On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis

Michael Backes, *Saarland University and Max Planck Institute for Software Systems (MPI-SWS);* Sven Bugiel and Erik Derr, *Saarland University;* Patrick McDaniel, *The Pennsylvania State University;* Damien Octeau, *The Pennsylvania State University and University of Wisconsin—Madison;* Sebastian Weisgerber, *Saarland University*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/backes_android

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis

Michael Backes
CISPA, Saarland University & MPI-SWS
Saarland Informatics Campus

Sven Bugiel
CISPA, Saarland University
Saarland Informatics Campus

Erik Derr
CISPA, Saarland University
Saarland Informatics Campus

Patrick McDaniel
Pennsylvania State University

Damien Octeau
Pennsylvania State University &
University of Wisconsin

Sebastian Weisgerber
CISPA, Saarland University
Saarland Informatics Campus

Abstract

In contrast to the Android application layer, Android's application framework's internals and their influence on the platform security and user privacy are still largely a black box for us. In this paper, we establish a static runtime model of the application framework in order to study its internals and provide the first high-level classification of the framework's protected resources. We thereby uncover design patterns that differ highly from the runtime model at the application layer. We demonstrate the benefits of our insights for security-focused analysis of the framework by re-visiting the important use-case of mapping Android permissions to framework/SDK API methods. We, in particular, present a novel mapping based on our findings that significantly improves on prior results in this area that were established based on insufficient knowledge about the framework's internals. Moreover, we introduce the concept of permission locality to show that although framework services follow the principle of separation of duty, the accompanying permission checks to guard sensitive operations violate it.

1 Introduction

Android's application framework—i.e., the middleware code that implements the bulk of the Android SDK on top of which Android apps are developed—is responsible for the enforcement of Android's permission-based privilege model and as such is also a popular subject of recent research on security extensions to the Android OS. These extensions provide various security enhancements to Android's security, ranging from improving protection of the user's privacy [26, 46], to establishing domain isolation [29, 12], to enabling extensible access control [21, 8].

Android's permission model and its security extensions are currently designed and implemented as best-effort approaches. As such they have raised questions about the efficacy, consistency, or completeness [3] of the policy enforcement. Past research has shown that even the best-efforts of experienced researchers and developers working in this environment introduce potentially exploitable errors [15, 44, 35, 33]. In light of the framework size (i.e., millions of lines of code) and based on past experience [15, 44, 16, 33, 36], static analysis promises to be a suitable and effective approach to (help to) answer those questions and hence to demystify the application framework from a security perspective. Unfortunately, on Android, the technical peculiarities of the framework impinging on the analysis of the same have not been investigated enough. As a consequence, past attempts on analyzing the framework had to resort to simple static analysis techniques [7]—which we will show in this paper as being insufficient for precise results—or resort to heuristics [33].

In order to improve on this situation and to raise efficiency of static analysis of the Android application framework, one is confronted with open questions on *how to enable more precise static analysis of the framework's codebase*: where to start the analysis (i.e., what is the publicly exposed functionality)? Where to end the analysis (i.e., what are the data and control flow sinks)? Are there particular design patterns of the framework runtime model that impede or prevent a static analysis? For the Android application layer, those questions have been addressed in a large body of literature. Thanks to those works, the community has a solid understanding of the sinks and sources of security- and privacy-critical flows within apps (e.g., well-known Android SDK methods) and a dedicated line of work further addressed various challenges that the Android *application* runtime model poses for precise analysis (e.g., inter-component communication [28, 40, 24, 27] or modelling the Android

app life-cycle[25, 6]). Together those results form a strong foundation on which effective security- and privacy-oriented analysis is built upon. In contrast to the app layer, for the application framework we have an intuitive understanding of what constitutes its entry points, but no in-depth technical knowledge has been established on the runtime model, and almost no insights exist on what forms the security and privacy relevant targets of those flows (i.e., what technically forms the sinks or "protected resources").

Our Contributions. This paper contributes to the demystification of the application framework from a security perspective by addressing technical questions of the underlying problem on *how* to statically analyze the framework's code base. Similar to the development of application layer analyses, we envision that our results contribute some of the first results to a growing knowledge base that helps future analyses to gain a deeper understanding of the application framework and its security challenges.

How to statically analyze the application framework. We present a systematic top-down approach, starting at the framework's entry points, that establishes knowledge and solutions about analyzing the control and data flows within the framework and that makes a first technical classification of the security and privacy relevant targets (or resources) of those flows. The task of establishing a precise static runtime model of the framework was impeded by the absence of any prior knowledge about framework internals beyond black-box observations at the framework's documented API and manual analysis of code fragments. Hence we generate this model from scratch by leveraging existing results on statically analyzing Android's application layer at the framework layer. The major conceptual problem was that the design patterns of the framework strongly differ from the patterns that had been previously encountered and studied at the application layer. Consequently we devised a static analysis approach that systematically encompasses all framework peculiarities while maintaining a reasonable runtime. As result of this overall process, we have established a dedicated knowledge base that subsequent analyses involving the application framework can be soundly based upon.

Axplorer *tool and evaluation.* Unifying the lessons learned above, we have built an Android application framework analysis tool, called Axplorer. We evaluate Axplorer on four different Android versions—v4.1.1 (API level 16), v4.2.2 (17), v4.4.4 (19), and v5.1 (22)—validate our new insights and demonstrate how specialized framework analyses, such as message-based IPC analysis and framework component interconnection analysis, can be used to speed up subsequent analysis runs (e.g. security analyses) by 75% without having to sacrifice precision. As additional benefit the resulting output can be used by independent work *as is* to create a precise static runtime model of the framework without the need to re-implement the complex IPC analysis.

Android permission analysis. Finally, to demonstrate the benefits of our insights for security analysis of the framework, we conduct an Android permission analysis. In particular, we re-visit the challenge of creating a permission map for the framework/SDK API. In the past, this problem has been tackled [32, 7] without our new insights in the peculiarities of the framework runtime model, and our re-evaluation of the framework permission map reveals discrepancies that call the validity of prior results into question. Using Axplorer, we create a new permission map that improves upon related work in terms of precision. Moreover, we introduce a new aspect of permission analysis, permission locality, by investigating which framework components enforce a particular permission. We found permissions that are checked in up to 10 distinct and not necessarily closely related components. This indicates a violation of the separation of duty principle and can impede a comprehensive understanding of the permission enforcement.

2 Background

We first provide necessary technical background information on the Android software stack and the abstract control and data flows in the system. Android OS is an open source software stack on top of the Linux OS. Between the apps at the top of the stack and the Linux kernel at the bottom is the Android middleware. This middleware consists of the application runtime environment, default native libraries (like SSL), and the Java-based application framework (see Figure 1).

2.1 Android Application Framework

The application framework consists of the various services that implement the Android app API (e.g., retrieving location data or telephony functionality). Every framework service is responsible for providing access to one specific system resource, such as geolocation, radio interface, etc.

Bound services. These services are implemented as `bound services` [4] as part of the `SystemServer`.

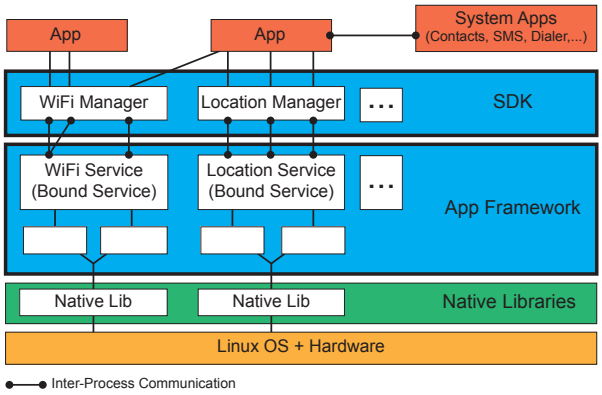

Figure 1: Android Software Stack with abstract control and data flows.

