts database (line 29). Since the phone booth mode is activated from within the phone app, which is shown in foreground, deactivating these rules results in forcing the phone app to stay alive and in foreground on the screen as well as blocking any access by the phone app to the contacts data (and calllog).

Listing 10: *Example policy snippet implementing a phone booth mode.*

```
1  bool phoneBooth_b = false;
2
3  context phoneBooth_con;
4
5  switchBoolean
6  {
7      context=phoneBooth_con;
8      auto_reverse=true;
9      phoneBooth_b=true;
10 };
11
12 self: app_c {checkPermission};
13 self: activity_c {finish moveTask};
14 self: broadcast_c {receive send};
15
16 allow {app_system_t app_contacts_t app_launcher_t}
         allContactsData_t: contactsProvider_c {query};
17
18 allow {app_system_t app_telephony_t app_contacts_t
         app_launcher_t} {app_system_t app_telephony_t
         app_contacts_t app_launcher_t}: package_c
         {getPackageInfo getPackageInfoWithUninstalled
         getPackageUID getPackageGIDs getPackagesForUid
         getNameForUid getUidForSharedUser
         findPreferredActivity queryIntentActivities
         getInstalledApplications
         getInstalledApplicationsWithUninstalled
         getInstalledPackages
         getInstalledPackagesWithUninstalled};
19
20 allow {app_system_t app_telephony_t app_contacts_t
         app_launcher_t} {app_system_t app_telephony_t
         app_contacts_t app_launcher_t}: app_c
         {checkPermission};
21
22 allow {app_system_t app_telephony_t app_contacts_t
         app_launcher_t} {app_telephony_t app_contacts_t}:
         activity_c {start};
23
24 allow {app_system_t app_telephony_t app_contacts_t
         app_launcher_t} {app_system_t app_telephony_t
         app_contacts_t app_launcher_t}: activity_c {moveTask
         finish};
25
26 if(~phoneBooth_b)
27 {
28     allow {app_system_t app_telephony_t app_contacts_t
             app_launcher_t} {app_system_t app_telephony_t
             app_contacts_t app_launcher_t}: activity_c {start
             moveTask finish};
29     allow app_telephony_t allContactsData_t:
             contactsProvider_c {query};
30 };
```

6.7 App Developer Policies (Saint)

Ongtang et al. present in [59] an access control framework, called *Saint*, that allows app developers to ship their apps with policies that regulate access to their app's 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.

Referring to our notation of access control in Section 4.2, the runtime enforcement of Saint is defined as a whitelisting access control $\mathcal{AC}^{Saint}(p_1^{Saint}, p_2^{Saint}, p_3^{Saint}, p_4^{Saint})$ with the following parameters:

p_1^{Saint} = Source	Source app and optional parameters for an Intent object (e.g., action string)
p_2^{Saint} = Destination	Destination app
p_3^{Saint} = Conditions	Optional, conjunctional conditions (e.g., permissions or signature key of the destination app)
p_4^{Saint} = $\text{State}_{\text{System}}$	System state (e.g., physical location or bluetooth en-/disabled)

Solution: Instantiating Saint's runtime access control on FlaskDroid is achieved by mapping Saint's parameters to the ones supported by FlaskDroid: p_1 = *subject_type*, p_2 = *object_type*, p_3 = *object_class*, and p_4 = *operation* (cf. Section 4.3). Thus, p_1^{Saint}, p_2^{Saint}, and p_3^{Saint} can be 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 application 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 represented by FlaskDroid policy.

Listing 11 shows an instantiation of the developer policy in [59] on our architecture. The depicted 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 8-22) for the specific apps with which the shopping app is allowed to interact and describe some of the allowed interactions by means of Intent types `intent_actionPay_t` and `intent_recordExpense_t` (lines 5-6 and lines 24-28). Afterwards, we declare access control rules that reflect the desired policy described in the example in [59] (lines 36-38). For instance, the rule in line 36 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 27), which in turn is only assigned to apps with the developer signature `308201...` (line 10). The rule in line 37, on the other hand, defines that the shopping app is allowed to interact in any way with an app with type `app_trustedPWVault`, which is only assigned to apps with package name `com.secure.passwordvault` and minimum version `1.2` (lines 15-16). Naturally, the developers of the other apps (e.g., of `com.secure.passwordvault` or with signature `308201...`) have to ship corresponding policies to concur to this interactions (e.g., receiving Intent from the shopping app).

Listing 11: *Policy snippet showing instantiation of Saint [59] example for runtime policy enforcement. Policy snippet is from the policy deployed by the shoping app.*

```
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  {
10     Developer:signature=308201...;
11 };
12
13 appType app_trustedPWVault_t
14 {
15     Package:package_name=com.secure.passwordvault;
16     Package:min_version=1.2;
17 };
18
19 appType app_noInternetPerm_t
20 {
21     Package:permission=~android.permission.INTERNET;
22 };
23
24 intentType intent_actionPay_t
25 {
26     Action:action_string=ACTION_PAY;
27     Components:receiver_type=app_trustedPayApp_t;
28 };
29
30 intentType intent_recordExpense_t
31 {
32     Action:action_string=RECORD_EXPENSE;
33     Components:receiver_type=app_noInternetPerm_t;
34 };
35
36 allow self_t intent_actionPay_t: intent_c { send };
37 allow self_t app_trustedPWVault_t: any { any };
38 allow self_t intent_recordExpense_t: intent_c { send };
```

Although both the original Saint policy and its instantiation on our solution achieve the same security objectives, the policy language between the two systems differ. Most noteworthy is, that in Saint's policy language the subjects and objects are defined within the access control rules. Thus, if two rules use identical subjects and objects, the definition of those is redundantly repeated. Moreover, in Saint it is not possible to group several distinct subjects or objects into sets and thus ease writing of rules which differ only in the object class or operation. In FlaskDroid, the policy language requires the policy author to first clearly define all possible types (for subjects and objects) as well as object classes and their respective operations. While this might seem at first glance more tedious, it greatly eases authoring of policy rules afterwards since one can use these types without redundantly repeating their definition and, moreover, can group types, classes, or operations into sets and thus achieve more compact and readable rules.

7. EVALUATION AND DISCUSSION

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

7.1 Policy

Policy Assessment.

Security policies, including access control policies, are generally evaluated to be *good* based on different properties. Some of the most important properties are *safety*, *completeness*, and *effectiveness*.

Safety: A policy is *safe* when it enforces that subjects can only obtain the rights that were intended for them but no other [38].

