# Android Security Framework: Extensible Multi-Layered Access Control on Android

Michael Backes, Sven Bugiel, Sebastian Gerling, Philipp von Styp-Rekowsky
{backes,bugiel,sgerling,styp-rekowsky}@cs.uni-saarland.de

Saarland University/CISPA, Germany

## ABSTRACT

We introduce the *Android Security Framework (ASF)*, a generic, extensible security framework for Android that enables the development and integration of a wide spectrum of security models in form of code-based security modules. The design of ASF reflects lessons learned from the literature on established security frameworks (such as Linux Security Modules or the BSD MAC Framework) and intertwines them with the particular requirements and challenges from the design of Android's software stack. ASF provides a novel security API that supports authors of Android security extensions in developing their modules. This overcomes the current unsatisfactory situation to provide security solutions as separate patches to the Android software stack or to embed them into Android's mainline codebase. This system security extensibility is of particular benefit for enterprise or government solutions that require deployment of advanced security models, not supported by vanilla Android. We present a prototypical implementation of ASF and demonstrate its effectiveness and efficiency by modularizing different security models from related work, such as dynamic permissions, inlined reference monitoring, and type enforcement.

## 1. INTRODUCTION

For several decades now, the need for operating system security mechanisms to provide strong security and privacy guarantees has been well understood [24, 34, 26, 5]. Yet, recent classes of attacks against smartphone end-user's privacy and security [19, 41, 29, 9] have shown that the fairly new smart device operating systems fail to provide these strong guarantees, for instance, with respect to access control or information flow control. To remedy this situation, security research has proposed a wide spectrum of security models and extensions for mobile operating systems, most of them for the popular open-source Android OS. These extensions include context-related access control [10], developer-centric security policies [28], and dynamic, fine-grained permissions [42, 21, 3]. They also comprise security models [7, 33, 36, 8] such as domain isolation and type enforcement, which are usually at the heart of enterprise and governmental security solutions.

However, the lack of a comprehensive security API for the development and modularization of security extensions on Android has created the unsatisfactory situation that all of these novel and warranted security models are either provided as model-specific patches to the Android software stack, or they became an integrated component of the Android OS design [36]. When considering the body of literature on established security frameworks, such as *Linux Security Modules* (LSM) [40] or the *BSD MAC Framework* [39], their history has taught that the need to patch the OS or the hardwiring of a specific security model impairs both the practical and theoretical benefits of security solutions. First, there is in general no consensus on the *"right"* security model, as demonstrated by the broad range of Android security extensions [10, 28, 3, 42, 7, 36]. Thus, OS security mechanisms should not limit policy authors to one specific security model by embedding it into the OS design. Second, providing security solutions as *"security-model-specific Android forks"* impedes their maintainability across different OS versions, because every update to the Android software stack has to be re-evaluated for and applied to each fork separately.

**Contributions.** In this paper, we propose the design and implementation of ANDROID SECURITY FRAMEWORK (ASF), which allows security experts to develop and deploy their security models in form of modules as part of Android's platform security. This provides the means to easily extend the Android security mechanisms and avoids that security designers have to choose "the right Android security fork" or that the OS vendor has to impose a specific security model. In the design of ASF we transfer the lessons learned and guiding principles from the literature on established OS security infrastructures to Android and intertwine them with new requirements for efficient security policies for multi-tiered software stacks of smart devices. In contrast to concurrent, independent work [20], which introduced extensibility for security *apps* (i.e., add-ons), our design establishes a generic and extensible security framework that allows instantiating security models *by design* as part of Android's platform security and enables not only extending but also replacing Android's default security mechanisms. This is particularly beneficial when tailoring Android for higher-security deployments like enterprise phones, where the default mechanisms are insufficient or even obsolete (e.g., when the IT department is an additional stakeholder that decides on apps' privileges and installation). We make the following contributions:

*ACSAC'14,* December 08–12, 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-6/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664265.

*1. Policy-agnostic, multi-tiered security infrastructure:* The security infrastructure must avoid committing to one particular security model and enable authors of security extensions to develop as well as deploy their solutions in form of code. This requires special consideration of Android's multi-tiered software stack and the dominant programming languages at each layer. For ASF we solve this by integrating security-model-agnostic enforcement hooks into the Android kernel, middleware and application layer and exposing these hooks through a novel security API to module authors.

*2. Enabling edit automata policies:* Various Android security solutions realize edit automata policies that not only truncate but also modify control flows. In ASF, the application layer and middleware hooks are specifically designed to allow module authors to leverage the rich semantics of Android's application framework and to implement their security policies as edit automata. This required a re-thinking of the "classical" object manager design from the literature by shifting the edit automata logic from the infrastructure into the security modules.

*3. Instantiation of existing security models:* We demonstrate the efficiency and effectiveness of our ASF by instantiating different security models from related work on type enforcement [8, 36] and inlined access control [3] as well as from Android's default security architecture as modules.

*4. Maintenance benefits for security extensions:* Our ported security modules show how ASF simplifies maintainability of security extensions across different OS versions by shifting the bulk of effort to the security framework maintainer. This is similar to the maintenance of the application framework for regular apps. Hence, a comparable benefit to regular apps in adaption and stability across OS versions can be expected of security modules.

*5. Research and development benefits:* We postulate that developing security solutions against a well documented security API also greatly contributes to *a)* a better understanding and analysis of new security models that form a self-contained unit instead of being integrated to various components of the Android software stack, *b)* a better reproducibility and dissemination of new solutions since modules can be easily shared and instantiated, and *c)* a more convient application of security knowledge to the Android software stack without the requirement to be familiar with the deep technical internals of Android.

## 2. BACKGROUND ON ANDROID

In this section we provide necessary technical background information on Android.