Bound service is the fundamental pattern to realize Android services that are remotely callable via a well-defined interface. Such interfaces are described in the *Android Interface Definition Language* (AIDL) and an AIDL compiler allows automated generation of `Stub` and `Proxy` classes that implement the interface-specific Binder-based RPC protocol to call the service. Here, `Stubs` are an abstract class that implements the `Binder` interface and needs to be extended by the actual service implementation. `Proxies` are used by clients to call the service. On top of `Stubs` and `Proxies`, the Android SDK provides `Managers` as abstraction from the low-level RPC protocol. `Manager` classes encapsulate pre-compiled `Proxies` and allow developers to work with `Manager` objects that translate local method calls into remote-procedure calls to their associated service and hence enable app developers to easily engage into RPC with the framework's services. However, `Proxies` and `Managers` are just abstractions for the sake of app developer's convenience and nothing prevents an app developer from bypassing the `Managers` or re-implementing the default RPC protocol to directly communicate with the services.

A small number of framework services does not use AIDL to auto-generate their `Stub`/`Proxy`, but instead provides a custom class that implements the `Binder` interface. The most prominent exception is the `ActivityManagerService` (AMS), which provides essential services such as application life-cycle management or Intent distribution. Since its interface is also called from native code, for which the AIDL compiler does not auto-generate native `Proxies`/`Stubs` and hence requires manual implementation of those, the RPC protocol for the AMS is hardcoded to avoid misalignment between manually written and auto-generated code.

The services are an essential part of the middleware front-end to the application layer and calling their interfaces triggers control and data flows within the application framework. Naturally, the flows of some services lead to interaction with the underlying platform through the native libs. For instance, the `WifiService` is interacting with the WiFi daemon. Other services, such as Clipboard, do not rely on any hardware features. However, the exact control and data flows have not yet been studied or charted (see blank boxes in Figure 1) and facilitating this mapping by enabling analysis of the framework is part of the contributions of this work.

System apps. System apps, such as Contacts, Dialer, or SMS complement the application framework with commonly requested functionality. However, in contrast to the application framework services that are fixed parts of any Android deployment, system apps are exchangeable or omittable (as can be observed in the various vendor customized firmwares) and, more importantly, are simply apps that are programmed against the same application framework API as third-party applications.

2.2 Permissions

One cornerstone of the Android security design are *permissions*, which an app must hold to successfully access the security and privacy critical methods of the application framework. Every application on Android executes under a distinct Linux UID and permissions are simply Strings[1] that are associated with the application's UID. There is no centralized policy for checking permissions on calls to the framework API. Instead, framework services that provide security or privacy critical methods to applications (must) enforce the corresponding, hard-coded permission that is associated with the system resources that the services expose. To enforce permissions, the services programmatically query the system whether their currently calling app—identified by its UID—holds the required permission, and if not take appropriate actions (such as throwing an exception). For instance, the `LocationManagerService` would query the system whether a calling UID is associated with the String `android.permission.ACCESS_FINE_LOCATION`, which represents the permission to retrieve the GPS location data from the `LocationManagerService`.

In this model, system apps differ from third-party apps in that they can successfully request security

[1] Permissions that map to Linux GIDs do not involve the framework and are not further considered here.

and privacy critical system permissions from the framework, which are not available to non-system apps. Moreover, like framework services (and any non-system application), they are responsible for enforcing permissions for resources they manage and expose on their RPC interfaces (e.g., contacts information or initiating phone calls). The difference to non-system applications is, that they usually enforce well-known permissions defined in the Android SDK, although the Android design does—in contrast to the framework services—not hardcode where those permissions are enforced, thus allowing system apps to be exchanged.

3 Related work

Static (app) analysis. Different related works have analyzed Android apps for vulnerabilities and privacy violations. Enabling precise static app analysis required solving essential questions like what are the entry points of the app, what are the security relevant sinks and how can we achieve a static runtime model that takes the application peculiarities into account? Among the static analysis approaches, *CHEX* [25] was the first tool to accommodate for Android's event-driven app lifecycle with an arbitrary number of entry points. *FlowDroid* [6] further improved the runtime model by automatically generating per-component lifecycle models that take into account the partial entry point ordering. While *FlowDroid* still analyzed components in isolation, a number of related works specifically addressed the problem of inter-component communication (ICC). The initial work *Epicc* [28] devised a new analysis technique to create specifications for each ICC sink and source. *Amandroid* [40] combined a lifecycle-aware program dependence graph with ICC analysis to generate an inter-component model of the application to improve precision for various security applications. Similarly, *IccTA* [24] extended *FlowDroid* with a precise inter-component model. Finally, *IC3* [27] uses composite constant propagation to improve retargeting of ICC-related parameters enabling a more precise ICC resolution. Moving from best effort approaches, *SuSi* [5] took a machine-learning approach for classifying and categorizing sources and sinks in the framework code that are relevant for application analysis. All of those solutions contribute to analyzing Android apps more efficiently. The focus of this work is on establishing similar knowledge on Android's application framework and on making a first essential but non-trivial step towards enabling a holistic analysis of Android that includes the framework code with its security architecture.

Application Framework Abstractions. The application framework is generally regarded as too complex to be considered in an app analysis (cf., *CHEX* [25]) and very recent works dealt specifically with this problem of abstracting the application framework [13, 18] or making it amenable for app analysis [11]. *EdgeMiner* [13] links callback methods to their registration methods and generates API summaries that describe implicit control flow transitions through the framework. *DroidSafe* [18] distills a compact, data-dependency-aware model of the Android app API and runtime from the original framework code. *Droidel* [11] differs in its approach by explicating the reflective bridge between the application framework and applications, while trying to model the framework as less as possible. It generates app-specific versions of the application framework and replaces reflective calls with app-specific stubs. All of these approaches try to pre-compute data-dependencies through the framework API that can be used by app analyses in favor of using the complex and huge framework code base. In contrast, our work makes a first step towards enabling in-depth analyses of the application framework beyond just data dependencies in order to enable future reasoning about framework security architectures or extensions (such as guiding and verifying hook placement or separation of duties).

Permission Mapping and Inconsistencies. Both *Stowaway* [32] and *PScout* [7] built permission maps for the framework API. Stowaway used unit testing and feedback directed API fuzzing of the framework API to observe the required permission(s) for each API call. PScout, in contrast, used static reachability analysis between permission checks and API calls to create a permission mapping of different Android framework versions that improves on the results of Stowaway. Permission maps have since been a valuable input to different Android security research, such as permission analysis [20] and compartmentalization [31, 34] of third-party code, studying app developer behavior [32, 38], detecting component hijacking [25], IRM [23, 10] and app virtualization [9], or risk assessment [30, 19, 45, 42]. In this work, we re-visit the challenge of creating a permission map for Android. In contrast to the prior work, we build on top of our new insights on how to statically analyze the application framework (see Sections 4 and 5), which allow us to achieve a map that is more precise for the application framework API and that calls the validity of some prior results [7] into question. We discuss how recent work [33] that focused on inconsistent security enforcement within the framework

could benefit from a deeper understanding of the framework's peculiarities separately in Section 8.

Android Security Frameworks. Various security extensions have been proposed, such as [26, 46, 29, 12, 21, 8] to name a few, which integrate authorization hooks into Android's application framework to enforce a broad range of security policies. At the moment, those extensions are designed and implemented as best-effort approaches that raise questions about the completeness and consistency of the enforcement and indeed past research has shown that even the best-efforts of highly experienced researchers and developers working in this environment introduce potentially exploitable errors [15, 44, 35, 33]. This unsatisfying situation has strong parallels to earlier work on integrating authorization hooks into the Linux and BSD kernels [41, 39], where a dedicated line of work [15, 44, 16] has established tools and techniques to reason about the security properties of proposed extensions or to automate the hook placement. Prerequisite for those solutions was a clear understanding of what constitutes a resource that is (or should be) protected by an authorization hook. To allow development of similar tools for the Android application framework, we hence have to also answer the question about Android's protected resources first. In this work we want to make a first essential step in this direction by enabling a deeper analysis of the framework and by providing a first high-level taxonomy of protected resources in the application framework.

