ARF: Identifying Re-Delegation Vulnerabilities in Android System Services

Sigmund Albert Gorski III
North Carolina State University
sagorski@ncsu.edu

William Enck
North Carolina State University
whenck@ncsu.edu

ABSTRACT

Over the past decade, the security of the Android platform has undergone significant scrutiny by both academic and industrial researchers. This scrutiny has been largely directed towards third-party applications and a few critical system interfaces, leaving much of Android's middleware unstudied. Building upon recent efforts to more rigorously analyze authorization logic in Android's system services, we revisit the problem of permission re-delegation, but in the context of system service entry points. In this paper, we propose the Android Re-delegation Finder (ARF) analysis framework for helping security analysts identify permission re-delegation vulnerabilities within Android's system services. ARF analyzes an interconnected graph of entry points in system services, deriving calling dependencies, annotating permission checks, and identifying potentially vulnerable deputies that improperly expose information or functionality to third-party applications. We apply ARF to Android AOSP version 8.1.0 and find that it refines the set of 15,483 paths between entry points down to a manageable set of 490 paths. Upon manual inspection, we found that 170 paths improperly exposed information or functionality, consisting of 86 vulnerable deputies. Through this effort, we demonstrate the need for continued investigation of automated tools to analyze the authorization logic within the Android middleware.

ACM Reference Format:
Sigmund Albert Gorski III and William Enck. 2019. ARF: Identifying Re-Delegation Vulnerabilities in Android System Services. In *12th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '19), May 15–17, 2019, Miami, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3317549.3319725

1 INTRODUCTION

Android has become a dominant computing platform, running on billions of devices world-wide. Android-based phones and Internet-of-Things devices are used for entertainment, social networking, finance, and business, among many other purposes. As security vulnerabilities in the Android platform have widespread impact, it has been the subject of a significant amount of security research for the last decade [3, 35].

At the core of Android's security model is a permission framework that prevents third-party applications from performing malicious or privacy-invasive actions. Android's permissions have been well studied [12, 13, 41]. Moreover, research has studied the usability of permission notifications [16, 17], contrasted permissions with application descriptions [33, 34], and mapped application programming interfaces (APIs) to the permissions that protect them [2, 5, 6, 15]. More recently, research has used consistency analysis to identify missing permission and related authorization checks within system services [1, 23, 36].

Most relevant to this work is *permission re-delegation* [18]. Conceptually, Android permissions are a form of capability. However, permissions are primarily static capabilities and do not directly support delegation (with a few exceptions related to content provider URIs). To date, existing work studying permission re-delegation has focused on applications. For example, Davi et al. [8], Felt et al. [18], and Quire [9], consider scenarios where a third-party application with permission p creates an API that allows other third-party applications without permission p to indirectly call a system API protected with permission p. Woodpecker [24] and SEFA [38] extended this analysis to system applications (e.g., dialer, contacts) and applications bundled with the firmware by an OEM.

The goal of this paper is to investigate permission re-delegation within core system services. Android's APIs provide wrappers around an inter-process communication (IPC) mechanism called *binder*. Every system service exposes a broad set of binder remote procedure call (RPC) interfaces, which recent literature terms *entry points*. Not all entry points are accessible via APIs, but any third party application can use reflection to attempt to call any entry point directly (though permission checks or other authorization checks may deny access). This paper studies permission re-delegation with respect to these entry points.

Prior work studying authorization checks in system services [1, 23, 36] only considered the logic of the first entry point called by a third-party application. However, entry points commonly call other entry points, creating complex inter-dependencies among one another. These inter-dependencies reveal the challenge of static capabilities. Consider the case where third party application a calls entry point e_1, which in turn calls entry point e_2. Further, consider the case where the authorization logic allows application a to call e_1, but not e_2. In this case, e_1 is privileged (i.e., a deputy) and therefore can call e_2. Sometimes this functionality is desired (e.g., e_1 sanitizes and endorses requests from application a before calling e_2). However, other times e_1 will improperly expose e_2's functionality to application a. This improper exposure is the focus of this paper.