### 2.1 Primer on Android

Android is an open-source software stack for embedded devices. The lowest level of this stack consists of a Linux kernel responsible for elementary services such as memory management, device drivers, and an Android-specific lightweight inter-process communication called Binder. On top of the kernel lies the extensive Android middleware, consisting of native libraries (e.g., SSL) and the application framework. System services in the middleware implement the bulk of Android's application API (e.g., the location service) and pre-installed system apps at the application layer, like Contacts, complement this API.

Although application layer and middleware apps and services are commonly written as Java code, they are compiled to *dex* bytecode and run inside the *Dalvik Virtual Machine* (DVM). In addition to dex bytecode, apps and services can use native code libraries (i.e., C/C++) for low-level interactions with the underlying Linux system. Native code can be seamlessly integrated into dex bytecode by means of the *Java Native Interface*.

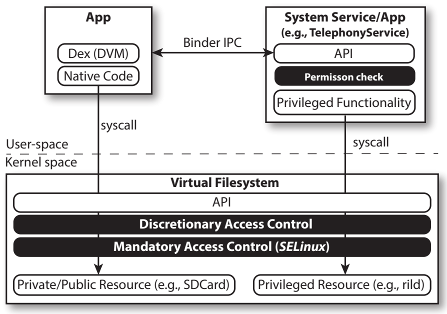

Figure 1: Android's default security architecture.

Android apps are generally composed of different components. The four basic app components are *Activities* (GUI for user interaction), *BroadcastReceivers* (mailbox for broadcast *Intent* messages), *ContentProviders* (SQL-like data management), and *Services* (long running operations without user interaction). All components can be interconnected remotely across application boundaries by using different abstractions of Android's Binder IPC mechanism, such as *Intent* messages.

### 2.2 Android's Security Philosophy

Android's security philosophy dictates that all apps are sandboxed by executing them in separate processes with distinct user IDs (*UID*) and assigning them private data directories on the filesystem.

To achieve privilege separation between apps, Android introduces *Permissions*, i.e., privileges that an app is granted by the user at install-time. In accordance with the least privilege principle, an app without permissions is not able to access security and privacy sensitive resources. Permissions are assigned to the app's UID and enforced at two different points in the system architecture (cf. Figure 1): First, every app sandbox can directly interact with the kernel through system calls, for instance, to edit files or open a network socket. These resources are either of private nature (i.e., are within the app's private directory) or public resources (e.g., SDCard). Access control in the filesystem ensures that the apps' processes have the necessary rights (i.e., Permissions) to issue particular syscalls, e.g., opening a file. The filesystem access control consists of the traditional Linux Discretionary Access Control, which is complemented (since Android v4.3) by SELinux based Mandatory Access Control (MAC).

Second, apps can interact through the Android API in a strictly controlled manner with highly privileged resources. To ensure system security and stability, apps are prohibited to access those resources directly. Instead, those resources are wrapped by system services and apps that implement the API. For instance, the *TelephonyService* communicates on behalf of apps with the radio interface layer daemon (*rild*) to initiate calls or send text messages. Whether an app is sufficiently privileged to successfully call the API is determined by a Permission check within the system services/apps. For this check, the Binder mechanism provides to the callee (system service/app) the UID of the caller (app).

## 3. RELATED WORK

We first provide a synopsis of the development of extensible kernel security frameworks and discuss afterwards the current status of security extensions and frameworks for Android.

### 3.1 Extensible Kernel Access Control

The importance of the operating system in providing system security has been very well studied in the last decades [34, 24, 5, 26] and different approaches to extending operating systems with access control and security policies have been explored. These include system-call interposition [15, 30], software wrappers [16], and extensible access control frameworks like *DTE* [4], *GFAC* [1], and *Flask* [37]. All of these solutions have been provided as kernel patches for Linux or UNIX. However, this led to an intricate situation: On the one hand, maintaining these solutions as patches incurred high maintenance costs for adapting the patches to kernel changes. On the other hand, none of these solutions was included in the vanilla kernel because this would constrain security policy authors to one specific security model. This constrain would be unsatisfying since there exists in general no consensus on the "right" security model. To remedy this situation, extensible security frameworks have been proposed [40, 39] that allow the extension of the system with trusted code modules that implement specific security models. Module authors are supported with an API that exposes kernel abstractions as well as operations and facilitates the implementation of the desired security architecture and model. The results of this research have been integrated into the mainline kernels as the *Linux Security Modules* framework (LSM) [40] and the *BSD MAC Framework* [39].

### 3.2 Android Security

Closest to our approach is the independently and concurrently developed *ASM* [20], which also provides a programmable interface for security extensions. In contrast to ASF, however, it targets "security apps" added in addition to the default Android security architecture. As a consequence, ASM has to address the intricate problem of including untrusted code into highly-privileged context for access control enforcement and consolidating it with existing policies. It avoids this Gordian knot through a trade-off between policy expressiveness and sandboxing of security apps. In contrast, our ASF framework resides beneath the default Android security framework and hence allows instantiation of security models that complement or even substitute parts of the default platform security (cf. Section 6). Hence, ASM can even be implemented as a module in ASF. By definition, we must trust the developer of security solutions for ASF.

In recent years, Android's security has been quite scrutinized, and a wide spectrum of security extensions has been brought forward. To name a few: *CRePE* [10] provides a context-related access control, where the context can be, e.g., the device's location. *Saint* [28] enables developer-centric policies that allow app developers to ship their apps with rules that regulate the app's interactions with other apps. Different approaches to more dynamic and fine-grained permissions have been proposed based on system-centric enforcement (e.g., *TISSA* [42]) or inlined reference monitors (*Dr. Android and Mr. Hide* [21] or *AppGuard* [3]). *XManDroid* [6] enforces Chinese Wall policies to prevent confused deputy and collusion attacks. *TrustDroid* [7] and *MOSES* [33] isolate different domains such as "Work" and "Private" from each other. *SE Android* [36] and *FlaskDroid* [8] bring type enforcement to Android, where SE Android focuses on the kernel layer and has been partially included into the mainline Android source code, and FlaskDroid extends type enforcement to Android's middleware layer on top of SE Android.