4 Enabling In-Depth Application Framework Analysis

In contrast to the various related works on static analysis at the application level, there is no existing prior work on in-depth analysis of the application framework. Moreover, as the architecture of the framework fundamentally differs from the architecture of applications, open questions have to be answered first to be able to conduct in-depth static analysis of the framework. For instance, *"what are the entry points to the application framework?"* or *"how to establish a static runtime model of the framework's control flows?"* In the following we identify challenges that arise for static analyses at framework level and present a systematic, top-down approach to cope with these problems (an implementation of our approach is presented in Section 5). Solving the discussed challenges lays the foundation on which a wide range of security analyses of the application framework can be constructed, from which we (re-)visit the use-case of permission analysis in Section 7.

4.1 Defining Framework Entry Points

The first question to be answered is how to identify and select the starting points for the framework analysis? At application level this has already been studied in depth [25, 6, 14, 17, 43]. From a high-level view, most approaches parse the declared components from the application manifest and determine the components as well as dynamically registered callbacks as entry points; or they build component/application life-cycle models with a single main entry method.

> **Challenge:** *The framework model is conceptually different from the application layer and existing approaches for application layer analysis do not apply in a framework analysis. Instead one has to identify the framework API methods that are exposed to app developers as analysis entry points.*

To identify the entry point methods, we have to locate the relevant framework entry point classes. Starting with the official API of the Android SDK (e.g., `Managers` in Figure 1) is not reliable as there are no means to prevent an app developer from bypassing the SDK by immediately communicating with the framework services or by using reflection to access hidden API methods of the SDK. Consequently, we do not consider the API calls within the SDK as entry points but instead the framework classes that are entry points for accessing framework functionality (i.e., framework classes that are being called by the SDK, see Figure 1). We exclude entry points that are not accessible by app developers, such as Zygote, service manager, or the property service, which are under special protection (e.g., SELinux [37]) and will not accept commands by third-party apps that have tangible side-effects on the system or other apps. This restriction is in accordance with the design of existing Android security extensions, which exclusively focus on the exported functionality of the app framework (e.g., framework's bound services).

Inter-component communication in Android is by design based on `Binder` IPC and, thus, framework classes have to expose functionality via `Binder` interfaces to the application layer. To this end, interfaces must be derived from `IInterface`, the base class for `Binder` interfaces. Binder interfaces might be automatically generated by AIDL, in this case the entry point classes extend the auto-generated `Stub` class, or in case of `Binder` interfaces that are not generated by AIDL, a custom `Binder` implementation

like `ActivityManagerNative`[2] has to be provided, which in turn is extended by the entry point classes. These class relationships can be resolved via a class hierarchy analysis (CHA) to determine the set of all entry classes. Besides bound services this also includes callback and event listener classes that expose an implementable interface to app developers. Hence, we define entry points (EP) as the public methods of framework classes that are exposed via a `Binder` interface. In addition, permission-protected entry points (PPEP) are defined as entry points from which a permission check is control-flow reachable.

4.2 Building a Static Runtime Model

> **Challenge:** *Generating a static model that approximates the runtime behaviour of the application framework again strongly differs from the problems that arise at application level where the component life-cycles are mimicked to approximate runtime behavior. The bound services—as entry points to the framework—might be queried simultaneously from multiple clients (apps) via IPC and hence have to handle multi-threading to ensure responsiveness of the framework. In contrast to the application space at which utility classes like* `AsyncTask` *are used for threading, we discovered that the framework services make intensive use of more generic but also more complex threading mechanisms like* `Handler`*,* `AsyncChannel`*, and* `StateMachines`*. Disregarding these concurrency patterns results in imprecise data models that cause a high number of false positives during framework analysis.*

In the following, we provide technical background for those asynchronicity patterns and explain how they can modeled correctly for static analyses.

Handler. The class `android.os.Handler` provides a mechanism for reacting to messages or submitting Java `Runnable` objects for execution on a (potentially remote) thread. `Handlers` either schedule the processing of a message or the execution of a `Runnable` at some point in the future or process a message/`Runnable` on a separate thread.

To illustrate the `Handler` mechanism, consider the example shown in Listing 1. It includes the relevant sections of the framework class `com.android.server.BluetoothManagerService`. When the service is constructed, it instantiates a `HandlerThread` object (line 6), a traditional `Thread` object associated with a `Looper`. The

[2]By convention non-generated class names end with *Native*.

```
1  class BluetoothManagerService {
2    private HandlerThread mThread;
3    private BluetoothHandler mHandler;
4
5    public BluetoothManagerService() {
6      mThread = new
           HandlerThread("BluetoothManager");
7      mThread.start();
8      mHandler = new
           BluetoothHandler(mThread.getLooper());
9    }
10   public void enable() {
11     Message msg =
           mHandler.obtainMessage(MESSAGE_ENABLE);
12     mHandler.sendMessage(msg);
13   }
14   public void disable() {
15     Message msg =
           mHandler.obtainMessage(MESSAGE_DISABLE);
16     mHandler.sendMessage(msg);
17   }
18
19   class BluetoothHandler extends Handler {
20     public void handleMessage(Message msg) {
21       switch (msg.what) {
22         case MESSAGE_ENABLE:
23           // process enable message
24           break;
25         case MESSAGE_DISABLE:
26           // process disable message
27           break;
28         // Other cases.
29   }}}
30 }
```

Listing 1: Bluetooth Handler in the Bluetooth manager service. Code was simplified for readabililty.

purpose of the `Looper` class is to sequentially process the messages in a message queue. At line 8, the class-specific `BluetoothHandler` object is created and associated with the newly created `Looper` from the `HandlerThread`. This allows messages sent to the `BluetoothHandler` to be pushed to the message queue for this `Looper`. Methods `enable` and `disable` can be called by applications via RPC on `IBluetoothManager` to turn the bluetooth functionality on or off. Method `enable` sends a message with code `MESSAGE_ENABLE` to the `BluetoothHandler` (line 12). When the associated `Looper` instance processes the message, it calls method `handleMessage` in the `BluetoothHandler` (line 20), which then processes the request.

Statically resolving message-based IPC, requires to overcome several challenges. First, the target `Handler` type has to be inferred, to determine the concrete `handleMessage` method of the receiving class that processes the message. Second, to add precision to the analysis, it is best to make it locally path-sensitive by inferring the possible message codes of the arguments to `sendMessage` methods. For the example presented in Listing 1, this enables the analysis to be limited to the feasible paths for a given message in the switch at line 21. While it is possible to per-

form the analysis without this information, doing so results in a significant loss of precision and, thus, an increase in the number of false positives, which may distort the results of security analyses built on top. In light of the prevalence of the `Handler` pattern, this loss of precision is not an acceptable solution. Finally, since messages can also be associated with runnable tasks instead of message codes, the concrete `Runnable` types associated with each message have to be inferred to determine the runnable code executed when such a message is processed.

AsyncChannel. Closely related to `Handlers`, `com.android.internal.util.AsyncChannel` implements a bi-directional channel between two `Handler` objects. It provides its own `sendMessage` and `replyToMessage` methods, both of which delegate to the `sendMessage` methods in its associated `Handler`. In order to precisely model `AsyncChannel` objects, it is necessary to infer the types of the sender/receiver `Handler` objects. Similarly to `Handlers`, path-sensitivity should be added to the analysis by inferring the message codes that are sent through the channel.

StateMachine. Building on the `Handler` concept, the `com.android.internal.util.StateMachine` class models complex subsystems such as the DHCP client or the WiFi connectivity manager. This class allows processing of messages depending on the current state of the modeled system. It effectively constitutes a hierarchical state machine in which messages cause state transitions. States are organized in a hierarchical manner, such that parent states may process messages that are not handled by child states. In order to precisely model state machines, several challenges must be addressed. First, the subtype of the state machine itself must be inferred, with all the states and possible transitions. Second, the hierarchy of the states must be inferred, in order to know which `enter` and `exit` state methods are called upon state transitions. Moreover, this is necessary to know which state may handle a given message. Third, for eliminating further false positives one needs to infer the possible states for any given program location at which interaction with the state machine occurs.

4.3 Identifying Protected Resources

While the previous sections show how static analysis of the android application framework code base can be enabled, we now classify the resources inside the application framework that actually have to be protected. Unfortunately, there is a lack of consensus in the community on what constitutes a security-sensitive resource/operation [5, 16] and no one-size-fits-all definition exists as the concrete definition depends on various aspects like operating system, programming language, or even the domain. To avoid ambiguities on what we denote as protected resource in the remainder of this paper, we note that protected resources for us are security sensitive operations that have a tangible side effect on the system state or use of privacy.