In this paper, we propose the Android Re-delegation Finder (ARF), an analysis framework for identifying permission re-delegation vulnerabilities within Android's system services. ARF identifies

WiSec '19, May 15–17, 2019, Miami, FL, USA

ACM ISBN 978-1-4503-6726-4/19/05…$15.00
https://doi.org/10.1145/3317549.3319725

potentially vulnerable deputies by (1) constructing an interconnected graph of entry points, (2) annotating entry points with their permission checks, and (3) identifying when a deputy is less restrictive than another entry point. This list of potentially vulnerable deputies is further refined by using a combination of insights to eliminate instances that are highly unlikely to be vulnerabilities. In doing so, ARF reduces the set of potentially vulnerable deputies to a size reviewable via manual inspection.

We applied ARF to the Android system services in AOSP version 8.1.0_r1 (API 27). Of the 15,483 entry point paths in the interconnected graph, our analysis identified 490 requiring manual inspection. The manual inspection identified 86 unique deputies with at least one vulnerability. These vulnerabilities include system state modification, information leaks, usability disruption, failures to enforce multi-user separation, and the ability to keep the device from sleeping. Additionally, of the 86 vulnerable deputies, 14 were not related to re-delegation, but were weaknesses stemming from missing authorization checks in the deputies. The discovered vulnerabilities have been reported to Google.

The remainder of this paper proceeds as follows. Section 2 provides background and motivation. Section 3 overviews ARF. Section 4 describes ARF's design. Section 5 evaluates ARF and presents our findings. Section 6 discusses limitations. Section 7 surveys related work. Section 8 concludes.

2 BACKGROUND AND MOTIVATION

While Android uses the Linux kernel, it differs significantly from GNU/Linux distributions. The Android middleware is built using the same four component abstractions as third-party applications: activity, broadcast receiver, content provider, and service. This paper focuses specifically on service components, which provide daemon-like functionality. Interested readers are referred to prior work [13] for detailed discussion of component functionality.

The core Android framework is built upon a set of more than 100 *system services* accessible to third-party apps (e.g., ActivityManagerService, PackageManagerService, and LocationManagerService). Each service defines a set of remote procedure call (RPC) interfaces, which are invoked using Android's *binder* inter-process communication (IPC) mechanism. At a low level, a process constructs a *parcel* message that specifies the target service method identifier (an integer) and the method arguments. On the server side, the parcel is demultiplexed using the method identifier to invoke the corresponding *entry point* in the service. This entry point can determine the identity of the caller via system methods that return contextual state (e.g., getCallingUid).

We build upon the ACMiner [23] program analysis framework, which is designed to identify inconsistent authorization checks in Android's system services. ACMiner identifies service entry points and constructs a call graph using Class Hierarchy Analysis (CHA). It annotates each entry point with a flow-insensitive set of authorization checks, which includes both permission checks and other security-relevant authorization checks. To do so, ACMiner defines two concepts: (1) a *context query* is a system method that either encompasses authorization checks or returns data used in authorization checks (e.g., checkPermission) and (2) a *control predicate* is any conditional check that leads to an authorization denial. To determine which conditional checks are related to authorization logic, ACMiner uses a semi-automated algorithm based on the textual values of strings, variable names, and method names. Readers are referred to the ACMiner paper [23] for additional details. Of specific note, ACMiner's call graph analysis of an entry point *stops* when it reaches another entry point, regardless of the service to which the entry point belongs.

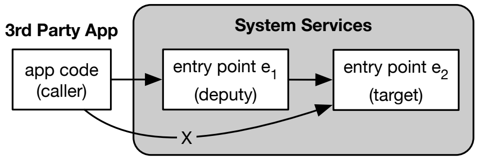

Figure 1: The deputy loses the caller's identity and may improperly expose the target's functionality to third-party applications.

```
void whitelistAppTemporarily(String package, long d,
    int userId) {
  StringBuilder reason = new StringBuilder("from:");
  UserHandle.formatUid(reason, Binder.getCallingUid());
  mDeviceIdleController.addPowerSaveTempWhitelistApp(
    package, d, userId, reason.toString());
}
```

Figure 2: Entry point whitelistAppTemporarily performs no checks before calling entry point addPowerSaveTempWhitelistApp, which requires the system-level permission CHANGE_DEVICE_IDLE_TEMP_WHITELIST and is not accessible third-party applications.