Completeness: However, since the verification of the safety property was shown to be undecidable [38] for general access control models (like RBAC or Lampson's access matrix [47]), alternative approaches to achieve the safety property were devised. For instance, *constraints* were introduced which define a safety policy. Each change in policy configuration is verified against this safety policy to ensure that it preserves the safety property. This concept is also used in SELinux, where constraints statements define prohibited permission assignments [75] (e.g., a process could be prevented from creating a file with a different user identity then this process). However, constraints do not only noticeably increase the policy complexity and thus easily introduce conflicts in the access rights specification, but they also subdivide the space of possible policy configurations in different, meaningful subspaces. These subspaces are denoted *access control spaces* [43, 42]. In particular, there is a subspace of permission assignments that are permissible by the policy configuration, a subspace of assignments prohibited by the constraints, and a subspace of assignments which are neither permissible nor prohibited, i.e., the assignment status is *unknown*. A policy is called *complete* when all unknown assignments are eliminated.

Effectiveness: Further, a policy is called *effective* when it enforces least privilege, i.e., subjects hold only the privileges they require for their tasks. Effectiveness, for instance, can also depend on the granularity of the enforcement points (i.e., in our policy the granularity of the object classes and their operations).

Our derived example policy: The development of a *good* 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 [31]. A similar development can be currently observed for the SE Android policies which are written from scratch [74]. 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, and hence lay the foundation for the development of a *good* policy.

Policy Generation.

Established approaches: Different approaches exist to generate access control policies. For instance, manual authoring of very special purpose and use-case specific policies as we have shown in Section 6. For more complex policies, (semi-)automatic methods have been proposed. For example, the *polgen* tool for SELinux [77] processes the traces of the dynamic behavior of target process (e.g., information flow patterns) and generates new types and policy rules. *polgen* operates human-aided and semi-automated, since a human has to determine the exact security policies and adjust the generated policies via wizard-style interface. Similarly, the benefits of using the system call traces for guided generation of a policy for fine-grained process confinement have been shown [64]. Also static analysis of target binaries seems a feasible approach to help automating policy generation and has been already employed to verify the information flow properties of SELinux policies [70].

Policy generation for our evaluation: To evaluate our FlaskDroid architecture, we derived an example policy that covers the *pre-installed* system apps that we explained in Section 4.4. To generate this example policy, we opted for an approach that is similar to the one used in TOMOYO Linux' learning phase[13]. The underlying idea of this approach, as illustrated in Figure 7, is to derive policy rules directly from observed trusted application behavior. In our evaluation, we consider the set of *pre-installed* system applications during the time-frame of testing as being trustworthy.

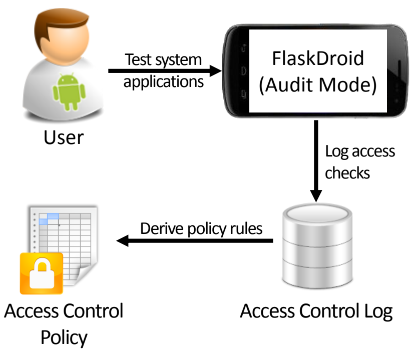

Figure 7: *Access control policy generation for system apps based on audited application behavior.*

To observe the application behavior, we deploy on our test devices our FlaskDroid architecture in auditing mode, i.e., permission denials are only logged but not enforced, and with a pre-installed policy. The deployed policy contained a complete set of pre-configured types, attributes, classes, and operations for the available system object managers, but an empty allow rule set $\mathcal{R}$ (cf. Section 4.2). The types, attributes, and classes have been derived from manual analysis of the USOMs described in Section 4.4. We simply refer to this pre-configured policy as *No-allow-rule* policy in the following. Since the policy did not contain policy rules, every access control check based on this policy would result in an access control violation. But, because the system is in auditing (i.e., non-enforcing) mode, all violations are merely logged. Thus, after this auditing phase one obtains a log containing a clear trace of all accesses that occurred in the system and that describes the applications' behavior. Under the assumption, that the system contained (in this learning phase) only trusted apps, this trace can be used to generate policy rules which limit the applications' access rights to the ones observed during the learning phase.

A crucial aspect of this approach is to avoid unexpected access denials at runtime (after the learning phase) and hence requires that one observes/logs all possible behavior of the target apps during the learning phase. Otherwise, the policy set is incomplete and legitimate actions of applications are denied. For instance, if a trusted contacts management app never inserted but only queried contacts information during the learning phase, no policy rule is derived that allows

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

this app to insert contacts information (after the learning phase). In order to achieve a high coverage of application functionality and thus log all required access rights of the pre-installed system apps, we opted for testing with human user trials. We chose this approach because of the following reasons: First, automated testing has been shown to exhibit a potentially very low code coverage [30, 31] and, second, Android's extremely event-driven and concurrent execution model complicates static analysis of the Android system [88, 31].

To achieve the required high code coverage, the users' task was to thoroughly use the pre-installed system apps by performing various every-day tasks (e.g., maintaining contacts, writing SMS, browsing the Internet, or using location-based services). While these tasks primarily trigger interaction between apps and the system components (e.g., location manager or sensors), we were also interested in the possible interaction between apps. Thus, 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 or contacts entry), which defines the required access control rules across app-boundaries. Since this testing targeted only pre-installed system apps, no 3^{rd} party apps were allowed during testing. For testing, the users were handed out Samsung Galaxy Nexus devices running FlaskDroid in auditing mode. The devices were also pre-configured with test accounts (e.g., GMail) and test data (e.g., fake contacts data). Using the logged access control checks from the user trials, we derived access control rules of the form ($subject\_type, object\_type.object\_class, operation$) required for the correct operation of the system components. These rules (together with the above stated type and object definitions) constitute our *derived example policy*.

Derived example policy: Table 2 provides an overview of the example policy derived from all user test results and the additionally deployed SEAndroid policy (above the double line in the Figure). In total 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). For instance, 57 types define the various sensor types and possible data rates of each sensor (e.g., fastest data rate for the orientation sensor), additionally grouped in 5 attributes to ease policy design. The derived policy after evaluating the audited user tests contained 109 allow rules which grant each pre-installed system app only the rights necessary for correct operation (as observed during testing).

Comparison to established SELinux policies: Table 2 provides statistics on the policy complexities of different SELinux deployments such CentOS, Fedora, and Debian, as well as the SELinux reference policy. Obviously, these policies cannot be directly compared to FlaskDroid since they target desktop operating systems and include policies for various pre-installed 3^{rd} party applications. However, the difference in policy complexity (which is in the order of several magnitudes) clearly underlines our second observation (cf. Section 1) and the experiences from [86, 74, 73, 54], that the design of mobile operating systems facilitates a clearer mandatory access control architecture (e.g., separation of duties). This clearer architecture profits an easier security policy design.