> **Challenge:** *Defining the security-relevant resources is, in contrast to entry points, more challenging. For privacy leak analysis at application-level, there is a well-defined list of API methods that can be classified as sinks. Since the analysis now shifts into the API methods of the framework, it is unclear what kind of resources are protected by Android's permissions and can, thus, be used as sinks for security analysis within the framework.*

To create a first high-level taxonomy of protected resources that can help to automatically discover such resources, we first have to create a ground truth about what technically forms a protected resource. To this end, we manually investigated control flows of a number of identified PPEP in the framework's source code. Here, we make the assumption that every existing permission check within the application framework indeed controls access to at least one security- or privacy-critical system resource. Checks are usually located at the very beginning of PPEP, so that any subsequent operation is indeed authorized. Using expert knowledge in combination with descriptions of expected side effects from the Android documentation we identify and annotate relevant statements that modify the service and/or system state. To avoid a potential bias in the types of protected resources, we chose entry points from eight different entry classes. To cover a variety of disjunct cases, we based our selection on the available information such as return value, number/type of EP input arguments, or number/type of permission checks collected during the entry point discovery. After manually investigating flows from 35 entry points, distinct repetitive patterns for protected resources appeared across the different control flows, which we summarized in a taxonomy of the high-level protected resource types.

Taxonomy of protected resources. Figure 2 presents our high-level taxonomy of the protected resource types. In contrast to work at application-level that disregards field instructions [5], we found that *field update* instructions are highly relevant in the

context of the framework and in fact are the most prevalent type of protected resources that we discovered. Relevant *method invocations* can be further subclassified into native method calls (e.g. for file system access or modification of device audio settings) and broadcast sender. We consider native method calls generally as protected resources, since distinguishing non-/security-relevant native calls would require a dedicated analysis for the native code, which is currently a general, open problem for the community and out of scope for this work. Broadcast senders are protected resources as they can potentially cause in the receivers side-effects on the system or apps. However, this is statically unresolvable, as the concrete side-effects strongly depend on the current system configuration, e.g. on the installed apps and the set of active broadcast listener. We consider non-void *return values* of security-sensitive entry methods as protected resource. Returned objects of such methods constitute sensitive data, e.g., a list of WiFi connections. Return values of primitive types `int` or `boolean` may constitute sensitive values like for the method `isMultiCastEnabled` of the `WifiService` or some status/error code in method `enableNetwork` of the same service. We also found cases in which a *throw RuntimeException* (RTE) has to be considered as a protected resource. For instance, in the `crash` method of the `PowerManagerService`, which requires from the caller the permission to reboot the device, an RTE causes the runtime to crash and the device to reboot in consequence.

Coverage of the taxonomy. An inherent limitation of our taxonomy based on small-scaling manual analysis is, that there are no guarantees that corner cases are included in the current classification. To cover all corner cases in our taxonomy, a comprehensive manual analysis of the framework would be required, which would defeat the purpose of enabling a static analysis in the first place. This constitutes a high-level taxonomy of protected resource (types) in the framework. Distilling a more refined set for security analyses is discussed separately in Section 8.

5 Implementation

We combined all aforementioned steps from Section 4 for analysis of an arbitrary framework version into a tool called Axplorer. We leverage the static analysis framework WALA [2], although our approach is equally applicable to other analysis frameworks such as Soot [1]. Additional code for realizing our approach comprises ≈15 kLOC of Java.

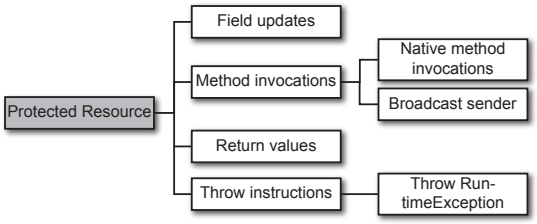

Figure 2: High-level taxonomy of protected resource operation types.

Call-graph generation. For each identified entry class, we generate an inter-procedural call-graph (CG). As opposed to related approaches [7] that use class hierarchy analysis to generate low-precision call-graphs due to the overall framework complexity—Android version 4.2.2 already includes over 35,000 classes—we generate high-precision call-graphs with object-sensitive pointer resolution. For each virtual or interface invocation we infer the runtime type(s) and hence precisely connect the invocation to its target(s). Although the costs for the points-to computation are computational very expensive, the increased precision lowers the complexity of the overall call-graph, since we do not introduce imprecision by considering all subclasses of a virtual method call as potential receivers. Avoiding this imprecision in the call-graph also lowers the number of false positives. The complexity is further reduced by the design decision to not follow RPC calls to other entry classes. We complement the call-graph with message-based IPC edges during the control-flow slicing (see below).

Slicing & on-demand msg-based IPC resolution. We conduct a forward control-flow slice for each identified entry point method. The slicer stops at native methods, RPC invocations to classes other than the current one, and when the entry point method returns. During slicing, we perform an on-demand message/handler resolution to add message-based IPC edges to the call-graph, thus avoiding a huge computational overhead of computing all edges in advance when only a subset of them are required for analysis (e.g. if PPEP are analyzed only).

When the slicer reaches a `sendMessage` call, we infer the concrete handler type and add a call edge from the `sendMessage` call to the `handleMessage` method of the receiving handler. We augment this process with inter-procedural backwards slicing for two reasons: First, since existing type inference algorithms (like the ones implemented in WALA) work intra-procedurally, type inference fails if `Handler` objects are stored in fields whose declared field type is the

`Handler` base class and not the concrete subtype. Using inter-procedural backwards slicing starting at the message-sending instruction, we obtain a more precise set of possible handler types in AXPLORER. Second, `Messages` are usually not constructed explicitly but indirectly obtained via calls to `Message.obtain` or `Handler.obtainMessage` and contain a public integer field that carries a sender-defined message code that allows the recipient to identify the message type. To statically identify the message code we compute a backwards slice starting from the message-sending instruction and check the resulting set of instructions for calls that construct/obtain a message. We then repeat this approach starting from the message obtain call to infer the concrete message code used to initialize the `Message`.

`Handlers` use `switch` statements to match the provided message code and to transfer control-flow to a specific basic block of the method's control-flow graph (cf. line 25 et seqq. in Listing 1). To avoid infeasible paths, we have to recreate path-sensitivity intra-procedurally and map the message code(s) to the individual execution path(s). The control-flow slicer then continues at this specific execution path to avoiding a huge number of false positives. `Runnable` types on a `post` call of the `Handler` are resolved in the same way and a call edge to the `Runnable`'s `run` method is added. The approach slightly differs in case of `StateMachines`. Here, there is no single `handleMessage` function. Instead, each `State` implements its own `processMessage` function. In this case, we recreate path-sensitivity for each of these functions and delegate the control-flow to any matching switch statement.

6 Framework Complexity Analysis

We apply our gained insights from Section 4 to collect complexity information about the application framework. By doing this, we demonstrate how the analysis complexity can be held manageable to allow such in-depth analysis within a reasonable amount of time. Finally, we collect the framework's protected resources as denoted in our taxonomy and validate the results (a detailed discussion on how security analyses can benefit from this is given in Section 8). Using AXPLORER we analyze four different Android versions: 4.1.1 (API level 16), 4.2.2 (17), 4.4.4 (19), and the latest Lollipop release 5.1 (22).

6.1 Handling Framework Complexity

Table 1 summarizes different complexity statistics generated for the four analyzed versions. Unsurprisingly, the complexity in terms of code increases with each version, whereas the gap to the most recent major version is significantly larger as between the minor version changes due to new features like Android TV. The entry class discovery algorithm identified between 242–383 entry classes of which ≈25% include at least one PPEP. The evaluation was conducted on a server with four Intel Xeon E5-4650L 2.60 GHz processors with 8 cores each and 768 GB RAM. Initial processing of the frameworks finished in reasonable time, ranging from 14–126 hours. Note that this computation has to be done only once per Android version and that there are no real-time constraints as, e.g., in application vetting. The most time-consuming task (about 85% of the overall time) was generation of the high-precision call-graphs. In the following, we describe the use of entry-class interconnection and IPC analysis to speed up processing time without loosing the precision of our data model.