Problem Statement: Consider the scenario in Figure 1, where application a calls a system service with entry point e_1, which in turn calls entry point e_2 (which may or may not be in the same service as e_1). The authorization checks allow application a to directly call entry point e_1 but not entry point e_2. When the execution enters entry point e_1, the code can determine the calling identity and use methods such as getCallingUid to evaluate authorization checks (e.g., checkPermission). However, when execution transitions from entry point e_1 to entry point e_2, application a's calling identity is lost (i.e., getCallingUid returns the UID of the system service). Therefore, if the authorization checks of entry point e_1 are not at least as restrictive as entry point e_2, then information or functionality may be improperly leaked to application a. However, there is one caveat to this definition. Whenever entry point e_1 and entry point e_2 share the same process (e.g., they are in the same service), unless explicitly cleared by e_1, the calling identity remains the same for both entry points. Section 4.3.1 provides more details of this problem as well as the solution ARF applies to filter out these non-vulnerable paths.

In classic terms, entry point e_1 in Figure 1 is a privileged deputy. Deputies are necessary in systems without capabilities (or those with only static capabilities). They act as gatekeepers that provide sanity checks (i.e., endorsement) before calling security-sensitive operations. While deputies are not inherently bad, they are often the source of security problems (e.g., confused deputies [25]). The goal of this paper is to identify deputies that improperly leak information or functionality to third-party applications.

Motivating Example: Figure 2 provides a motivating example identified by ARF. Entry points whitelistAppTemporarily and addPowerSaveTempWhitelistApp are in the services UsageStatsService and DeviceIdleController respectively. As shown in the figure, whitelistAppTemporarily calls addPowerSaveTempWhitelistApp without performing any authorization checks or service-specific endorsement. Entry point `addPowerSaveTempWhitelistApp` (definition not shown) grants its callers the ability to access the network and keep the device awake. It requires that the caller has the `CHANGE_DEVICE_IDLE_TEMP_WHITELIST` permission, which has the "system" protection-level and is not grantable to third-party applications. Therefore, a third-party application cannot call `addPowerSaveTempWhitelistApp`, but by calling `whitelistAppTemporarily`, it can achieve the same functionality.

3 OVERVIEW

As described in Section 2, Android generally loses calling context when one entry point (i.e., deputy) calls another entry point (i.e., target). The goal of this paper is to identify deputies that improperly expose information or functionality from targets. As deputies are not inherently insecure, ARF seeks to minimize the effort required by a domain expert to identify improper re-delegation of access. To do so, we must overcome the following research challenges.

- *There are multiple ways to perform the same check.* Android frequently uses shortcuts to grant access to special callers. Therefore, the deputy and target may semantically perform the same checks but use different code.
- *Android's system service entry points are heavily interconnected.* There are tens of thousands of unique paths through the entry points. We must reduce the set of potentially vulnerable paths to a reasonable size for domain expert review.

We address the first research challenge with the following key insights. Android primarily uses permission checks to authorize third-party applications. Therefore, when an entry point has permission checks, the other authorization checks are present as shortcuts to grant quick access for special callers. When an entry point does not have permission checks, but does have other authorization checks, it is only accessible to special callers (which are not third-party applications). This insight allows ARF to primarily use permission checks when detecting authorization re-delegation.

We address the second research challenge by observing that ARF only needs to analyze single edges between entry points. For example, consider the path $e_1 \rightarrow e_2 \rightarrow e_3$. If ARF identifies that entry point e_1 is less restrictive than entry point e_2, then a vulnerability in entry point e_1 may expose the information or functionality in both entry point e_2 and entry point e_3. The vulnerability results because all system entry points are privileged, and hence act as deputies. Note that ARF must also check if entry point e_2 is at least as restrictive as e_3 because, by definition, all entry points are callable by third-party applications. Finally, ARF uses a number of heuristics stemming from domain knowledge to eliminate paths that would not contain re-delegation vulnerabilities.