Lesson learned: A particular lesson learned from deriving the example policy in FlaskDroid is that several permissions operationally depend on each other. For instance, the permission check for starting an Activity followed almost always a permission check for *queryIntentActivity* to resolve the target Activity. Thus, a launcher app which starts other apps would by default require the permission to perform *queryIntentActivity* for any other app in order to function properly. Similarly, the application Context class inquires information about its corresponding app from the system PackageManagerService upon initialization. Thus all 3^{rd} party apps require in general the permission to perform the *getPackageInfo* operation. These insights can be used to further optimize the system policy (e.g., establishing sets of fundamental privileges for different application categories like *Launcher App*) and we intent to investigate these relations further in future work.

3^{rd} Party Policies.

The derived example policy can act as the basis policy on top of which additional user, 3^{rd} party, and use-case specific policies can be deployed (cf. Section 6). In particular, we are currently extending the example 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 7.2). A particular challenge to tackle, 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 investigations on how the user can be involved in the policy configuration [90, 13].

However, as discussed in Sections 3 and 4.4.3, multiple policies by different stakeholders with potentially conflicting security objectives requires 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. For instance, [66] presented policy conflict resolving in the particular case of the NTFS file-system and [41, 15] address conflict detection and resolution in firewall policies. We explained further possible strategies in Section 3.

In our current approach, we opted for a consensus approach with deny precedence (i.e., the access is denied if one policy denies it) and a mandatory system policy (cf. Section 5.2) to resolve conflicts between 3^{rd} party policies and the system policy. Thus, 3^{rd} party policies are generally independent of the system policy and they can only refine the access rights of their host app as granted by the system policy. Resolving conflicts between two 3^{rd} party policies is out of scope of our conflict resolution, since we cannot interpret the high-level security objectives that the respective 3^{rd} parties had in mind when developing their policies. In this case we apply again a consensus strategy, i.e., both 3^{rd} party policies must allow an operation, otherwise access is denied.

7.2 Effectiveness

Authorization of security relevant operations: It is important that all operations that may access security or privacy critical data are correctly mediated and authorized by an access control policy. For instance, in SELinux this means that the hooks of the underlying Linux Security Module framework are correctly placed and thus enforce access control checks on security relevant operations. Related work [22]

Policy	# Types	# Attributes	# Classes	# Permissions	# Rules
SEAndroid (Master branch, checkout 12/04/2012)	232	19	84	249	1359
FlaskDroid middleware MAC (example policy from 12/04/2012)	111	9	18	63	109
SELinux reference policy (v2.20120725, no distribution option)	661	132	81	239	278
SELinux Fedora 17 (targeted, policy.27 from 12/04/2012)	3900	313	83	248	103235
SELinux CentOS 6.3 (targeted, policy.24 from 12/05/2012)	3508	277	81	235	275791
SELinux Debian 6.0.6 (default, policy.24 from 12/05/2012)	1285	190	77	229	49159

Table 2: *Overview of policy complexity: Different SELinux policies vs. SEAndroid and FlaskDroid*

has developed tools to verify the correct placement of the LSM hooks with a combination of runtime and static analysis and discovered inconsistencies in the hook placement. The work from [22] has been extended towards a static analysis based tool to automatically place authorization hooks in the LSM framework [29]. Specifically for mobile platforms, related work [86, 54] has investigated the placement of authorization hooks in the operating system and user-space services on the OpenMoko platform [54] and the LiMo (Linux Mobile) platform [86], both running SELinux, and our approach follows along the ideas of [86, 54] but for the Android middleware.

Empirical testing for FlaskDroid: We decided to evaluate the effectiveness of FlaskDroid based on empirical testing using the security models presented in Section 6 as well as a testbed of known malware retrieved from [1, 2] and synthetic attacks (cf. Table 3). Alternative approaches like static analysis [22] would benefit our evaluation but are out of scope of this paper and will be addressed separately in future work.

In Section 6, we presented use-cases and instantiations for how architecture can realize different security models to mitigate specific attacks and to enhance the system security.

To verify the effectiveness of FlaskDroid against known attacks on Android in practice, we used a testbed comprised of various widespread attacks in the wild as well as synthetic attacks (cf. Table 3). The malware samples were collected from the *Android Genome Project*[14] and the *Contagio Mobile Malware Mini Dump Website*[15].

7.2.1 Empirical testing at Kernel layer

Root Exploits.

Description. Root exploits on Android target privileged system **Services/Daemons** executed by the root user. The root user on stock Android is the highest-privileged user from the kernel perspective and inherits all privileges. Thus, he is able to read all files regardless of kernel-level permissions, perform privileged operations and has direct read and write access to all memory areas on the device.

Cases. Only few root exploits are known for Android 4.0. To test kernel-layer mitigation we used the well-known *mempodroid* exploit[16], which is a privilege escalation attack based on an insufficient permission check when writing to a processes memory via `/proc/pid/mem` from a setuid binary, in this case *run-as* tool.

Attack	Test
Root exploit	mempodroid Exploit
App executed by root	Synthetic Test App
Over-privileged apps and Information-stealing trojans	Known malware Synthetic Test App WhatsApp v2.8.4313 Facebook v1.9.1
Sensory malware	Synthetic Test App emulating [85, 14, 68]
Confused deputy attack	Synthetic Test App
Collusion attack	Synthetic Test Apps emulating [68]

Table 3: *List of attacks considered in our testbed*

Mitigation. SE Android successfully mitigates the effect of the *mempodroid* attack. While the exploit still succeeds in writing to `/proc/pid/mem` and hence elevating its process to root privileges, the process is constrained by the underlying SE Android policy to the limited privileges granted to the root user [74, 71, 73]. More details on the effectiveness of SE Android against other root exploits have been shown in [74, 73] and we focus in the remainder of this section on our extension to the middleware layer, which complement SE Android.

7.2.2 Empirical testing at Middleware layer

Over-privileged and nosy apps.

Description. Android's Permission framework is too coarse grained and inflexible to allow end-users to fine-grained configure to which private information an application has access. For example, the `READ_CONTACTS` permissions grants an app access to the entire contacts database, and there is no possibility to further restrict access to specific data, such as names, email addresses or phone numbers.

[14] `http://www.malgenomeproject.org/`
[15] `http://contagiominidump.blogspot.de/`
[16] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0056`

Cases. Designated information-stealing malware, such as Android.LoozPhone[17] and Android.Enesoluty[18], use the permissions granted to them by the user during installation to access and exfiltrate sensitive information, in this case contacts and device information (phone numbers, IMEI/IMSI number etc.)

Over-privileged apps, in contrast, are not malicious by design, but overstep the permissions required for their correct functionality. Older versions of the Facebook app, for example, requested the `READ_SMS` permission from the user without strictly requiring it. Another example is the WhatsApp messenger, which is granted access to the entire contacts database by requesting the `READ_CONTACTS` permission, although it only needs access to the contacts' phone numbers and names.