Entry class interconnection. IPC-interfaces of framework entry classes are not only used by the application layer, but also by other framework services. Analyzing the communication behavior of entry classes does not only provide a deeper understanding of how the framework services are interconnected but also facilitates analyses that rely on permission checks as security indicator (e.g., see Section 7). Exploiting the knowledge about which service EP triggers which RPCs along its control flow enables pre-computation of execution path conditions and restricting the scope of a service analysis to only subsets of dependent services rather than the entire framework (i.e, it allows to efficiently divide and conquer the framework analysis). In a post-processing step the analysis results for distinct services can be stitched together at RPC boundaries. Appendix A illustrates the RPC interconnections for Android 5.1.

Message-based IPC Analysis. A precise model of the message sender to handler relations is crucial for the generation of a static runtime model of the framework with a low number of false connections. The last row in Table 1 shows the prevalence of the message sending pattern. Between 38–52% of PPEP include at least one message sending call. Across API levels we found 300 (API 16) to over 500 (API 22) distinct message sender calls used within PPEP. The evaluation of our IPC analysis showed that in 7% of all cases the message was sent to a `StateMachine`, and in 27% of all cases to a `Handler`. In the remaining 66% a `Runnable` was posted. This ratio remains approximately the same in all versions. Overall, our IPC analysis was able to fully resolve about 76%

Android version	4.1.1 (16)		4.2.2 (17)		4.4.4 (19)		5.1 (22)	
# of classes	27,749		29,804		31,023		46,192	
- inner classes	14,784		15,936		17,525		28,933	
# of entry point classes	242		256		284		383	
- with at least one PPEP	64	(26.4%)	73	(28.5%)	75	(26.4%)	81	(21.2%)
# entry methods (EP)	2,583		2,734		2,861		3,225	
- with perm check (PPEP)	863	(33.4%)	1,018	(37.2%)	1,227	(42.9%)	1,250	(38.8%)
- incl. message sending	328	(38.0%)	532	(52.2%)	518	(42.2%)	597	(47.8%)

Table 1: Comparing complexity measures for different Android versions (percentages relate to preceding line).

of all message sending instances, yielding already a very valuable data set of the message sender to handler relationships. Reasons for failed resolution are that either the `Handler` (81%) or `Runnable` (5%) could not correctly be inferred while in the remaining 13% of cases the message code could not be inferred. The root cause of most of these failures is the missing/incomplete support of `AsyncChannels` and the `Message.sendToTarget()` API call. At the time of writing this support is work-in-progress.

During our initial analysis run AXPLORER records both an RPC-map per entry class as well as a list of resolved sender-to-handler relationships. This data is then re-applied as expert knowledge in subsequent analysis *re*-runs to significantly reduce the analysis runtime, e.g., for API level 17, the processing time drops by ~75% to about 7 hours. By publishing this data we hope that independent analyses can equally benefit from this by removing the burden to re-implement a comparable IPC resolution algorithm.

Reflection We analyzed reflection usage within framework code by counting the number of calls to methods within the `java.lang.reflect` package. The absolute numbers range from 89 (API 16) to 118 (API 22). Across API levels less than 50% targeted the `Method` class while the remaining calls were distributed among other reflection classes. In many cases reflection is used in utility or debug classes and we found only one entry class that makes use of reflection (`ConnectivityService`), but the respective method was removed in API level 20. In SDK code the total numbers are slightly higher across API levels (115–288). However, the additional usage of reflection is mainly due to `View`/`Widget` classes. Overall, reflection is only rarely used in framework code and not used at all by main service components.

6.2 Android's Protected Resources

To validate our established taxonomy, we collect the protected resources for each Android version and classify them with respect to the taxonomy. Across versions the total number ranges from 6,5k (API 16) to 10k (API 22). Although these numbers seem quite high at first glance, they are reasonable in relation to the overall size and complexity of the framework. AXPLORER recorded the context depth (in terms of method invocations) at which the protected resources were found. While for simple methods that include few (or only even one) resource the call depth is lower than two, the median call depth ranges from 8–11 across Android versions. This emphasizes that approaches that do not perform in-depth analysis are not suitable to detect resources located deeper in the control-flow. The relative distribution of resources per type is stable across all versions. We validated our statement of Section 4.3 that field update instructions are the most prevalent resource type (with a share of about 75%). They are followed by native method calls (about 21–23%), which are most frequently used as a gateway to the device hardware (e.g. file system, audio, nfc). There is a surprisingly low number of PPEP that return a protected value, the absolute number ranges from 51–69 entries. Another unexpected result is that runtime exceptions occur with a frequency that is about as high as protected broadcast senders. Besides the already mentioned example within the `PowerManagerService`, we found occurrences in UI widget classes and even in the default XML parsing library on Android.

Appendix B gives more detailed statistics on protected resources, as well as a manual validation and assessment of the use of RTE in the framework.

7 Permission Analysis

Building on top of our new insights we re-visit an important aspect of Android's permission specification, that is *permission mapping* between permission check and SDK method, and further introduce *permission locality* to study which framework components perform which permission checks. To this end we extend AXPLORER as follows:

1) A PPEP only indicates the presence of a permission check in the control-flow from this

entry-point, but there is no information yet about the number of checks or the concrete permission strings. We extend our slicing-based approach to also resolve the permission strings in common permission check API invocations (e.g., as defined in the `Context` class). Non-constant strings are resolved in a similar way like message codes in Section 5. From 520 distinct permission checks found in API level 16, we were able to resolve 99% of the permission strings. Among the failing cases, one case was located in the `ActivityManagerService$PermissionController` class where the permission string is an argument of the entry point method, which is only called from native code and hence was not statically resolved.

2) Entry class interconnection, i.e., RPC transitions to other PPEP (see Section 6.1), usually accumulates all permissions required by the additionally called entry classes for the UID that called the first entry class in the control-flow. However, those transitions are irrelevant for permission analysis when the RPC is located between calls to `Binder.clearCallingIdentity` and `Binder.restoreCallingIdentity`. Clearing the calling UID in the framework's bound services resets it from the calling app's UID to the privileged system server UID. Thus, outgoing IPC edges after clearing and before restoring the UID should be ignored in permission analysis, since the additional PPEP are called with a UID that is different from the calling app's UID.

3) We add a light-weight SDK analysis to reason about required permissions of documented APIs. To this end, we conduct a reachability analysis from public SDK methods to framework EPs (SDK to framework layer in Figure 1). Combining this mapping with the mapping from framework EPs to permissions creates a permission map for the documented API.

7.1 Re-Visiting Permission Mapping

The *Stowaway* project [32] were the first to generate a comprehensive permission map for Android 2.2. Their dynamic analysis approach (feedback directed API fuzzing) generates precise but incomplete results. Moreover, the involved manual effort makes it difficult to re-use it for newer API versions. *PScout* [7] improved on this situation by statically analyzing the framework code, thus increasing the code coverage. In direct comparison *PScout*'s results contain notably more permission mappings. To handle the complexity induced by the framework size, *PScout* resorts to low-precision data models based on class hierarchy information. In the following, we demonstrate that this has negative implications for their resulting permission map. Using our insights we provide permission mappings that call the validity of prior mappings into question.

We compare our results with *PScout* using their latest available results (for Android 4.1.1) [3]. Since we exclude `Intent` and `ContentProvider` permissions, which both require supplemental analysis effort such as manifest or `URI` object parsing, we restrict the comparison to un-/documented APIs. For the evaluation we include the standard system apps and make identical assumptions as *PScout*, i.e., we assume that any permission found for a particular API is indeed required (a more precise analysis would require path-sensitivity). Moreover, like *PScout*, we did not conduct a native code analysis.