## 4. REQUIREMENTS ANALYSIS

The current development of Android security extensions has strong parallels to the initial development of the above mentioned Linux and BSD security extensions, since current Android security extensions are provided as patches to the software stack or, in the case of SE Android [36], are embedded into the Android source tree. For the same, above mentioned reasons as for the early Linux and BSD security extensions, this impedes the applicability and adaption of Android security extensions and additionally precludes many of the benefits that a modular composition could offer in terms of maintenance: Embedding SE Android's security model into Android's source tree limits policy authors to the expressiveness and boundaries of type enforcement, whereas provisioning security models and architectures as patches to Android's software stack forces policy authors to chose a solution-specific Android fork. This requires for every version update to the Android OS a re-evaluation and port of each separate fork. Moreover, security solutions cannot be easily compared with each other, because their infrastructures are deeply embedded into the Android software stack.

In this paper, we develop in the spirit of the two de facto most established security frameworks, *Linux Security Modules* (LSM) [40] and the *BSD MAC Framework* [39], a generic and extensible ANDROID SECURITY FRAMEWORK that allows the instantiation and deployment of different security models as modules at Android's application layer, middleware, and kernel. The two most important guiding principles from LSM and the BSD MAC framework that govern the design of our ANDROID SECURITY FRAMEWORK are: *1)* provisioning of policies as code instead of data; and *2)* providing a policy-agnostic OS security infrastructure. In the remainder of this section, we analyze the requirements and challenges for their transfer to the Android software stack.

**Policy as code and not data.** The first guiding principle is that policies should be supported as code instead of data (such as rules written in one predetermined policy language). Providing an extensible security framework that supports integration of policy logic as code avoids committing to one particular security model or architecture. For Android, this removes the need to chose a particular extension-specific Android fork or to be limited to one specific security model in the mainline Android software stack. Additionally, developing modules against an OS security API provides the benefits of modularization for developing and maintaining security extensions. This includes, foremost, a higher functional cohesion of security modules and lower coupling with the Android software stack and, hence, can significantly reduce the maintenance overhead of modules, especially in case of OS changes. Moreover, it allows a better dissemination, comparison, and analysis of self-contained security modules.

Transferring this principle to an extensible security framework for Android poses the additional requirement to consider the semantics and dominant programming languages of the

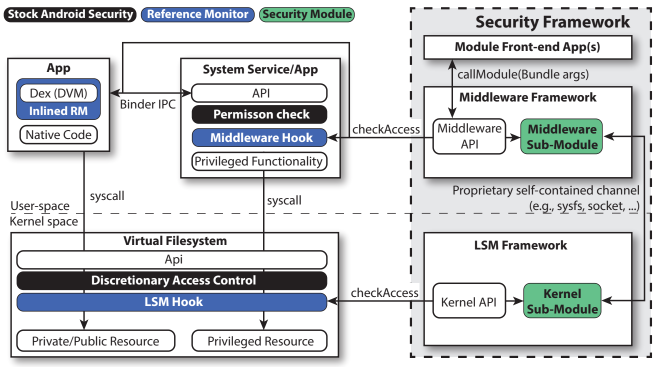

Figure 2: Android Security Framework architecture.

different layers of Android's software stack. LSM and the BSD MAC Framework, for instance, as part of the kernel, support modules written in C and operate on kernel data structures (e.g., filesystem inodes). While this applies to the Android Linux kernel as well, an Android security framework should additionally support modules written for Android's semantically-rich middleware and application layers. That means modules written in Java and operating on application framework classes (e.g., Intents or app components).

**Policy-agnostic security infrastructure.** The second principle is that the security framework and its API should be policy-agnostic. This means that policy-specific intrusions into the software stack are avoided and policy-specific data structures and logic are confined to security modules.

A particular additional requirement for a security framework on Android are enforcement hooks in the middleware and application layer that support *edit automata* [23] policies, as promoted by different solutions [42, 7, 21, 3]. Edit automata, in contrast to truncation automata, can not only abort control flows but also divert or manipulate them and, thus, give policy authors a higher degree of freedom in implementing their enforcement strategies. For instance, when querying a ContentProvider component, the policy could simply deny access by throwing a Java Exception (truncation), but also modify the return value to return filtered, empty, or fake data (edit). To technically enable security modules to implement edit automata, our design requires a re-thinking of the "classical" object manager vs. policy server design that is used, e.g., in LSM. Object managers (i.e., enforcement points) are responsible for assigning security labels to the objects that they manage and for both requesting and enforcing access control decisions from the policy server (i.e., policy decision point). Because this design embeds the enforcement logic into the system independently from the security model, it is unfit for realizing edit automata. Thus, our design requires hooks that generically support different enforcement strategies and shift the enforcement and object labelling logic from the object managers to the security modules.

## 5. ANDROID SECURITY FRAMEWORK

In the following we present ASF. We provide further details in an anonymous long version of this paper [2].

### 5.1 Framework Overview

The basic idea behind our Android Security Framework is to extend Android with a new security API that incorporates the design principles explained in Section 4. This API allows to easily author, integrate, and enforce generic security policies. Figure 2 provides an overview of our ASF and we explain its building blocks in the following.

#### 5.1.1 *Reference Monitors*

In our design we differentiate between policy enforcing and policy decision making code. For enforcement we use reference monitors [22] at all layers of the Android software stack, i.e., at the application layer, the middleware layer, and the kernel layer. Each reference monitor protects one specific privileged resource and is placed such, that it mediates all access to the resource through the Android API. The benefit of this multi-tiered enforcement is that each reference monitor can operate with the semantics of its respective layer.