Figure 3 overviews ARF's five phases. (1) ARF obtains from ACMiner [23] the authorization checks and entry point interconnecting relationships for all entry points in the system services. (2) ARF then determines which of these entry points are callable by third-party applications by identifying the accessible services. (3) Next, ARF proceeds to identify all paths containing only two entry points that represent potential permission re-delegation vulnerabilities. These paths are output to be verified by a domain expert. (4) After the verification, those paths identified as vulnerable are then fed back into ARF as input. ARF is then run a second time to identify all paths that represent potential permission re-delegation vulnerabilities and contain at least one of the vulnerable sub-paths from (3). Once again these paths are output to be reviewed by a domain expert. (5) The verified vulnerable paths from phases (3) and (4) are then combined together as the final output of ARF.

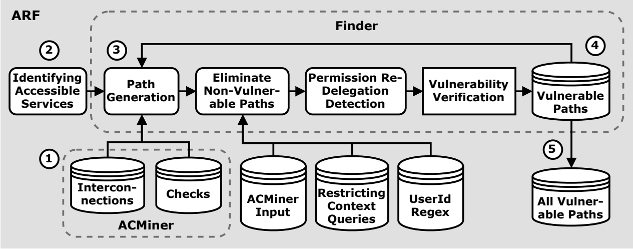
ARF
Finder
2
Identifying Accessible Services
3
Path Generation
Eliminate Non-Vulner-able Paths
Permission Re-Delegation Detection
Vulnerability Verification
4
Vulnerable Paths
1
Intercon-nections
Checks
ACMiner
ACMiner Input
Restricting Context Queries
UserId Regex
5
All Vulner-able Paths

Figure 3: ARF's input data and analysis stages.

4 DESIGN

The Android system is segmented into many different services that provide binder-based entry points for returning information and performing functionality. As described in Section 2, entry points are highly inter-dependent resulting in many privileged deputies. This design can introduce permission re-delegation vulnerabilities that improperly expose information or functionality to unprivileged third-party applications.

We designed ARF to detect permission re-delegation vulnerabilities in Android system services. ARF is constructed using the Soot [28, 37] Java static analysis framework, as well as authorization checks mined by ACMiner [23]. This section proceeds as follows: (§4.1) we formally define permission re-delegation and related terms, (§4.2) we describe ARF's detection methodology, and (§4.3) we detail techniques to eliminate paths that are highly unlikely to be re-delegation vulnerabilities.

4.1 Permission Re-Delegation

This section formally defines *permission re-delegation* and related terms for their use in ARF. For these definitions, let:

E := set of all entry points in the Android system,
A := set of all possible permission (access) checks, and
$\mathcal{A}(\cdot)$:= $E \rightarrow \mathbb{P}(A)$, set of all permission checks for some entry point, where $\mathbb{P}(A)$ is the power set of A.

Next, to formalize the definition of permission re-delegation we define the key terms *less restrictive* and *path*.

Definition 1 (*Less Restrictive*). Entry point e_1 is *less restrictive* than e_2 ($e_1 \sqsubseteq e_2$) if $\mathcal{A}(e_1) \not\supseteq \mathcal{A}(e_2)$ (i.e., e_1 is missing one or more of the permission checks of e_2).

Definition 2 (*Path*). For entry points $e_1, e_2 \in E$, $e_1 \neq e_2$, there is a *path* between e_1 and e_2 ($e_1 \rightsquigarrow e_2$) if either of the following conditions hold: (1) the call graph of e_1 directly invokes e_2 or (2) the call graph of e_1 indirectly calls e_2 by invoking entry point $e_i \in E$, $e_i \neq e_1$ and $e_i \neq e_2$, for which there is a path to e_2 ($e_i \rightsquigarrow e_2$).

By definition, a path must contain at least two entry points but may contain as many entry points as $|E|$. To differentiate these situations, we will refer to the former as *single edge paths* and the later as *multiple edge paths*.

Before defining permission re-delegation, we must formally define *deputy* and *target*.

Definition 3 (*Deputy*). An entry point $e_1 \in E$ is a *deputy* if $\exists e_2 \in E$ such that $e_1 \neq e_2$ and $e_1 \rightsquigarrow e_2$. Put plainly, a *deputy* is the first entry point in a *path*.