Mitigation. 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.LoozPhone[19] and Android.Enesoluty[20] which steal user's 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. Listing 12 shows an example policy which prevents untrusted apps (type `app_untrusted_t`) from accessing the fine-grained location information, but allows all apps of type `app_trusted_t` to access this information.

Listing 12: *Policy snippet showing access control for the system LocationManager.*

```
[...]

attribute allLocationData_t;

type fineLocation_t, allLocationData_t;
type coarseLocation_t, allLocationData_t;

allow {app_trusted_t} allLocationData_t:
      locationManager_t {requestLocationUpdates};
allow {app_untrusted_t} coarselocation_t:
      locationManager_t {requestLocationUpdates};

[...]
```

However, while blocking access to these system services did not cause application crashes, the applications often exhibited impeded functionality. For instance, prohibiting that the WhatsApp app can access the ContactsProvider and thus upload our contacts' phone numbers to its server, resulted in an empty WhatsApp contacts list and thus prevented us from messaging with our contacts. This effect concurs with the observations made in [40] and motivates a more fine-grained enforcement within ContentProviders to enable the user to configure policies to share data (e.g., contacts phone numbers and names) and simultaneously protect her sensitive data (e.g., all other contacts data like email addresses). Listing 13 presents an example on how such fine-grained access control in the system ContactsProvider can be implemented in our policy language.

Listing 13: *Policy snippet showing access control for the system ContactsProvider.*

```
[...]

attribute allContactsData_t;

type contacts_email_t, allContactsData_t;
type contacts_postal_t, allContactsData_t;
type contacts_name_t, allContactsData_t;
type contacts_number_t, allContactsData_t;

allow {app_trusted_t} allContactsData_t:
      contactsProvider_c {query insert delete update};
allow {app_whatsapp_t} {contacts_name_t,
      contacts_number_t}: contactsProvider_c {query};

[...]
```

Sensory malware.

Description. This class of malware uses the device's current usage context in combination with data obtained from onboard sensors, such as the acceleration sensor data or call state, to derive and exfiltrate sensitive information.

Cases. The TouchLogger [14] and TapLogger [85] attacks use information from the acceleration sensor to derive which keys the user has pressed on the on-screen keyboard. This information can be used to indirectly retrieve passwords the user has entered using the touchscreen.

Another example is the SoundComber Trojan [68], which extracts credit card information from recorded calls. By analyzing the current phone state using a `PhoneStateListener`, SoundComber gets notified about incoming and outgoing calls to the users bank. It records and analyzes the call to extract credit card information.

Mitigation. We emulate the behaviour of such malware using a custom synthesized test app. To mitigate the attack, we deployed a context-aware FlaskDroid policy (cf. 14) which causes the SensorManager USOM to filter acceleration sensor information delivered to registered SensorListeners while the on-screen keyboard is active, indicated when the `keyboard_b` boolean is set to `true` by a Context Provider dedicated to monitoring which application is shown in foreground on the screen.[21] While `keyboard_b` is `true`, access to sensor data exposed by the SensorManager USOM is denied to all apps. Similarly, a second policy prevents the SoundComber attack by denying any access to the audio record functionality implemented in the MediaRecorderClient USOM while a call is in progress, which is indicated by the `boolean telephony_b` that is set by a Context Provider monitoring the call state in the TelephonyManager Service.

Listing 14: *Policy snippet for context-aware access control on the SensorManager and MediaRecorder.*

[17] `http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99`

[18] `http://www.symantec.com/security_response/writeup.jsp?docid=2012-090607-0807-99`

[19] `http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99`

[20] `http://www.symantec.com/security_response/writeup.jsp?docid=2012-090607-0807-99`

[21] Technically this required an extension to the ActivityStack of the ActivityManagerService to provide this information to the Context Provider, since by default Android's application framework does not divulge this information.

```
[...]

bool keyboard__b = false;
bool telephony__b = false;

if(~keyboard__b)
{
    allow app__allApp__t allSensorData__t: sensorManager__c
          {update};
};

if(~telephony__b)
{
    allow app__allApp__t audio__t: mediaRecorderClient__c
          {record};
};

[...]
```

Malicious apps with root privileges..

Description. A particular use-case showing the effectiveness of our middleware extensions as a complement to SE Android are malicious apps executed with *root* privileges. By default, the permission check of vanilla Android would at runtime always grant a requested permission if the app runs with the root UID, regardless of whether or not this permission was granted by the user at installation time.

Cases. The user installs a malicious Android app which requests no permissions but launches a root exploit. If the attack is successful, the app acquires root privileges on the device. As described in Section 7.2.1, the root user will still be restricted by SE Android when accessing low level resources, such as the file system. However, even though the app did not request any permissions at installation time, it can use its omnipotent middleware privileges to steal sensitive information, such as contacts data or user location.

Mitigation. While SE Android inherently deprives root of its privileges at kernel level, we achieve the same security at middleware level. In FlaskDroid, the privileges of apps running with the root UID are restricted to the ones granted by the system policy to the root type (cf. *AID_ROOT* and `aid_root_t` in Section 5.3). During our tests, we had to define only one 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 *AID_ROOT* user (cf. Listing 15). Thus, a malicious app gaining root privileges despite SE Android, e.g., by employing the *RageAgainstTheCage*[22] or *mempodroid* (cf. Section 7.2.1) exploit [74, 73, 71], is in FlaskDroid restricted at both kernel and middleware level.

Listing 15: *Policy snippet for restricting the root user on the middleware layer.*

```
[...]

type aid__root__t, systemApps__a;

appType app__launcher__t
{
  Package:package__name=com.android.launcher;
};

appType app__provision__t
{
  Package:package__name=com.android.provision;
};

appType android__t
{
  /* All of packages under this UID */
  Package:package__name=android;
  Package:package__name=com.android.keychain;
  Package:package__name=com.android.settings;
  Package:package__name=com.android.seandroid__manager;
  Package:package__name=com.android.providers.settings;
  Package:package__name=com.android.systemui;
  Package:package__name=com.android.vpndialogs;
};

allow aid__root__t { app__launcher__t android__t
       app__provision__t }: activity__c {start};

[...]
```

[22]Also known as *adb setuid exhaustion attack*; Technical information at `http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/`

Confused Deputy and Collusion Attacks..

Description. A confused deputy on Android is an app which holds the necessary permissions to access sensitive data or Services, and exposes them to other apps using an unprotected interface without malicious intent. Collusion attacks are based on two (or more) inconspicuous apps which for themselves are not security-critical but display malicious behavior when working together.