#### 5.1.2 *Security Modules*

Security extensions are deployed in the form of code modules and loaded during boot into the security frameworks at the middleware and kernel level. Modules should be signed to ensure their integrity and trustworthiness, and the verification key is embedded in the kernel (or a secure location like a secure execution environment). Each module implements a policy engine that manages its own security policies and acts as policy decision making point. Security modules are integrated into the security frameworks through a security API that exposes objects and operations of the different software stack layers.

To provide a clear separation between policy decision logic using kernel level semantics and logic using middleware/application layer semantics, each module consists of two sub-modules: a Kernel Sub-Module leveraging the already existing Linux Security Module (LSM) infrastructure of the Linux kernel and a Middleware Sub-Module, for which we designed and implemented a novel security infrastructure at the application and middleware layers.

#### 5.1.3 *Front-end Apps*

To enable user configurable policies or graphical event notifications, modules might want to include user interfaces. To this end, the module developers (or external parties being aware of the modules) can deploy standard Android apps that act as front-end and that communicate through the framework API with the module. We enable such proprietary module interfaces through a *Bundle* based communication protocol. A Bundle is a key-value store that supports heterogenous value types (e.g., Integer and String) and that can be transmitted via Binder IPC. It is the responsibility of the module to verify that the caller is sufficiently privileged.

### 5.2 Framework Infrastructure

We present now in a bottom-up approach details about the ASF infrastructure that has been prototypcially implemented for Android v4.3 and currently comprises 4606 lines of code.

#### 5.2.1 *Kernel Space*

At kernel level we employ the existing Linux Security Module (LSM) [40] framework of the Linux kernel. LSM implements an infrastructure for mandatory access control and provides a number of enforcement hooks within kernel components such as the process management or the virtual filesystem. The Kernel Sub-Module is implemented as a standard Linux Security Module that registers through

Listing 1: Exemplary enforcement functions

```
1 public boolean deliverToRegisteredReceiver (Intent intent,
        ComponentName targetComp, String requiredPermission,
        int targetUid, int targetPid, String callerPackage,
        ApplicationInfo callerApp, int callingUid, int callingPid);
2 public Location getLastLocation (Location currentLocation,
        LocationRequest request, int callingUid, int calingPid);
```

the LSM API for the LSM hooks in the system and that operates with kernel level semantics. Kernel Sub-Module can be an existing Linux security module like SELinux or proprietary ones [20]. Kernel-level policies form truncation automata that terminate illegal control flows.

Since there might be operational inter-dependencies between the Kernel Sub-Module and user-space processes like the Middleware Sub-Module (e.g., propagation of access control decisions), the kernel module can implement proprietary channels for communication between kernel- and user-space (e.g., sysfs entries).

#### 5.2.2 Middleware Layer

At the middleware layer we extended the system services and apps that implement the Android API with hooks that enforce access control decisions made by the Middleware Sub-Module. The middleware security framework is executed as a new Android system service and mediates between our hooks and the Middleware Sub-Module. The hooks are policy-agnostic and not tailored to one specific security model. Each hook takes as arguments all relevant, ambient information of the current control flow that led to the hook's invocation. For instance, Listing 1 presents two exemplary hooks in our system: one for the Intent broadcasting subsystem of the *ActivityManagerService* (line 1) and one for the *LocationManagerService* that implements the location API of Android (line 2). Both provide to the Middleware Sub-Module information about the current caller to the Android API, i.e., App in Figure 2 (parameters `callingUid` and `callingPid`). However, all other parameters are specific to the hooks' contexts, e.g., the hook in line 1 provides information about the Intent being broadcast and the app component that should receive this Intent (parameters `targetComp` through `targetPid`). Thus, the hooks support policies that use the rich middleware-specific semantics.

In general, all hooks support truncation automata as policies by either allowing the module to throw exceptions that terminate the control flow and that are returned to the caller of the Android API, or by explicitly requiring a boolean return value that indicates whether the hook truncates the control flow or not (line 1 in Listing 1). A subset of the hooks additionally supports edit automata policies, that is the module can modify or replace return values of the Android API function or modify/replace arguments that divert or affect the further control flow after the hook. For instance, the *LocationManagerService* hook in Listing 1 (line 2) allows the module to edit or replace the Location object that is returned to the app that requested the current device location.

#### 5.2.3 Application Layer

At the application layer, our Android Security Framework provides a mechanism to inject access control hooks into apps themselves. This access control technique is based on the concept of *inlined reference monitors* (IRM) pioneered by Erlingsson and Schneider [14]. The basic idea is to rewrite an untrusted app such that the reference monitor is directly embedded into the app itself, yielding a "self-monitoring" app. Although using IRMs might seem counter-intuitive or redundant in our design, IRMs are the only way in Android's current app model to achieve privilege separation between the components within an app (e.g., ad libs [19]) or to enforce edit automata policies on file-system and network interfaces (e.g., HTTPS-everywhere). The former are DVM internal operations and the latter do not involve the middleware, but instead the app processes interact directly with the file-system and network API of the kernel, whose semantics are rather unsuitable for enforcing edit automata policies. Thus, until this app model has been retrofitted to enable a system-centric solution for such kind of policies, our design relies on IRM. ASF provides an instrumentation API that enables security modules to dynamically hook any Java function within an app's DVM. Hooked functions divert the control flow of the program to the reference monitor, which thereby not only gains access to all function arguments but can also modify or replace the function's return value. Furthermore, in contrast to the hooks placed in the Android middleware, application layer hooks are dynamic: Hooks are injected by directly modifying the target app's DVM memory when a new app process is started. This design enables security modules to dynamically create and remove hooks at runtime as well as to inject app-specific hooks.

### 5.3 Middleware Framework API

We elaborate now in more detail on our framework API and the interaction between modules and the security infrastructure. Since we use the existing LSM framework as is, we focus here on our newly introduced middleware security framework and refer to the kernel documentation [25] for details on the LSM API. The middleware framework API of our current implementation contains 168 callback functions; a full listing is provided in the long version of this paper [2]. This API can be broken down into the following categories:

**Enforcement functions.** These functions form the bulk of the API (136 methods) and are called by the framework whenever the enforcement hooks in system apps and services are triggered. Each hook has a corresponding callback function in the module API, which has the same method signature as the hook (cf. Listing 1) and which implements the policy decision logic for its hook.