Definition 4 (*Target*). An entry point $e_2 \in E$ is a *target* if $\exists e_1 \in E$ such that $e_1 \neq e_2$ and $e_1 \rightsquigarrow e_2$. Put plainly, a *target* is the last entry point in a *path*.

Finally, we can now define permission re-delegation.

Definition 5 (*Permission Re-Delegation*). If $\exists e_1, e_2 \in E$ such that $e_1 \neq e_2$, $e_1 \rightsquigarrow e_2$, and $e_1 \sqsubseteq e_2$ then permission re-delegation occurs. Put plainly, the *deputy* e_1 reaches the *target* e_2 without performing all the permission checks of the *target* on the caller of the *deputy*, permitting the caller of the *deputy* access to the *target* as if they were the *deputy*.

4.2 Design Considerations

The following section outlines the technical considerations behind the design of ARF. We begin by explaining why we target only permission re-delegation instances that can be exploited by third-party apps in Section 4.2.1. Then we detail how ARF identifies services that are accessible to third-party apps in Section 4.2.2. In Section 4.2.3, we discuss our reasoning for limiting ARF to the detection of permission re-delegation instances. Finally, in Section 4.2.4, we explain how ARF reduces the number of reported permission re-delegation instances a domain expert needs to review from the tens of thousands to just a few thousand.

4.2.1 Third-Party Access. It is possible for permission re-delegation to be exploited by both third-party applications and other system entities (e.g., system applications). However, ARF focuses only on permission re-delegation that can be exploited by third-party apps. For an application or service to be considered part of the system, it must be included as part of the signed system image at build time. For the purposes of this paper, we assume these applications are benign. Devices that include malicious applications directly from the OEM are outside the scope of this paper and represent larger security concerns that should be addressed via other means. Finally, while permission re-delegation vulnerabilities in system applications are possible, prior work [24, 38] proposes solutions.

4.2.2 Identifying Accessible Services. ARF uses ACMiner [23] to identify all the services and their entry points that may be accessible through binder IPC. Not every binder-capable service is accessible to third-party applications. Only services that register with the `ServiceManager` are accessible. As shown in Figure 3, ARF identifies when system services register with the `ServiceManager` by statically analyzing the Android system using Soot [28, 37]. Only the identified registered services are considered by ARF for possible permission re-delegation vulnerabilities.

To identify the services registered with the `ServiceManager`, we first used the fact that the `ServiceManager` contains two methods with the name `addService`, which must be called to register a service. We then examined the Android system source code to identify any other methods that provide the same functionality (i.e., that wrap the two core `addService` methods). This examination identified an additional two methods in the `SystemService` class with the name `publishBinderService`, which can be used to register a service with the `ServiceManager`. Using this knowledge, ARF analyzes the system service code to identify invocations of the four register methods. From the arguments for each invoke statement, ARF retrieves two values: (1) the `String` id for the service, which can be used to get the service from the `ServiceManager` and (2) the reference to an object representing the actual service. Next, ARF performs several interprocedural and intra-procedural analyses on the object references and their definition statement(s) in an attempt to resolve the object references to concrete types (i.e., the name of the class of the service being registered).

For object references that cannot be resolved to concrete types, ARF performs additional analysis on the methods used to retrieve services from the `ServiceManager`, specifically `getService` and `getServiceOrThrow` in the `ServiceManager`, `getBinderService` in the `SystemService` class, and `getService` in the `FrameworkFacade` class. These methods were discovered in a manner similar to the add services methods. Using the `String` ids of the remaining unidentified object references, ARF analyzes the system services code to identify all statements that invoke the four get methods with one of the `String` ids as an argument. As services retrieved from the `ServiceManager` must be cast to the appropriate type before invoking any entry points through binder IPC, ARF is able to infer the concrete type of the remaining object references from the cast expressions used on the object returned by the get methods.

4.2.3 Focusing on Permission Re-Delegation. As discussed in ACMiner [23], the Android system is complex and contains many different types of authorization checks (e.g., those involving `UID`, `PID`, `GID`, `AppId`, `UserId`, and package name) along with the standard permission checks. However, these other authorization checks are primarily used internally by Android system entities to allow access by other system entities. Permission checks remain the dominant way third-party apps are granted access to entry points that return information or perform functionality.