Cases. Confused deputies have been shown to be a widespread problem among Android apps. For example, the Settings Widget of previous Android versions contained a confused deputy in a Broadcast Receiver component of the SettingsAppWidgetProvider class [63]. By sending a Broadcast Intent with a specific action string, a non-privileged app can enable or disable the GPS receiver and Wifi connections.

While Collusion attacks are currently mainly an academic topic, Android's system design has been shown to be be vulnerable to these attacks [68, 20, 51]. The previously mentioned SoundComber trojan [68] uses two apps which exchange data over a covert channel, for instance, the system audio volume settings. The first app has the `READ_PHONE_STATE` and `RECORD_AUDIO` permission. It records and analyzes phone calls and encodes sensitive information into volume settings. The second app possesses the `INTERNET` permission. It monitors volume setting changes, decodes the sensitive data, and sends it to a remote server.

Mitigation. The previously described confused deputy attack in the `SettingsAppWidgetProvider` class is addressed by our fine-grained access control rules on ICC. Listing 16 shows a policy that restricts the app types that may send (broadcast) Intents reserved for system apps to the type `android_t`. By limiting the allowed set of broadcast Intent senders and receivers, unprivileged apps are prevented from controlling the GPS and Wifi state.

Listing 16: *Policy snippet for restricting the root user on the middleware layer.*

```
[...]

appType android__t
{
  /* All of packages under this UID */
  Package:package__name=android;
  Package:package__name=com.android.keychain;
  Package:package_name=com.android.settings;
  Package:package_name=com.android.seandroid_manager;
  Package:package_name=com.android.providers.settings;
  Package:package_name=com.android.systemui;
  Package:package_name=com.android.vpndialogs;
};

intentType systemAppWidgetIntent_t
{
  Action:hasAction=
    "android.appwidget.action.APPWIDGET_UPDATE" |
    "android.appwidget.action.APPWIDGET_DISABLED" |
    "android.appwidget.action.APPWIDGET_ENABLED";
  Categories:hasCategory=
    "android.intent.category.ALTERNATIVE";
};

allow android_t systemAppWidgetIntent_t: broadcast_c
      {send sendSticky receiveSticky registerReceiver
      unregisterReceiver};

[...]
```

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 can be used that does not allow ICC between the colluding apps based on specifically assigned app types. However, to address collusion attacks *efficiently*, more flexible policies are required. We already discussed in Section 4.4.4 a possible approach to instantiate the XManDroid framework [11] based on our Context Providers and we elaborate in the subsequent Section 7.2.3 on particular challenges for improving the mitigation of collusion attacks.

7.2.3 Challenges and Trusted Computing Base

Information flows within applications: 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 unknown confused deputy or collusion attacks [51, 20, 68], which require more policy engineering to implement the rules presented in [11]. A particular challenge for addressing this problem and controlling access and separation (*non-interference*) of security relevant information are information flows within (untrusted) 3^{rd} party applications. Unless the application is a trusted USOM (like our *system* application USOMs, see Section 4.4.2), access control frameworks usually operate at the granularity of application inputs/outputs but do not cover the information flow within applications. For Android security, this control can be crucial when considering attacks such collusion attacks and confused deputy attacks (cf. Section 3.1). For SELinux based systems, related work has investigated approaches based on security typed programming languages to tackle this problem [39]. Specifically for Android, language based approaches have also been explored [17], but also taint tracking based approaches [23, 40] and extensions to Android's IPC mechanism [21]. To which extend these approaches could augment the coverage and hence effectiveness of FlaskDroid has to be explored in future work.

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 SecurityServer.

7.3 Performance Overhead

Middleware layer.

We evaluated the performance overhead of our architecture based on the no-allow-rule policy and the example policy presented in Section 7.1. Table 4 presents the mean execution time μ and standard deviation σ for performing a policy check at the middleware layer in both policy configurations (measured in μs) as well as the mean memory consumption (measured in MB) of the process in which our Policyserver executes (i.e., the system server). Average 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 8 presents the cumulative frequency distribution for our policy checks and shows that 99.33% of the policy checks with our example policy are performed in less than $2ms$. Thus, even if several policy checks have to be performed consecutively for executing an operation at the middleware (e.g., starting an Activity that lists the content of the contacts database), the accumulated overhead is magnitudes smaller than the human perceivable response delay of approximately $100 - 200ms$ and also than the default watchdog timers on Android (by default in the range of seconds).

	μ (in μs)	σ (in μs)	memory (in MB)
FlaskDroid			
No allow rule	329.505	780.563	15.673
Example policy	452.916	4887.24	16.184
Vanilla Android 4.0.4			
Permission check	330.800	8291.805	15.985

Table 4: *Runtime and memory overhead of FlaskDroid vs. vanilla Android*

	μ (in ms)	σ (in ms)
FlaskDroid (Example policy)	0.452	4.887
XManDroid [11] (Amortized)	0.532	2.150
TrustDroid [12]	0.170	1.910

Table 5: *Total performance comparison to closest related works*

In comparison to closest related work [11, 12] (cf. Section 8), FlaskDroid achieves a very similar performance. Table 5 provides an overview of the *total* performance overhead of the different solutions. *TrustDroid* [12] profits from the very static policies it enforces, while FlaskDroid slightly outperforms *XManDroid* [11]. 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

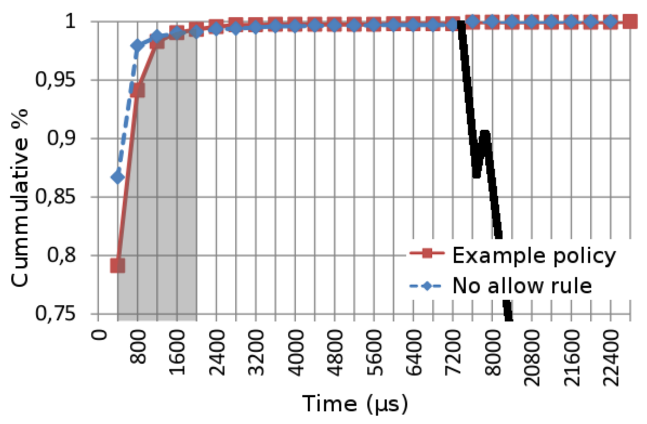

Figure 8: *Cumulative frequency distribution of the performance overhead with an example policy (solid line) and no policy rules (dashed lined). The grey shaded area represents the 99.33% confidence interval for the example policy.*

different baseline measurement. Both [11, 12] report on an baseline of approximately 0.18*ms* for the default permission check, which differs from the 0.33*ms* we observed in Android 4.0.4 (cf. Table 4).

Kernel layer.

The impact of SE Android on Android system performance has been evaluated previously by its developers [74, 73]. 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 [74, 73] still applies to our current implementation.