7.1.1 Documented API map

Figure 3 shows for our documented API map (SDK EP to permissions) how often a certain permission is required. For some permissions *PScout* reports higher numbers while for others AXPLORER reports higher numbers. Since the results are fairly deviating, we manually inspected various cases, including a full analysis of NFC and bluetooth, to verify correctness of our generated numbers. *PScout*'s higher method count, particularly for the two cases of NFC and bluetooth, originates from adding package-protected methods that are not exposed to app developers and from improper handling of the `@hide` javadoc[4] attribute, resulting in an overcounting of the documented API methods. Our higher numbers of `BROADCAST_STICKY` and `SET_WALLPAPER` mainly refer to abstract methods from the `Context` class that are implemented in its subclass `ContextWrapper` and then inherited by 18 non-abstract subclasses (for API 16). Instead, *PScout* only lists those methods for the `Context/-Wrapper` class, thus missing to count the non-abstract subclasses.

Figure 4 provides a different view of the mapping by showing the distribution of required permissions per API. The main difference is the smaller number of outliers in our data set: four mappings with three or more required permissions, compared to 58 such outliers in the *PScout* data set. While the different results in Figure 3 mainly originate from technical shortcomings in the SDK analysis, Figure 4 hints at the different quality of the undocumented API map as result of a more precise

[3] We use *PScout's* results as published on their website at `http://pscout.csl.toronto.edu`. Last visited 01/25/2016.

[4] EP methods annotated with the `@hide` attribute are not included in the SDK.

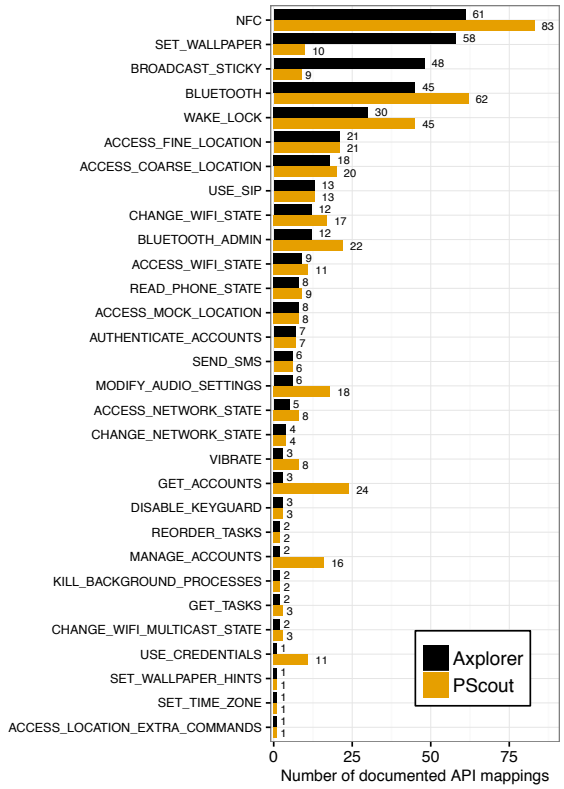

Figure 3: Number of documented APIs per permission.

framework analysis (see next Section 7.1.2). *PScout*'s more light-weight framework analysis results in an over-approximation of permission usage of EPs. For their outliers with more than five permissions in the `ConnectivityManager` class they either over-approxmiate the receivers of a `sendMessage` call and/or did not resolve the message code and the correct path in the `handleMessage` method. In such cases the over-approximation in the framework analysis negatively influences the quality of the SDK map when IPC calls from the SDK to the application framework are connected. We manually validated all outliers and found that no method actually requires more than three permissions, thus contradicting the *PScout* results. The four outliers in our dataset check at most two permissions, independent of the EP call arguments. Additional permission checks might be required for specific arguments/parameters. For instance the `setNetworkPreference(int)` function of the `ConnectivityService` will tear down a specific type of network trackers depending on a preference integer argument. Some subtypes such as the `BluetoothTetheringDataTracker` require both bluetooth permissions to execute this functionality while other subtypes require no additional permission. Adding parameter-sensitivity to the analysis is required to resolve such cases automatically and to annotate permission checks with conditions.

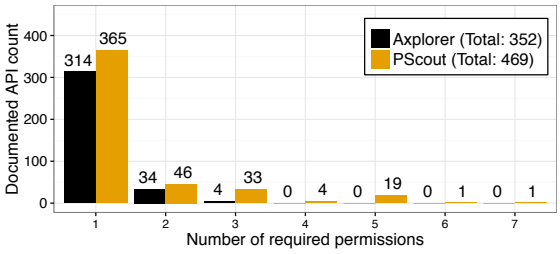

Figure 4: Number of permissions required by a documented API.

7.1.2 Undocumented API map

A fair, direct comparison of permission maps for undocumented APIs is unfortunately very difficult due to shortcomings in the original paper. Although *PScout* did not explicitly define the term *undocumented API*, we assume after manual inspection of their results that it refers to the publicly exposed framework interfaces and covers any functionality that can be called from application level (independent of whether it is provided by SDK or system apps). Hence we refer to undocumented API as the entire set of framework entry points (cf. Section 4.1).

In contrast to *PScout*'s documented API map, we discovered different inconsistencies in their undocumented mappings. Besides valid mappings from PPEP to permissions, they also include mappings for unrelated methods. First, public methods of AIDL-based entry classes (which we define as *Entry Points*) are counted up to five times: once in the SDK manager class, in the framework service class, in the AIDL interface class, and in the auto-generated `Stub` and `Proxy` classes. Second, their mapping contains methods of `StateMachine` `State` classes. `StateMachines` are used framework-internally and their functionality is not exposed to apps. Third, synthetic accessor methods as well as methods of anonymous inner classes are reported. We assume that this problem is related to the lack of a concise entry point definition that induces difficulties with the abort criteria during their backwards analysis starting from permission checks. In contrast, our forward analysis seems more suitable in this context, as permission checks are usually closely located to framework EPs.

Table 1 reports on the numbers of entry points per API level. For Android 4.1.1, Axplorer found 863 PPEP (33.4% of entry points) that require at least one permission. These numbers include signature/-

OrSystem permissions since this information, although not interesting for app developers, is of interest for understanding the Android permission model in its entirety. On average we found 1.17 permissions per PPEP, which leads to a total of 1,012 permission mappings that cover 129 distinct permissions. This is a magnitude less than the 32,304 permission mappings reported by *PScout* for normal and dangerous permissions only. However, due to our more concise definition of what constitutes public framework functionality and the inclusion of all permission levels, we argue that our number is more substantiated.

7.2 Permission Locality

The application framework implements a separation of duty: every bound service is responsible for managing a certain system resource and enforcing permissions on access by apps to them. For instance, the `LocationService` manages and protects location related information or the `PhoneInterfaceManager` facilitates and guards access to the radio interfaces. Permission strings already convey a meaning of the kind of system resource they protect and app developers might have an intuition where those permissions are required. We study whether permission checking also follows the principle of separation of duty and permissions are checked by only one particular service. We call this aspect *permission locality*. A low permission locality indicates that a certain permission is enforced at different (possibly unrelated) services. This potentially contributes to the app developer's permission *in*comprehension that can lead to overprivileged apps [32]. Moreover, a strict separation of duty, i.e., high permission locality, significantly eases the task of implementing (and verifying) authorization hooks for resources, for instance in the design of recent security APIs [21, 8]. Consequently, the permission that protects a set of sensitive operations is ideally checked only in one associated entry class.

To study the permission locality, we analyze the checked permission strings and map them to the enclosing class of the permission check call. In Android v4.1.1 (API level 16) we found that out of 110 analyzed permissions 22 (20%) are checked in more than one class. Among these permissions, 13 are checked in two classes, 5 in three classes and 4 in four classes. An example for seemingly unrelated classes are `LocationManagerService` and `PhoneInterfaceManager` that both check the dangerous permission `ACCESS_FINE_LOCATION`. While the permission is intuitively related to the first service, the connection to the latter one becomes only obvious by looking at the enclosing method that includes the check (e.g. `getCellLocation`). Interestingly, `PhoneInterfaceManager` is not a framework service but included in the telephony system *app*. Mixing framework services and system apps for enforcing identical permissions complicates permission validation and policy enforcement, since system apps might be vendor-specific. Grouping permissions by protection level results in 22.2% (12/54) of normal/dangerous permissions and 17.9% (10/56) of signature/-OrSystem permissions being checked in distinct classes. This implies that low permission locality equally affects all protection levels. Applying this analysis on API 22 results in a even lower overall permission locality. Focusing on the four outliers in API 16, changes in API 22 include three class renamings, two removals and nine additions (cf. Figure 6 in Appendix C). The permission `CONNECTIVITY_INTERNAL` more than doubled the number of classes (10) in which it is enforced. This evolution of permission checks indicates a disconcerting trend to lower permission locality.