**Kernel Sub-Module Interface.** To avoid policy-specific interfaces for the communication between middleware/application layer apps and the Kernel Sub-Module, we introduce a generic kernel module API as part of the middleware framework API. It allows apps and services a controlled access to Linux security modules. Each security module can implement this interface and internally translate the API calls to calls on the proprietary channel between the user-space and the Linux security module. Two particular challenges for establishing this interface were the self-contained security checks of the kernel module and the requirement that this interface is already available during system boot. To guarantee security, the kernel module is required to perform policy checks to verify that a user-space process is sufficiently privileged to issue commands to it. Additionally, the kernel module is called before the middleware framework can load any Middleware Sub-Module, e.g., it can be called by Zygote when spawning new app processes. To solve these challenges, our design avoids an additional layer of indirection (i.e., IPC) for communication with the kernel module and loads the interface implementations via the Java reflection API statically into the application framework when it is bootstrapped. This ensures that the calling processes communicate directly with the kernel module through our generic API and that the kernel module can be called independently of middleware services.

**Life-cycle management.** Every module must implement functions for life-cycle management, such as initialization or shutdown. This enables the framework to inform the module when the system has reached a state during the boot cycle from which on the module will be called or when the system shuts down. Modules should use these functions, e.g., to initiate their policy engines or to save internal states to persistent storage before the device turns off.

**Event notifications.** Event notification interfaces are used to propagate important system events to the module. For instance, modules should be immediately informed when an app was successfully installed, replaced, or removed. Although this information is usually propagated via a broadcast Intents, the time gap between package change and broadcast delivery might cause inconsistencies in module states. Hence these events must be delivered synchronously.

**Framework Callbacks.** The framework provides modules a callback interface for communicating in a more direct manner with system services, such as the *PackageManagerService*, and avoids the need to go through the Android API. This is desirable for policy authors that want to leverage the middleware internal information. Our current callback interface, for instance, includes functions that allow modules to efficiently resolve PIDs to application package names.

**Proprietary protocols.** We introduced in our framework API a *callModule()* function that allows modules to implement proprietary communication protocols with other apps that are aware of this specific module, e.g., the front-end apps (cf. Section 5.1). When using *callModule()*, these protocols are based on *Bundles* and enable a protocol similar to the Parcel-based Binder IPC: apps serialize function arguments to a Bundle and add an identifier for the proprietary function the module should execute with the deserialized arguments. It is the task of the module to verify that the sender is sufficiently privileged to send commands.

**IRM Instrumentation.** The framework provides an instrumentation API that enables security modules to hook any Java function within selected app processes. To the best of our knowledge, ASF is the first solution for Android to provide a generic instrumentation API. Hooks injected via the instrumentation API are local to the app process that the API is called from. Therefore, all calls to the instrumentation API need to be performed from within a target app's process. We solve this by placing an instrumentation hook in the *ActivityManagerService* that is triggered when a new app process is about to be launched. A module that implements this hook has to return a Java class for the instrumentation logic that will be executed within the app's process. To ensure that this code is executed before control flow is passed to the app itself, we modify the arguments passed to Zygote to start this new app process via a special wrapper class that loads and executes the instrumentation code first.

## 5.4 Middleware Security Modules

We elaborate in more detail on the structure of security modules. Again, we use Linux security modules as is [25] and, thus, focus here on the Middleware Sub-Module. A middleware module is simply a *Jar* file that is created with an Android SDK that includes our new security API. It is deployed to a protected location on the file system, from where it is loaded during boot. This Jar file contains all the module's code, resources, and manifest file (cf. Figure 3):

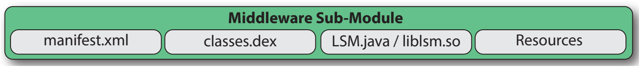

Figure 3: Middleware security module structure.

**Module Manifest.** The manifest (formatted in XML) declares properties such as the module author or code version, and, more importantly, the name of the main Java class that forms the entry point for the module.

**Classes.dex.** The *classes.dex* file contains, as in regular Android apps, the Java code compiled to *Dalvik executable bytecode* (DEX). It contains all Java classes that implement the security module's logic. During the load process of the Middleware Sub-Module, the middleware framework uses the Java reflection API to load the module's main class (as specified in the manifest) from *classes.dex*. To ensure that the reflection works error-free, the main class must implement the API as described in Section 5.3. Since the API defines currently more than a hundred methods, but a security module very likely requires only a subset of those, our SDK provides an abstract class that implements the API. That abstract class can be sub-classed by the module's main class, which then only needs to override the required functions. The abstract class returns for each non-overridden enforcement function an allow decision.

**LSM interface.** The proprietary interface between the user-space processes and the Linux security module in the kernel is implemented through a native library *liblsm.so* and a corresponding Java class *LSM.java*, which exposes the native library via the Java Native Interface. *LSM.java* has to implement the generic interface for the communication with the kernel that was explained in the previous section. The generic kernel module interface of ASF loads *LSM.java* through the Java reflection API into Android's application framework. This allows apps and services to communicate with the kernel module and avoids a policy-specific interface. We exemplified this mechanism by integrating SELinux through API into Zygote (cf. Section 6.2).

**Resources.** Each module can ship with proprietary resources, such as initial configuration files or required binaries. During module instantiation, the framework informs the module about the filesystem location of its Jar file, enabling the module to extract these resources on-demand from it.

## 5.5 Stackable and Dynamic Loadable Modules

Finally, two desirable properties for implementing an extensible security framework such as our ASF are dynamically loadable policies and policy composition (i.e., stacking modules). In the following we explain why we chose, in contrast to closest related work [20], to *permit* these features by design, but *not* consider them a requirement for our solution.