Using this insight, ARF only needs to identify permission re-delegation, a subset of the authorization re-delegation problem. To perform this optimization, ARF only selects the permission checks in the set of authorization checks identified by ACMiner for each entry point. In other words, when evaluating a path for the possibility of permission re-delegation, entry points without any permission checks are treated as if they contain no authorization checks and those with permission checks are treated as if the permission checks are the only authorization checks. Note that entry points with non-permission authorization checks may be targets of re-delegation vulnerabilities not identified by ARF. Section 6 discusses this limitation in further detail.

Finally, ARF's analysis focuses on deputies that only check permissions that can be granted to third-party applications (i.e., permissions with a protection level of *normal*, *dangerous*, *instant*, *runtime*, or *pre23*). A deputy that requires permissions not accessible to third-party applications is unlikely to be accessible via other means. This observation eliminates paths that are inaccessible to third-party applicationss and thus do not constitute permission re-delegation.

4.2.4 Multi-Edge Paths. ARF generates paths using the entry point reaching relationships output by ACMiner [23]. These relationships detail only the entry points directly calling other entry points (i.e., single edge paths). However, this information can be used to generate multiple edge paths, as multiple edge paths are a combination of single edge paths.

Our goal when designing ARF was to detect as many instances of permission re-delegation as possible, as any may be a vulnerability in the Android system. Our initial design analyzed all possible paths (i.e., both multiple edge paths and single edge paths). When analyzing AOSP 8.1.0, this design resulted in 15,483 paths containing permission re-delegation. It is unreasonable to perform a manual inspection to identify vulnerabilities for this many paths.

To reduce the analysis scope, we observed that the analysis does not need to consider all possible paths. As a multiple edge path is the combination of some number of single edge paths, if none of the single edge paths comprising a multiple edge path are a permission re-delegation instance, then the multiple edge path cannot result in a permission re-delegation vulnerability.

For example, consider the path $e_1 \rightsquigarrow e_3$ defined as $e_1 \rightarrow e_2 \rightarrow e_3$. By Definitions 1 and 5, we know that if the multiple edge path is not a permission re-delegation instance, then $e_1 \not\sqsubseteq e_3 = \mathcal{A}(e_1) \supseteq \mathcal{A}(e_3)$. The expression $\mathcal{A}(e_1) \supseteq \mathcal{A}(e_3)$ can be further expanded to $\mathcal{A}(e_1) \supseteq \mathcal{A}(e_2) \wedge \mathcal{A}(e_2) \supseteq \mathcal{A}(e_3)$ since $e_1 \rightsquigarrow e_3$ is comprised of the single edge paths $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$. By negating the expression, we find $e_1 \sqsubseteq e_3 = \mathcal{A}(e_1) \not\supseteq \mathcal{A}(e_2) \vee \mathcal{A}(e_2) \not\supseteq \mathcal{A}(e_3)$, which implies that for path $e_1 \rightsquigarrow e_3$ to be a permission re-delegation instance, at least one of its single edge paths must also be a permission re-delegation instance.

We use this insight to reduce the analysis scope by splitting our analysis into two phases. First, ARF only analyzes single edge paths for permission re-delegation. Then manual inspection verifies if the reported permission re-delegation instances constitute a vulnerability. Second, we take the single edge paths that are vulnerable and rerun ARF on all multiple edge paths that contain the vulnerable single edge paths. Again, manual inspection is used to verify if the reported permission re-delegation instances constitute a vulnerability. Using this method, we significantly reduced the number of paths containing a permission re-delegation instance that required manual review. For AOSP 8.1.0, only 2,816 required manual inspection, an almost 82% reduction.

4.3 Eliminating Non-Vulnerable Paths

While an 82% reduction is significant, reviewing 2,816 permission re-delegation instances still requires considerable manual effort. To further reduce the effort a domain expert would need to expend to identify vulnerable paths, we manually reviewed all 2,816 paths in a systematic manner. First, for each path, we identified the general behavior of the target (i.e., if it returns data, performs some function, or both). Next, we studied the deputies to determine if it is possible to access their targets in some meaningful way (i.e., retrieve all or a portion of the data being returned and/or manipulate the functionality of the target from the deputy). If it seemed as if the deputy might be exposing the data or functionality of the target, we set them aside for later review. Otherwise, we studied the non-vulnerable paths to determine how we might be able to automate their elimination. Overall, the process took approximately 92 hours.