Instead, the permission locality should be increased by, ideally, associating each permission with a single service. Once a designated owner service has been identified for each permission, a dedicated permission check function could be publicly exposed via its `Binder` interface, e.g., a method to check the `ACESSS_FINE_LOCATION` permission could be added to the `ILocationManager` interface. The addition and removal of callers to such methods then no longer affects the number of decision points and preserves the separation of duty for permission checks.

8 Discussion of Other Use-Cases

We briefly discuss further use-cases that can benefit from our work, particularly from our taxonomy of protected resources and the insights from our permission locality analysis.

Permission check inconsistencies. Prior work *Kratos* has shown that the default permission check is inconsistent and can lead to attacks [33]. However, this approach explicitly did not make the attempt to identify protected resources in Android's application framework but instead relied on arbitrary shared code as heuristic to identify security relevant hotspots in the framework's code base. While this approach has successfully demonstrated the need for such analysis, we argue that using our definition of protected resources as refinement of shared code can further improve the precision of their analysis, since, by definition, protected resources describe sensitive operations. False positives originating from shared logging

or library code are automatically eliminated then. Distilling a more concise definition of field-update and native method call resources from our high-level taxonomy is a promising future work. An example for such refinement is the removal of non-relevant field updates of a `this` reference within constructors. As there is no prior state for this object, such updates must not be flagged as protected resource.

Authorization hook placement. Different Android framework extensions [26, 46, 29, 12, 21, 8] augment the application framework with authorization hooks in a best effort approach. On commodity systems, a comparable situation for the Linux and BSD kernels has been improved through a long process that established a deeper understanding of the internal control and data flows of those kernels and that allowed development of tools to verify or automate placement of authorization hooks. A similar evolution for Android's application framework has yet been precluded due to open technical challenges: first, one must be able to analyze control and data flows in the framework across process and service boundaries; second one must be able to track the execution state of the framework service along its internal control and data flows (e.g., tracking the availability of the subject identity); third, one has to establish a clear and very specific understanding of the protected resources of each service. This work at hand addresses the first of these challenges and provides necessary permission locality information to implement comprehensive, coarse-grained enforcement models. Additionally, with our high-level taxonomy of protected resources we made a first step towards solving the third challenge.

9 Conclusion

In this paper, we studied the internals of the Android application framework, in particular challenges and solutions for static analysis of the framework, and provided a first high-level classification of its protected resources. We applied our gained insights to improve on prior results of Android permission mappings, which are a valuable input to different Android security research branches, and to introduce permission locality as a new aspect of the permission specification. Our results showed that Android permission checks violate the principle of separation of duty, which might motivate a more consolidated design for permission checking in the future. To allow app developers and independent research to benefit from our results, we published our data sets as well as lint rules for our permission mappings for the Android Studio IDE at `http://www.axplorer.org`.

Acknowledgments

This work was supported by the German Federal Ministry for Education and Research (BMBF) under project VFIT (16KIS0345) and SmartPriv (16KIS0377K) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and the initiative for excellence of the German federal government.

References

[1] Soot - Java Analysis Framework. `http://sable.github.io/soot/`, 1999.

[2] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`, 2006.

[3] Anderson, J. P. Computer security technology planning study, volume ii. Tech. Rep. ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Oct. 1972.

[4] Android Developer Documentation. Bound services. `http://developer.android.com/guide/components/bound-services.html`. Last visited: 05/08/2015.

[5] Arzt, S., Bodden, E., and Rasthofer, S. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS '14)* (2014), The Internet Society.

[6] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI 2014)* (2014).

[7] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: Analyzing the android permission specification. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)* (2012), ACM.

[8] Backes, M., Bugiel, S., Gerling, S., and von Styp-Rekowsky, P. Android Security Framework: Extensible multi-layered access control on Android. In *Proc. 30th Annual Computer Security Applications Conference (ACSAC '14)* (2014), ACM.

[9] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. 24th USENIX Security Symposium (SEC '15)* (2015), USENIX.

[10] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard - enforcing user requirements on Android apps. In *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)* (2013).

[11] Blackshear, S., Gendreau, A., and Chang, B.-Y. E. Droidel: A general approach to android framework modeling. In *Proc. ACM SIGPLAN Workshop on State of the Art in Program Analysis (SOAP'15)* (2015), ACM.

[12] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. 22nd USENIX Security Symposium (SEC '13)* (2013), USENIX Association.

[13] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS '15)* (2015), ISOC.

[14] Chaudhuri, A., Fuchs, A., and Foster, J. SCanDroid: Automated security certification of Android applications. Tech. Rep. CS-TR-4991, University of Maryland, 2009.

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

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

[17] Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. 5th international conference on Trust and Trustworthy Computing (TRUST '12)* (2012), Springer-Verlag.

[18] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. Information flow analysis of android applications in DroidSafe. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS '15)* (2015), ISOC.

[19] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. Checking app behavior against app descriptions. In *Proc. 36th International Conference on Software Engineering (ICSE '14)* (2014), pp. 1025–1035.

[20] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12)* (2012), ACM.

[21] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC '14)* (2014), USENIX.

[22] Huang, H., Zhu, S., Chen, K., and Liu, P. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)* (2015), ACM.

[23] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *Proc. 2nd ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '12)* (2012), ACM.

[24] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and Mcdaniel, P. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proc. 37th International Conference on Software Engineering (ICSE '15)* (2015).

[25] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)* (2012), ACM.

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

[27] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proc. 37th International Conference on Software Engineering (ICSE '15)* (2015).

[28] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proc. 22Nd USENIX Conference on Security (SEC '13)* (2013), USENIX Association.

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

[30] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. Whyper: Towards automating risk assessment of mobile applications. In *Proc. 22nd USENIX Security Symposium (SEC '13)* (2013), USENIX.

[31] Pearce, P., Porter Felt, A., Nunez, G., and Wagner, D. AdDroid: Privilege separation for applications and advertisers in Android. In *Proc. 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)* (2012), ACM.

[32] Porter Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)* (2011), ACM.

[33] Shao, Y., Ott, J., Chen, Q. A., Qian, Z., and Mao, Z. M. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS '16)* (2016), ISOC.

[34] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: Separating smartphone advertising from applications. In *Proc. 21st USENIX Security Symposium (SEC '12)* (2012), USENIX Association.

[35] Song, D., Zhao, J., Burke, M. G., Sbirlea, D., Wallach, D., and Sarkar, V. Finding tizen security bugs through whole-system static analysis. *CoRR abs/1504.05967* (2015).

[36] Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y. Autoises: Automatically inferring security specifications and detecting violations. In *Proc. 17th USENIX Security Symposium (SEC '08)* (2008), USENIX.

[37] The Android Open-Source Project. Security-Enhanced Linux in Android. `http://source.android.com/devices/tech/security/selinux/index.html`. Last visited: 07/27/2015.

[38] Vidas, T., Christin, N., and Cranor, L. F. Curbing android permission creep. In *Proc. Workshop on Web 2.0 Security and Privacy 2011 (W2SP 2011)* (2011).

[39] Watson, R., Morrison, W., Vance, C., and Feldman, B. The TrustedBSD MAC Framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. FREENIX Track: 2003 USENIX Annual Technical Conference* (2003), USENIX Association.

[40] Wei, F., Roy, S., Ou, X., and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. 21th ACM Conference on Computer and Communication Security (CCS '14)* (2014), ACM.

[41] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux Security Modules: General security support for the Linux kernel. In *Proc. 11th USENIX Security Symposium (SEC '02)* (2002), USENIX Association.

[42] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The impact of vendor customizations on android security. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)* (2013), ACM.

[43] Yang, Z., and Yang, M. Leakminer: Detect information leakage on Android with static taint analysis. In *Proc. 2012 Third World Congress on Software Engineering (WCSE '12)* (2012), IEEE Computer Society.