**Dynamically Loadable Modules.** Being able to dynamically load and unload modules is desirable, for instance, to speed up the development and testing cycles of modules and, in fact, we used this feature during the development of our example use-cases (cf. Section 6). However, the arguments to support dynamically loadable modules beyond development (e.g., for security add-ons [20]) are disputed: First, dynamic loading is not always technically possible. A small set of static policy models, such as type enforcement [36, 8], require that all subjects and objects are labeled with a security context. Supporting such extensive labeling operations at runtime is an intricate problem. Second, there exist security considerations. The loading and unloading of modules must be strictly controlled to ensure that only integrity protected, trusted modules are loaded. Otherwise, given the privileges of modules, this would open the way to powerful malware modules. In our design we agree with the conclusions of the various Linux security module authors [11] and consider the drawbacks of dynamically loadable modules to outweigh their benefits. Therefore, we load the module *once during the system boot* and *permit* users of our framework to additionally activate dynamic unloading and loading of modules. But we currently do not consider this feature a requirement for our solution.

**Stackable Modules.** Composing the overall policy from multiple, simultaneously loaded and independent policies is a desirable feature, since usually no "one-size-fits-all" policy exists. Android, for instance, implements currently a quadruple-policy approach consisting of Permissions, SE Android type enforcement, AppOps, and Linux capabilities—each being responsible for a different aspect of the overall access control strategy. Multiple policies will naturally conflict and thus require the security framework to support different policy composition and reconciliation strategies (e.g., consensus or priority based) [32, 27]. However, supporting fully generic policy composition is quite a challenge and has been shown to be intractable [18]. Thus, despite its benefits, we decided in our design to follow the lessons learned by the LSM developers [40] and to only *permit* module developers to implement stackable modules, but we do not provide explicit interfaces for stacked modules in our framework infrastructure. In module combinations where policy consolidation is known to be feasible, the approach to stacking modules would be to provide a "composition module" that implements policy reconciliation and composition logic and which in turn can load other modules and multiplex API calls between them.

## 6. EXAMPLE SECURITY MODULES

In this section, we demonstrate the efficiency and effectiveness of our ANDROID SECURITY FRAMEWORK by instantiating different security models from related work. To illustrate the versatility of ASF, we chose models from the areas of inlined reference monitoring, Android's default security architecture, and type enforcement. We present further instantiations of other security models in our long version [2].

### 6.1 AppOps and IntentFirewall

Google introduced (unofficially) with Android v4.3 the *AppOps* infrastructure for dynamic, more fine-grained Permissions. It added hooks in different system services and apps, which query a central AppOpsService whether an application is allowed to perform an operation (e.g., retrieving the location of the device or querying a ContentProvider). The AppOps rules define a mapping from UID/package name to allowed operations. AppOps offers an interface to apps to retrieve the current configuration. Additionally, Google introduced (again unofficially) an *IntentFirewall*, which acts as a reference monitor for certain Intent-based operations

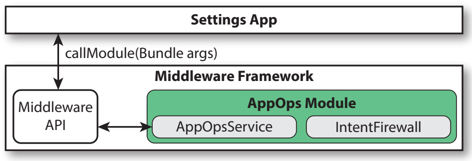

Figure 4: AppOps and IntentFirewall module

like starting an Activity. The IntentFirewall rules describe which caller is allowed to receive which kind of Intent object, using the Intent's attributes such as destination component.

**Implementation as a module:** We ported AppOps and IntentFirewall (from Android v4.3) to a security module for ASF (cf. Figure 4) by moving the AppOpsService and the IntentFirewall classes into a module. Our module comprises 2290 lines of code and differs in 33.71% of all LoC from the original implementation. The bulk of the changes (520 LoC), were required to move the hook logic of both services from the system apps and services of Android into the module by using our enforcement functions. For the IntentFirewall, this was straightforward and we only had to substitute a direct callback from IntentFirewall to the *ActivityManagerService* by our framework callback mechanism. For the AppOpsService, we had to add a mapping from caller PID to package name. By default the hooks of AppOps determine the caller's package name and pass this information to the AppOpsService for policy check. Since this is a policy-specific logic of the hooks, our framework hooks do not (by default) provide the caller's package name and we re-implemented this logic in our module by using our callback interface, which allows us to retrieve the package name for app PIDs. Moreover, we adapted the AppOpsService interface to retrieve/configure the current policies via a Bundle-based communication. AppOps is, furthermore, partially integrated into the *Settings* application to allow users to disable notifications from selected apps. We replaced this policy-specific channel between Settings and AppOps also with our policy-agnostic Bundle-based communication. Modules that support this Settings option, can return a value indicating whether notifications are disabled or not. If the module does not support this feature, Settings app by default allows notifications.

### 6.2 Type Enforcement [36, 8]

SE Android [36] brought SELinux type enforcement to the Android kernel and established the required user space support, e.g., it extended Zygote to label new app processes with a security type. FlaskDroid [8], developed for Android v4.0.3, extends SE Android's type enforcement to Android's middleware. Building on SEAndroid's kernel and low-level patches, it adds policy-specific hooks as policy enforcement points to various system services and apps in Android's middleware. The policy decisions at kernel level are made by the SELinux kernel module, while the decisions at middleware are made centrally in a policy server service. Both policy decision points decide based on subject type, object type, and object class reported by the hooks at their respective layer whether control flows should be truncated or not.

**Implementation as module:** We realized type enforcement with our ASF by porting FlaskDroid[1] as a module (cf. Figure 5), in this context porting the currently hard-

[1]Source code retrieved from `http://www.flaskdroid.org/`

| Existing solution | LoC of module policy engine | LoC added/removed/edited (total delta) |
|---|---|---|
| AppGuard [3] | 5059 | +828/-79/∘13 (18.18%) |
| AppOps / Intent Firewall | 2290 | +627/-106/∘39 (33.71%) |
| FlaskDroid [8] | 4968 | +749/-32/∘40 (16.53%) |

Table 1: Effort of porting different security extensions as module on our Android Security Framework.