8. RELATED WORK

In this section we provide an overview of related work.

8.1 Mandatory Access Control

Flux Advanced Security Kernel (Flask) [78] is an architectural framework for Mandatory Access Control (MAC). It proposes a security architecture that decouples policy enforcement from the security policy itself thus providing for a generic architecture where multiple, dynamic security policies can be supported.

The most prominent instantiation of the Flask architecture is SELinux [56]. However, its biggest disadvantage as a kernel-level MAC is that its dynamic policies are limited to the context of the kernel-level. Thus, it requires a user space (i.e., middleware) agent to react to higher level context changes by triggering dynamic policy changes at kernel-level (cf. Section 2.4). Our middleware extensions perfectly implement such a user space agent on top of SE Android and simultaneously integrate the kernel-level MAC into a dynamic, consolidated two-layer MAC framework on Android.

Although SE Android [55] is the most sophisticated port of SELinux [56] for the Android platform and even has been partially merged into the official Android sources—a process that is expected to be continued—the applicability of SELinux and other MAC mechanisms for mobile devices has been discussed before [69, 86, 54, 80]. For instance, the authors of [69] elaborate on the benefits and challenges of porting SELinux to Android and evaluate a prototypical port. Similarly, the authors of [80] show its applicability for Maemo OS based mobile platforms. SELinux has been used to protect the Linux Mobile (LiMo) platform's integrity against malicious third party software [86] using a size-optimized policy, and [54] presents an SELinux-based OpenMoko platform in which the operating system and user-space services are instrumented as references monitor to MAC policy.

Also TOMOYO Linux [37], a path-based MAC framework, has been ported to Android and leveraged in Android security extensions [11][12] (see Section 8.3). Although TOMOYO supports more easily policy updates at runtime and does not require extended file system attributes, SELinux is more sophisticated and supports richer policies.[23] For instance, SELinux provides a much higher coverage of objects (e.g., files, capabilities, local IPC, or memory protection) and supports MLS and RBAC policies.

However, as we state in Section 3, low-level MAC alone is insufficient for isolating domains on multiple layers of the Android software stack. In this paper we show how SE Android can be leveraged for low-level MAC policy enforcement and preserving security invariants, while we extend the security architecture into the middleware layer for high-level MAC policy enforcement and consolidate the enforced policies in this two layers (cf. Section 6). We provided a more detailed comparison between the middleware enforcement in FlaskDroid and SE Android in the subsequent Section 8.2.

8.2 SE Android MMAC

As explained in Section 2.4, the SE Android project was recently extended by different mechanisms that add mandatory access control to certain subsystems of the Android middleware. Altogether these extensions are generally denoted as MMAC. In particular, MMAC currently consists of an install-time MAC, dynamic permission revocation, and Intent MAC. Thus, it is interesting how and in which aspects MMAC differs from our FlaskDroid middleware MAC.

Permission revocation.

MMAC provides a simple permission revocation mechanism. This is very similar to implementations found in old versions of popular after-market roms such as CyanogenMod[24]. Permissions are dynamically revoked by augmenting the default Android permission check with a policy driven check. This additional check overrules and negates the result of the permission check if the policy prohibits an application the usage of a particular permission.

However, this permission revocation is in almost all cases unexpected for application developers, which rely on the fact that if their app had been installed, it had 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 fact, this situation has prompted the developers of popular after-market roms like CyanogenMod to remove this mechanism from their feature set.

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 ap-

[23]E.g., `http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison`
[24]`http://www.cyanogenmod.com`

ply enforcement mechanisms that are *graceful*, i.e., they do not cause unexpected behavior that can cause application crashes. Prior work and related work on Android security mechanisms (cf. Section 8.3) introduced some of these graceful enforcement mechanisms, e.g., filtering responses from ContentProviders [13, 11, 12].

Intent MAC.

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 [90, 11, 12]. The white-listing is based on attributes of the Intent objects, for instance the value of the action string of the Intent or the category of the payload, and the security type assigned to the Intent sender and receiver app. Thus, if a particular Intent object matches a white-listing rule, MMAC allows the Intent to be delivered, otherwise the Intent delivery is canceled.

In FlaskDroid, we apply a very similar mechanism by assigning Intent objects a security type based on the Intent information and the security type of sender and receiver app. Based on the assigned type, policy rules are enforced which regulate the sending and receiving of Intents by applications. 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 lower-level 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 Android Interface Definition Language compiler (cf. Section 4.4) 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.

Similar to *Kirin* [25], MMAC performs a policy-driven install-time check of new applications and denies installation when the 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 applications with a security type with the install-time MAC logic, cf. Section 4.4.2). However, to provide a more generic enforcement as it is targeted by our design, the install-time MAC should also consider more meta-information of applications than merely their permissions. For instance, in FlaskDroid *appType*s (cf. Section 4.3) could be blacklisted. Thus, instead of being installed and labeled, apps which match blacklisted appTypes are denied installation in the first place.

8.3 Android Security Extensions

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

Different approaches [58, 57, 19, 59] add mandatory access control mechanisms to Android, which are tailored for specific problem sets such as providing a DRM mechanism (*Porscha*[58]), providing the user with the means to selectively choose the permissions and runtime constraints each app has (*APEX* [57] and *CRePE* [19]), or fine-grained, context-aware access control to enable developers to install policies to protect the interfaces of their applications (*Saint* [59]). Essentially all these solutions perform an access control at middleware layer of the form presented in Section 4.2 and the explicit design goal of our architecture was to provide an ecosystem that is flexible enough to instantiate those related work based on policies (as demonstrated in Section 6 at the example of Saint) and additionally providing the benefit of a consolidated kernel-level MAC.

The pioneering framework *TaintDroid* [23] introduced the tracking of the propagation of tainted data from sensible sources (in program variables, files, and IPC) and successfully detected unauthorized leakage of this data. The subsequent *AppFence* architecture [40] 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 (cf. Section 7.2.3). The general feasibility of such “context propagation” has been shown in the *MOSES* [67] architecture. In MOSES, applications and data are compartmentalized into different *security profiles* (e.g., `work` and `private`) and MOSES enforces isolation of these profiles. To this end, it applies labeling of data objects with their assigned profile and introduced policy enforcement points in Android middleware services and libraries (e.g., `LibBinder`, `Socket` class, or `OSFileSystem`) for fine-grained access control based on the labels. MOSES relies on the TaintDroid framework to propagate the labels of data object across process boundaries and thus addresses the problems discussed in Section 7.2.3.

To achieve policy enforcement for 3^{rd} party apps *without* the need to modify the Android operating system, some recent works leverage so-called *Inlined Reference Monitors* (IRM) [84, 8, 44]. IRM places the policy enforcement code directly in the 3^{rd} party app 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. This means that native code, which is by design supported in Android, can be maliciously used to access the IRM memory region and disable it at runtime.