[44] Zhang, X., Edwards, A., and Jaeger, T. Using cqual for static analysis of authorization hook placement. In *Proc. 11th USENIX Security Symposium (SEC' 02)* (2002), USENIX.

[45] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)* (2013), ACM.

[46] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. Taming information-stealing smartphone applications (on Android). In *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST '11)* (2011), Springer-Verlag.

Appendix

A Entry Class Interconnection

A standard call graph gives information about the enclosing method/class of function calls. However, this information is insufficient to provide information about the interconnection of framework entry classes. Instead, the interesting information is the originating entry class that leads to an RPC rather than the actual class that encloses the RPC. To provide better information on the RPC dependencies of entry classes, AXPLORER creates an RPC map by recording RPCs to other entry classes and mapping them to the original entries during control-flow slicing. Figure 5 shows a subgraph of the overall RPC interconnections between flows from different entry classes on Android 5.1 that AXPLORER generated. Nodes correspond to entry classes and are weighted by in-/out-degree, thus highlighting highly-dependent classes such as `ActivityManagerService`. The source of a directed edge is the originating entry class: the control-flow starts at an entry method of this class and at some point along the flow, not necessarily in the same class, an RPC to the class of the edge target node is invoked.

Across all four investigated Android versions there is a median number of three distinct RPC receivers per entry class in our map. The `ActivityManagerService` is an exceptional case where flows from its entry methods reach 36 different entry classes. These numbers emphasize that large parts of the framework are strongly connected and that detailed knowledge about the communication behavior greatly simplifies further framework analyses as explained above.

B Evaluation of protected resources

B.1 Statistics on protected resources

Table 2 provides absolute numbers of protected resources by API version and the distribution by resource types.

B.2 Manual investigation of RTE

Due to the surprisingly high number of runtime exceptions and the fact that uncaught RTE might potentially crash the system, we manually investigated the corresponding code locations to better understand their security implications. The

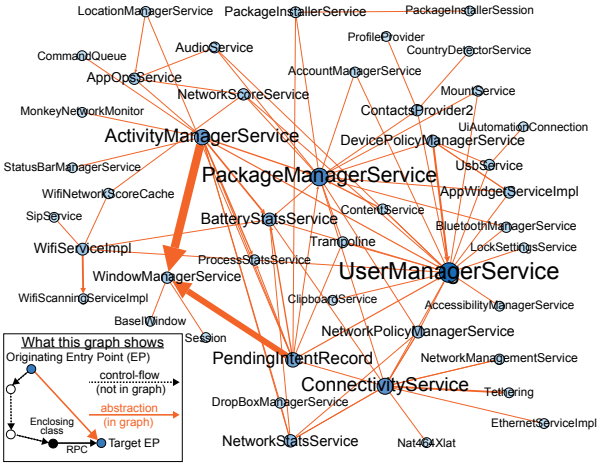

Figure 5: Subgraph of overall entry class interconnection via RPCs in Android 5.1. Directed edges show an RPC to an exposed `IInterface` method of target node.

source code analysis revealed that these instructions reside both in Android's code base and in external library code. Android, similar to other operating systems, relies on external libraries for specific tasks, such as XMLPullParser, Bouncy Castle, and J-SIP. Integrating library code into such a complex system usually raises the question on how to handle unchecked exceptions that are thrown by library functions. Either the library code is patched, which might be a tedious maintenance task, or exceptions are caught at caller site. Although the latter case is considered bad practice in case of runtime exceptions, using generic catch handler around library call sites is the easiest and most reliable approach in this situation.

Runtime exceptions in the Android code are thrown to indicate precondition violations, like crucial class fields being null, unrecoverable IO errors, or security violations. The implications of a runtime exception differ with respect to code location in which they are thrown. Exceptions in the application layer, i.e. in system applications, cause the respective app to crash. Similarly as for normal apps, they can be restarted by the user. Sticky system app services are restarted automatically by the system. Exceptions thrown in bound services of the `SystemServer` are handled in a special way: A system watchdog thread `com.android.server.Watchdog` constantly monitors the core services like `PowerManagerService` and `ActivityManagerService`. In case of a crash or deadlock, it reboots the entire system. Uncaught runtime exceptions in unmonitored system services cause the Android Runtime, including Zygote and

Android version	4.1.1 (16)		4.2.2 (17)		4.4.4 (19)		5.1 (22)	
# of protected resources	6,490		6,969		7,488		10,044	
- field updates	4,891	(75.36%)	5,268	(75.59%)	5,675	(75.79%)	7,520	(74.87%)
- native method calls	1,433	(22.08%)	1,499	(21.51%)	1,643	(21.94%)	2,305	(22.95%)
- return value	51	(0.79%)	60	(0.86%)	54	(0.72%)	69	(0.69%)
- broadcast sender	65	(1.00%)	72	(1.03%)	53	(0.71%)	78	(0.78%)
- throw runtime exception	50	(0.77%)	70	(1.01%)	63	(0.84%)	72	(0.71%)
Median context depth	8		9		11		9	

Table 2: Numbers on protected resources by type and Android version.

the `SystemServer` to be restarted (hot reboot) while the kernel keeps running—effects also described in recent research [22].

C Statistics on permission locality

Table 3 shows a detailed statistic about the number of permissions in API 16 that are checked in more than one class, grouped by their protection level. These numbers imply that there is no apparent correlation about the relative frequency and the protection level, in particular between the permissions available to third-party apps (normal+dangerous) and the ones reserved for the system. Hence, low permission locality cases occur across all permission protection levels.

Protection level	# permissions	
normal	2/16	(12.5%)
dangerous	10/38	(26.3%)
signature	2/25	(8%)
signatureOrSystem	8/31	(25.8%)

Table 3: Number of permissions in API 16 that are checked in more than one class grouped by permission protection level.

Figure 6 lists all four permissions that are checked in four distinct entry classes in API level 16 (colors denote changes in API 22). In case of `READ_PHONE_STATE`, those four classes even reside in four distinct packages of which one is part of the telephony system app. Although it may not always be easy to identify one dedicated service as the permission owner, an effort might be desirable to centralize permission checks into as few services as possible (in best case into a single service). Besides renamed classes (blue color) as result of a code refactoring process, the number of additions and removal for this small number of examples clearly confirms that permission checks are violating the separation of duty and underline the need for central enforcement points.

```
Permission : ACCESS_NETWORK_STATE
Level      : normal
Checked in :
  - com.android.server.ConnectivityService
  - com.android.server.ethernet.EthernetServiceImpl
  - com.android.server.ThrottleService
  - com.android.server.net.NetworkPolicyManagerService
  - com.android.server.net.NetworkStatsService

Permission : READ_PHONE_STATE
Level      : dangerous
Checked in :
  - com.android.internal.telephony.PhoneSubInfoProxy
  - com.android.phone.PhoneInterfaceManager
  - com.android.internal.telephony.SubscriptionController
  - com.android.server.TelephonyRegistry
  - com.android.server.net.NetworkPolicyManagerService

Permission : CONNECTIVITY_INTERNAL
Level      : signatureOrSystem
Checked in :
  - com.android.server.ConnectivityService
  - com.android.server.NetworkManagementService
  - com.android.server.NsdService
  - com.android.server.net.NetworkPolicyManagerService
  - com.android.server.net.NetworkStatsService
  - com.android.server.ethernet.EthernetServiceImpl
  - com.android.server.connectivity.Tethering
  - com.android.bluetooth.pan.PanService$BluetoothPanBinder
  - com.android.server.wifi.WifiServiceImpl
  - com.android.server.wifi.p2p.WifiP2pServiceImpl

Permission : UPDATE_DEVICE_STATS
Level      : signatureOrSystem
Checked in :
  - com.android.server.power.PowerManagerService$BinderService
  - com.android.server.LocationManagerService
  - com.android.server.am.BatteryStatsService
  - com.android.server.wifi.WifiServiceImpl
  - com.android.server.am.UsageStatsService
```

Figure 6: Permissions checked in four distinct classes in API 16. Colors denote changes in API 22: renamed classes (blue), additions (green) and removals (red).