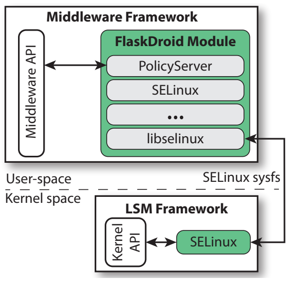

Figure 5: FlaskDroid module

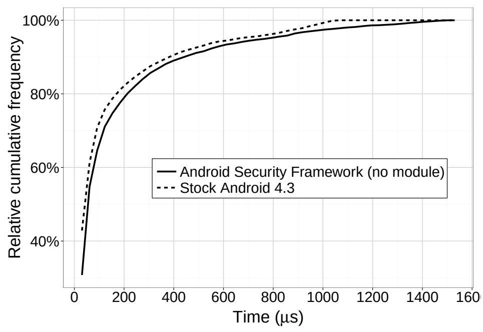

Figure 6: Cumulative frequency distribution of micro benchmarks in stock Android (dashed line) vs. ASF (solid line).

| | Frequency | Mean ($\mu s$) |
|---|---|---|
| **Stock Android 4.3** | 7320 | 116.182±4.550 |
| **ASF v4.3** | 6009 | 129.851±5.681 |

Table 2: Weighted average performance overhead of executing hooked functions in stock Android and in our ASF. The margin of error is given for the 95% confidence interval.

coded SELinux support of Android's middleware into an ASF module. At kernel level, we use the SE Android kernel and provide an SELinux-specific interface implementation for the kernel module. Further, we moved the middleware policy server and its dependencies into the middleware module. Using the enforcement functions of our API, we moved the policy-specific hook logic of FlaskDroid into the module as well. Additionally, we used SE Android's build system to label the file-system with security types.

Our port of FlaskDroid's middleware component as a security module consists of 4968 LoC (cf. Table 1) and differs in only 16.53% of all LoC from the original code. The bulk of these changes (550 LoC) is attributed to additions for implementing a mapping from the enforcement functions of our framework API to FlaskDroid's type checks. To confirm the correct enforcement of policies, we used the policies for middleware and kernel level that are provided with the FlaskDroid source code. Additionally, we noticed during our tests that the original implementation contains an error in assigning middlware security types to processes. Additional changes were necessary to fix this error in our module.

### 6.3 Inlined Reference Monitoring [3]

We use AppGuard [3] as the use-case to illustrate the applicability of our IRM instrumentation API, but similar application rewriting approaches [21] are also feasible. AppGuard is a privacy app for Android that enables end-users to enforce fine-grained access control policies on 3rd party apps by restricting their ability to access critical system resources. By injecting an IRM into apps, this approach supports security policies not easily enforceable by traditional external reference monitors in the Android middleware or kernel, e.g., to enforce the use of *https* over *http*.

**Implementation as a module:** We ported AppGuard[2] as a module for ASF by separating its privacy app into three components: We adapted the (1) AppGuard reference monitor with its dynamic hook placement and policy enforcement logic to use our IRM instrumentation API. The reference monitor is injected into selected app processes via our framework at app startup. The policy decision logic and persistent storage of policy settings was moved into (2) a middleware module. The middlware module selects the apps into which the IRM is injected. It also implements a Bundle-based communication protocol to exchange policy decisions and security events with the IRM component and with (3) a front-end app. The front-end app allows the user to adjust policy settings and to view logs of security-relevant events. We used the policies included in the original AppGuard implementation to confirm that policy enforcement by our module and by the original implementation are identical.

Our AppGuard security module consists of 5059 LoC in total (cf. Table 1), with 782 LoC residing in the middleware module and 4277 LoC in the IRM. Our module diverts in 18.18% of all LoC from the original code. The majority of the difference, 728 LoC, is attributed to moving the policy decision logic into the middleware module, while only 46 LoC were required to adapt the inlined reference monitor to use the provided instrumentation API.

## 7. EVALUATION AND DISCUSSION

We evaluate the performance of our ASF and discuss its current scope and prospective future work. An extended evaluation is provided in the long version of this paper [2].

### 7.1 Performance

Although the actual performance overhead strongly depends on the overhead imposed by the loaded module, we wanted to establish a baseline for the impact of our Android Security Framework on the system performance. The performance of LSM has been evaluated separately, e.g., for SEAndroid [36], and we are interested here in the effect of our new middleware security framework on the performance of instrumented middleware system services and apps.

[2]Source code provided by the original authors

**Methodology.** We implemented our ASF as a modification to the Android OS code base in version 4.3_r3.1 and used the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1*. We performed micro-benchmarks for all execution paths on which a hook diverts the control flow to our middleware framework: We first measured the execution time of each hooked function with no security module loaded and allowing all access. Afterwards we repeated this test with hooks removed to measure the default performance of the same functions and thus operating identical to a stock Android. All our micro-benchmarks were performed on a standard Nexus 7 development tablet (Quad-core 1.51 GHz CPU and 2GB DDR3L RAM), which we booted and then used according to a testplan for different daily tasks such as browsing the Internet, sending text messages and e-mails, contacts management, or (un-)installing 3rd party apps.

**Micro-benchmark results.** Table 2 presents the number of measurements for each test case and their mean values. To eliminate extreme outliers, we excluded in both measurement series the highest decile of the measurements. For ASF the mean is the weighted mean value with consideration of the frequency of each single hook. In overall, our framework with no loaded module imposed with 129.851 $\mu$s approximately 11.8% overhead compared to stock Android. Figure 6 presents the relative cumulative frequency distribution of our measurements series and further illustrates this low performance overhead. Major contributor to this overhead is the marshalling, sending, and unmarshalling of the hooks' parameters. Thus, a future optimization of the baseline overhead would be a framework configuration that enables only the hooks that are actually used by the loaded module(s) and, hence, avoids irrelevant hook invocations.