Through this analysis, we discovered that the majority of the non-vulnerable permission re-delegation instances could be divided into the seven general categories: paths where (§4.3.1) the deputy and target have an equivalent calling identity, (§4.3.2) the deputy or target are authorization checks, (§4.3.3) the target is used in an authorization check, (§4.3.4) the deputy is restricted to a special caller, (§4.3.5) the deputy does not operate across users, (§4.3.6) the deputy correctly handles the multi-user enforcement already, and (§4.3.7) the target is a result of backwards compatibility code that can never be called by system services. Using Soot [28, 37] and the output of ACMiner [23], we developed techniques to programmatically eliminate the non-vulnerable paths of each category. Figure 3 illustrates where the noise elimination phase occurs in ARF. We detail the seven techniques used by ARF below.

4.3.1 Equivalent Calling Identity. Within the Android system, authorization checks are usually conducted using the calling identity of the the most immediate caller of an entry point. For example, in Figure 1, the deputy's authorization checks would be performed on the caller and the target's authorization checks would be performed on the deputy only. This is a result of the cross entry point interactions being conducted through binder IPC which only stores the calling identity of the entry point occurring immediately before the current entry point in a path. However, when two entry points share the same process, it is possible for cross entry point interactions to occur without going through binder IPC. For such cross entry point interactions, unless an entry point explicitly clears the calling identity using `clearCallingIdentity` in the `Binder` class before calling another entry point, both entry points will conduct authorization checks using the same calling identity, eliminating permission re-delegation vulnerabilities. To detect such paths, ARF uses the fact that cross entry point interactions within the same process only ever occur when an entry point calls another (1) directly within the same service and (2) indirectly through a reference to the object of another service stored in a field of the calling service. As such, all single edge paths that match the criteria above and that do not call `clearCallingIdentity` before calling the target are excluded by ARF.

4.3.2 Deputy or Target is an Authorization Check. A number of entry points are solely used as context queries (e.g., `checkPermission` in `ActivityManagerService` and `PackageManagerService`). When these entry points occur as the target of some path, they are not part of the functionality and instead are responsible for restricting access to the deputy. Similarly, when these entry points occur as the deputy, all targets become a part of the larger authorization check and thus cannot be used to expose functionality or data. In either case, paths involving entry points that are also authorization checks will never constitute an permission re-delegation. As such, ARF excludes all paths that contain a deputy or target that is also a context query, as defined by ACMiner.

4.3.3 Targets Used in Authorization Checks. There are many entry points that return data that may be used for authorization logic. For example, `getUserInfo` in the `UserManagerService` returns an object containing information about a specific physical user (e.g., configuration information and access rights). Depending on how the data is used, `getUserInfo` could be a part of the functionality of an entry point or part of its authorization checks. When occurring as part of the authorization checks of the system, `getUserInfo` does not expose functionality or data, and thus does not constitute a permission re-delegation vulnerability. As such, ARF excludes all paths that contain a target who is called within a context query since by definition everything occurring within a context query is part of the authorization checks of the system.

4.3.4 Deputy is Restricted to a Special Caller. Some entry points are only intended to be accessed by callers within the system or by special callers (e.g., administrative apps). Such entry points contain authorization checks that are more restrictive than any permission check present, if any. When these entry points occur as deputies, the paths will not be accessible to third-party apps and thus do not contain a permission re-delegation vulnerability.

ARF removes any path containing a deputy matching one of the following criteria. (1) The first conditional statement within a deputy is an authorization check for a special `UID`, `PID`, `GID`, `AppId`, `UserId`, and package name whose branch leads to an exit statement (i.e., a return statement or a `SecurityException`). We use Soot [28, 37] to statically analyze each deputy to detect such situations. (2) A deputy calls a context query that performs the same authorization checks as (1). To determine which context queries match the criteria in (1), we examined all the context queries for all deputies. As the number of context queries was relatively low (205), the examination only took about 2 hours. See Appendix B for a list of all the context queries of the examined deputies that matched criteria (1). (3) A deputy calls a context query that only allows special callers access to the deputy. Such context queries are unique to the service and do not contain permission checks or the authorization checks described in (1) (e.g., the context query `getActiveAdminWithPolicyForUidLocked` in the `DevicePolicyManagerService` which requires callers to be an active device administrator app). Appendix B contains the complete list of context queries that match this criteria which were determined in the same manner as (2).