The closest related literature to our work with respect to a two layer access control are the *XManDroid* [11] and *TrustDroid* [12] architectures. Both leverage TOMOYO Linux (cf. Section 8.1) as kernel-level MAC on an Android 2.2 to establish a separate security domain for business applications [12], or to mitigate collusion attacks via kernel-level resources [11]. Although they cover MAC enforcement at both the middleware and the kernel, both systems support only a very static policy tailored to their specific purposes. They do not support the instantiation of different use-cases, which requires a more flexible framework that supports a more generic policy language as we aimed for in FlaskDroid. In contrast, FlaskDroid can instantiate the XManDroid and TrustDroid security models by adjusting the policies. For instance, different security types for business and private applications could be assigned at installation time, or Boolean flags can be used to dynamically prevent two applications from communicating if the current system context/state would evaluate this communication as a collusion attack.

9. 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.

10. REFERENCES

[1] Android Malware Genome Project. `http://www.malgenomeproject.org/`.

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

[3] Facebook Caught Reading User SMS Messages? | TalkAndroid.com. `http://www.talkandroid.com/94623-facebook-caught-reading-user-sms-messages/`.

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

[5] Path uploads your entire iPhone address book to its servers. `http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html`.

[6] WhatsApp storing messages of users up to 30 days | Your Daily Mac. `http://www.yourdailymac.net/2012/02/whatsapp-storing-messages-of-users-up-to-30-days/`.

[7] WhatsApp took all my contacts and sent to their servers without asking me - BlackBerry Forums at CrackBerry.com. `http://forums.crackberry.com/blackberry-apps-f35/whatsapp-took-all-my-contacts-sent-their-servers-without-asking-me-649363/`.

[8] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard - real-time policy enforcement for third-party applications. Technical Report A/02/2012, Max Planck Institute for Software Systems, 2012.

[9] T. Bradley. DroidDream becomes Android market nightmare. `http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html`, 2011.

[10] P. Brady. Google I/O 2008: Anatomy & Physiology of an Android. `http://sites.google.com/site/io/anatomy--physiology-of-an-android`, 2008.

[11] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Network & Distributed System Security Symposium (NDSS'12)*, 2012.

[12] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

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

[14] L. Cai and H. Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *6th USENIX conference on Hot topics in security (HotSec'11)*. USENIX Association, 2011.

[15] V. Capretta, B. Stepien, A. Felty, and S. Matwin. Formal correctness of conflict detection for firewalls. In *ACM workshop on Formal methods in security engineering (FMSE'07)*. ACM, 2007.

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

[17] A. Chaudhuri. Language-based security on android. In *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*. ACM, 2009.

[18] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*, 2011.

[19] M. Conti, V. Thien N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC'10)*, 2010.

[20] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC'10)*, 2010.

[21] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smartphone operating systems. In *20th USENIX Security Symposium*. USENIX Association, 2011.

[22] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In *9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM, 2002.

[23] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[24] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, 2008.

[25] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM conference on Computer and communications security (CCS'09)*. ACM, 2009.

[26] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy Magazine*, 2009.

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

[28] 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`, January 2013.

[29] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In *12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM, 2005.

[30] P. Gilbert, B.-G. Chun, L. Cox, and J. Jung. Automating privacy testing of smartphone applications. Technical Report CS-2011-02, Duke University, 2011.

[31] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *2nd international workshop on Mobile cloud computing and services (MCS'11)*. ACM, 2011.

[32] Google. Android developers: Permissions – uri permissions. `http://developer.android.com/guide/topics/security/permissions.html#uri`.

[33] Google. Android security overview. `http://source.android.com/tech/security/index.html`, 2012.

[34] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *ACM conference on Wireless network security (WiSec'12)*. ACM, 2012.

[35] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, January 2005.

[36] G. Halfacree. Android trojan captures credit card details. `http://www.thinq.co.uk/2011/1/20/android-trojan-captures-credit-card-details/`, 2011.

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

[38] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, August 1976.

[39] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From Trusted to Secure: Building and executing applications that enforce systems security. In *USENIX Annual Technical Conference (USENIX ATC '07)*. USENIX Association, 2007.

[40] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM conference on Computer and communications security (CCS'11)*. ACM, 2011.

[41] H. Hu, G.-J. Ahn, and K. Kulkarni. Detecting and resolving firewall policy anomalies. *IEEE Trans. Dependable Secur. Comput.*, 9(3):318–331, May 2012.

[42] T. Jaeger. Managing access control complexity using metrics. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01)*. ACM, 2001.

[43] T. Jaeger, X. Zhang, and A. Edwards. Policy management using access control spaces. *ACM Transaction on Information and System Security*, 6(3):327–364, 2003.

[44] J. Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodified android. Technical report, University of Maryland, Department of Computer Science, 2012.

[45] K. Kohel. Security enhanced postgresql, 2007.

[46] K. Kostiainen, E. Reshetova, J.-E. Ekberg, and N. Asokan. Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In *First ACM Conference on Data and Application Security and Privacy (CODASPY'11)*. ACM, 2011.

[47] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974.

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

[49] Lookout Mobile Security. Security alert: Geinimi, sophisticated new Android Trojan found in wild. `http://blog.mylookout.com/2010/12/geinimi_trojan/`, 2010.

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

[51] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[52] F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[53] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *IEEE Symposium on Security and Privacy (SP'02)*. IEEE, 2002.

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

[55] National Security Agency. Security Enhanced Android. `http://selinuxproject.org/page/SEAndroid`.

[56] National Security Agency. Security-Enhanced Linux. `http://www.nsa.gov/research/selinux`.

[57] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, 2010.

[58] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in Android. In *26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[59] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference (ACSAC'09)*, 2009.

[60] Palm Source, Inc. Open Binder. Version 1. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`, 2005.

[61] S. Perez. Security hole spotted in facebook android sdk, long tail apps may still be unpatched. `http://techcrunch.com/2012/04/10/security-hole-spotted-in-facebook-android-sdk-long-tail-apps-may-still-be-unpatched/`, April 2012.

[62] A. Porter Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'11)*. ACM, 2011.

[63] A. Porter Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*. USENIX Association, 2011.

[64] N. Provos. Improving host security with system call policies. In *12th conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, 2003.

[65] V. Rao and T. Jaeger. Dynamic mandatory access control for multiple stakeholders. In *ACM symposium on Access control models and technologies (SACMAT'09)*, 2009.

[66] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, and K. Vaniea. 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'11)*. ACM, 2011.

[67] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: supporting operation modes on smartphones. In *17th ACM symposium on Access Control Models and Technologies (SACMAT'12)*. ACM, 2012.