### 7.2 Current Scope and Future Work

**System setup.** Certain security models require a preparatory system setup. For instance, type enforcement requires a pre-labelling of all subjects and objects. After the system has been setup, ASF supports modularization of these security models (cf. Section 6.2).

**Module Integrity.** As part of the kernel, the KERNEL SUB-MODULE has the highest level of integrity. In contrast, the MIDDLEWARE SUB-MODULE, as a user space process, can be circumvented or compromised by attacks against the underlying system (e.g., root exploits) and thus requires support by the kernel modules to prevent low-level privilege escalation attacks. Inlined reference monitors are inherently susceptible to attacks by malicious applications, because the reference monitor executes in the same process as the application that it monitors and no strong security boundary exists between the monitor and the app code. To remedy this situation, we are currently retrofitting Android's application model to combine the benefits of inlined and of system-centric reference monitors. By splitting apps into smaller units of trust (e.g., app components and ad libs), system-centric reference monitors are able to differentiate distinct trust levels within apps [31, 38, 35].

**Completeness.** It is crucial for the effectiveness of our security framework, that *all* access to security and privacy sensitive resources is mediated by the reference monitors. We consider it out of scope for this submission to formally verify the completeness of our prototype framework, but plan to use recent advances in static and dynamic analysis on Android to verify the placement of our hooks, similarly to how it was done for the LSM framework [12, 17].

**Information flow control.** Our framework provides modules with the control over which subject (e.g., app) has *access to* which objects (e.g., device location), but it cannot control how privileged subjects *distribute* this information. Controlling information flows is an orthogonal problem specifically addressed by different solutions [13, 33]. We plan to integrate such data flow solutions into our framework and to extend our security API with new generic calls for taint labeling and taint checking.

## 8. CONCLUSION

In this paper we presented the ANDROID SECURITY FRAMEWORK (ASF), an extensible and policy-agnostic security infrastructure for Android. ASF allows security experts to develop Android security extensions against a novel Android security API and to deploy their solutions in form of modules or "security apps". Modularizing security extensions overcomes the current unsatisfactory situation that policy authors are either limited to one predetermined security model that is embedded in the Android software stack or that they are forced to confide in a security-model-specific Android fork instead of the mainline Android code base. Additionally, this modularization provides a number of benefits such as easier maintenance and direct comparison of security extensions. We demonstrated the effectiveness and efficiency of ASF by porting different security models from related work to ASF modules and by establishing a baseline for the impact of our infrastructure on the system performance.

## Availability

The ASF source code and example modules can be retrieved from `http://infsec.cs.uni-saarland.de/projects/asf/`.

## Acknowledgements

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

## 9. REFERENCES

[1] M. D. Abrams, K. W. Eggers, L. J. LaPadula, and I. M. Olson. A generalized framework for access control: An informal description. In *NIST NCSC'90*, 1990.

[2] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Enabling generic and extensible access control on android. Technical Report A/01/2014, Saarland University, April 2014.

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

[4] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *IEEE SP'95*. IEEE, 1995.

[5] D. B. Baker. Fortresses built upon sand. In *NSPW'96*. ACM, 1996.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming

privilege-escalation attacks on Android. In *NDSS'12*. The Internet Society, 2012.

[7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *SPSM '11*. ACM, 2011.

[8] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security'13*. USENIX, 2013.

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

[10] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for android. In *ISC'10*. Springer, 2010.

[11] J. Edge. The return of loadable security modules? Online: `http://lwn.net/Articles/526983/`, Nov. 2012.

[12] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In *CCS'02*. ACM, 2002.

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

[14] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE SP'00*. IEEE, 2000.

[15] T. Fraser. LOMAC: MAC you can live with. In *USENIX ATC'01*. USENIX, 2001.

[16] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE SP'99*, 1999.

[17] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In *CCS'05*. ACM, 2005.

[18] V. Gligor, S. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *IEEE SP'98*. IEEE, 1998.

[19] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WISEC'12*. ACM, 2012.

[20] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. Technical Report TUD-CS-2014-0063, Intel CRI-SC at TU Darmstadt, North Carolina State University, CASED / TU Darmstadt, Mar. 2014. To appear at USENIX Security'14.

[21] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *SPSM '12*. ACM, 2012.

[22] B. W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, Jan. 1974.

[23] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, 2005.

[24] T. A. Linden. Operating system structures to support security and reliable software. *ACM Computer Surveys*, 8(4):409–445, Dec. 1976.

[25] Linux Cross Reference. Linux Security Module framework. Online: `http://lxr.free-electrons.com/source/Documentation/security/LSM.txt`.

[26] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *NISSC'98*, 1998.

[27] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *IEEE SP'02*. IEEE, 2002.

[28] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC'09*. ACM, 2009.

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

[30] N. Provos. Improving host security with system call policies. In *USENIX Security'03*. USENIX, 2003.

[31] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security'03*. USENIX, 2003.

[32] V. Rao and T. Jaeger. Dynamic mandatory access control for multiple stakeholders. In *SACMAT'09*. ACM, 2009.

[33] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: supporting operation modes on smartphones. In *SACMAT'12*. ACM, 2012.

[34] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[35] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security'12*. USENIX, 2012.

[36] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS'13*. The Internet Society, 2013.

[37] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *USENIX Security'99*. USENIX, 1999.

[38] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce component-level access control in Android. In *CODASPY'14*. ACM, 2014.

[39] R. Watson, W. Morrison, C. Vance, and B. Feldman. The TrustedBSD MAC Framework: Extensible kernel access control for FreeBSD 5.0. In *USENIX ATC'03*. USENIX, 2003.

[40] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *USENIX Security'02*. USENIX, 2002.

[41] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE SP'12*, 2012.

[42] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on Android). In *TRUST'11*. Springer, 2011.