[68] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[69] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy Magazine*, 2010.

[70] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Network and Distributed Systems Security Symposium (NDSS'06)*, 2006.

[71] S. Smalley. The case for SE Android. `http://www.selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf`.

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

[73] S. Smalley. Security Enhanced (SE) Android (Presentation), August 2012.

[74] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Network & Distributed System Security Symposium (NDSS'13)*, 2013.

[75] Stephen Smalley. Configuring the selinux policy. Technical Report #02-007, NAI Labs, 2003.

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

[77] B. T. Sniffen, D. R. Harris, and J. D. Ramsdell. Guided policy generation for application authors, 2006.

[78] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*. USENIX Association, 1999.

[79] TrendMicro. Android Under Siege: Popularity Comes at a Price, 2012.

[80] B. Vogel and B. Steinke. Using selinux security enforcement in linux-based embedded devices. In *International conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (MOBILWARE'08)*, 2007.

[81] E. Walsh. Selinux support for userspace object managers, 2004. Initial: May 2004, Last revised: May 2004.

[82] E. Walsh. Application of the flask architecture to the x window system server, 2007.

[83] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium*. USENIX Association, 2002.

[84] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *21st USENIX Security Symposium*. USENIX Association, 2012.

[85] Z. Xu, K. Bai, and S. Zhu. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *ACM conference on Wireless network security (WiSec'12)*. ACM, 2012.

[86] X. Zhang, J.-P. Seifert, and O. Acıiçmez. SEIP: simple and efficient integrity protection for open mobile platforms. In *International conference on Information and communications security (ICICS'10)*, 2010.

[87] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 2012.

[88] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Network & Distributed System Security Symposium (NDSS'13)*, 2013.

[89] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Network & Distributed System Security Symposium (NDSS'12)*, 2012.

[90] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *4th international conference on Trust and trustworthy computing (TRUST'11)*. Springer-Verlag, 2011.

APPENDIX

A. POLICY EXAMPLE FOR DEFINING MIDDLEWARE TYPES

Listing 17: *Example policy snippet showing the definition of attributes and types.*

```
/*
Example attributes for apps
*/
attribute allApps_a;
attribute systemApps_a;

/*
Example attributes for data (e.g., contacts data)
*/
attribute contactsData_a;

/*
Attribute for all sensor data types
*/
attribute allSensorData_t;

/*
Attribute for all location data types
*/

attribute allLocationData_t;

/*
Type definitions for apps
*/
type android_t, systemApps_a, allApps_a;
type app_system_t, systemApps_a, allApps_a;
type app_contacts_t, systemApps_a, allApps_a;
type app_telephony_t, systemApps_a, allApps_a;
type app_launcher_t, systemApps_a, allApps_a;
type app_tag_t, systemApps_a, allApps_a;
type app_backupconfirm_t, systemApps_a, allApps_a;
type app_bluetooth_t, systemApps_a, allApps_a;
type app_browser_t, systemApps_a, allApps_a;
[...]

/*
Some types for UIDs with no package as defined in
system/core/include/private/android_filesystem_config.h
which popup in middleware policy checks as subject or object
*/
type aid_radio_t, systemApps_a;
type aid_root_t, systemApps_a;
type aid_media_t, systemApps_a;

/*
Our own types
*/
type app_cased_t, allApps_a;
type untrustedApp_t, allApps_a;

/*
Type definitions for some contacts data used for fine−grained
access control to the contactsProvider_c
*/
type contacts_email_t, contactsData_a;
type contacts_name_t, contactsData_a;
type contacts_phone_t, contactsData_a;
type contacts_postal_t, contactsData_a;
type allContactsData_t;


/*
Type definitions for sensor data, unrolled based on the
hardcoded data rates (fastest, game, ui, normal) for each
sensor type
*/
type sensor_accelerometer_t_fastest, allSensorData_t;
type sensor_accelerometer_t_game, allSensorData_t;
type sensor_accelerometer_t_ui, allSensorData_t;
type sensor_accelerometer_t_normal, allSensorData_t;

type sensor_ambient_temperature_t_fastest,
       allSensorData_t;
type sensor_ambient_temperature_t_game, allSensorData_t;
type sensor_ambient_temperature_t_ui, allSensorData_t;
type sensor_ambient_temperature_t_normal,
       allSensorData_t;
[...]

/*
Type definitions for location data, unrolled based on the
hardcoded data resolution (fine, coarse)
*/
type fineLocation_t, allLocationData_t;
type coarseLocation_t, allLocationData_t;

/*
Some example intent types
*/

type untrustedIntent_t;
type intentLaunchHome_t;
```

B. POLICY EXAMPLE FOR DEFINING MIDDLEWARE CLASSES

Listing 18: *Example policy snippet illustrating the definition of object classes including inheritance between classes.*

```

class activity_c
{
    start stop grantURIPermission finish moveTask
};

class service_c
{
    start stop bind callFunction find
};

class intentService_c inherits service_c
{
    onHandleIntent
};

class clipBoardService_c
{
    getPrimaryClip
};

class broadcast_c
{
    send receive sendSticky receiveSticky registerReceiver
          unregisterReceiver
};

class intent_c
{
    send receive
};

class contentProvider_c
{
    query insert update delete readAccess writeAccess
};

```

```
class contactsProvider_c inherits contentProvider_c;
class calendarProvider_c inherits contentProvider_c;
class downloadProvider_c inherits contentProvider_c;
class mediaProvider_c inherits contentProvider_c;
class mmssmsProvider_c inherits contentProvider_c;
class smsProvider_c inherits contentProvider_c;
class telephonyProvider_c inherits contentProvider_c;
class settingsProvider_c inherits contentProvider_c;

class app_c
{
    clearAppUserData checkPermission switch
};

class package_c
{
    getPackageInfo getPackageInfoWithUninstalled
    getPackageUID getPackageGIDs getPackagesForUid
    getNameForUid getUidForSharedUser
    findPreferredActivity queryIntentActivities
    getInstalledApplications
    getInstalledApplicationsWithUninstalled
    getInstalledPackages
    getInstalledPackagesWithUninstalled
};

class locationService_c
{
    getAllProviders getProviders requestLocationUpdates
    removeUpdates addGpsStatusListener sendExtraCommand
    addProximityAlert removeProximityAlert getProviderInfo
    reportLocation isProviderEnabled getLastKnownLocation
    addTestProvider removeTestProvider
    setTestProviderLocation clearTestProviderLocation
    setTestProviderEnabled clearTestProviderEnabled
    setTestProviderStatus clearTestProviderStatus
};

class sensorService_c
{
  getSensorList getDefaultSensor unregisterListener
      registerListener
};
```