4.3.5 Deputy Does Not Require Multi-User Enforcement. Recent versions of Android support multiple physical users. Multi-user accounts should be separate (i.e., one user cannot access the data of or operate on behalf of another user). However, some operations require system services and apps to access the data of multiple users or perform functionality on behalf of a non-active user. To allow this functionality, Android introduced the permissions `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL`, which grant callers limited and full access to operate as other users, respectively. Checks for these permissions are not needed on every entry point in the system as many entry points do not have the ability to function across users. In fact, for an entry point to perform operations as another user, it must be passed a `userId` indicating which user it should operate as. As such, we can detect if a entry point requires the multi-user permissions by checking if has a `userId` argument.

We use the argument variable name in the entry point's method declaration to determine if it is passed a `userId`. Unfortunately, this information is lost at compile time and is thus not available in the compiled code available to our analysis infrastructure. However, for AOSP versions of Android, the Java source code of the Android system is readily available. As such, we process the Java source code using features available in the standard `javac` compiler [32] and capture the variable names of all method arguments. We then apply text analytics in the form of the regular expression `(?i)^(?:(?:target|)user(?:id|handle|))$|^uid$` to each variable name to detect if the argument may be a `userId`. To create this regular expression, we searched for entry points that require the multi-user permissions in the output of ACMiner [23] and then manually inferred which argument represented the `userId` from available context. Using the above procedure, ARF excludes all paths whose deputies are not passed an argument that is a `userId` but whose targets are checking for the permissions `INTERACT_ACROSS_USERS` and/or `INTERACT_ACROSS_USERS_FULL` exclusively. Additionally, we perform similar exclusions when the target checks for all of the permissions `INTERACT_ACROSS_USERS`, `INTERACT_ACROSS_USERS_FULL`, `ACCESS_INSTANT_APPS`, and `VIEW_INSTANT_APPS`, since permission checks for `ACCESS_INSTANT_APPS` and `VIEW_INSTANT_APPS` were found to be a source of non-vulnerable permission re-delegation instances when combined with checks for the multi-user permissions.

4.3.6 Deputy Already Has Multi-User Enforcement. While the main multi-user enforcement in Android are checks for the permissions `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL`, some entry points require a more complex set of authorization checks. To handle this, Android created the context query and entry point `handleIncomingUser` in the `ActivityManagerService`, which includes checks for the multi-user permissions. Since `handleIncomingUser` is a separate entry point, the authorization checks inside this context query do not get included in the output of ACMiner [23] for any entry point that calls `handleIncomingUser`. As such, ARF is unable to detect that checks for `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL` actually take place when `handleIncomingUser` user is called. Therefore, ARF cannot recognize that a path is not an permission re-delegation instance when the deputy calls `handleIncomingUser` and the target checks for the multi-user permissions. To mitigate this, ARF excludes all paths where the deputy calls `handleIncomingUser` and the target checks for the permissions `INTERACT_ACROSS_USERS` and/or `INTERACT_ACROSS_USERS_FULL` exclusively. Similar to the previous case, we also perform exclusions when the target checks for all of the permissions `INTERACT_ACROSS_USERS`, `INTERACT_ACROSS_USERS_FULL`, `ACCESS_INSTANT_APPS`, and `VIEW_INSTANT_APPS`.

4.3.7 Backwards Compatibility. For compatibility reasons, the Android system has a number of code segments that are only reachable from apps compiled for certain API levels. One such code segment is located in the `getStringForUser` method of the `Settings$Secure` class and is only accessible by apps compiled for API levels below 23. The `Settings` class and its inner classes are wrappers around calls to entry points and other code used to access the global system settings. As such, all code in the `Settings` classes, including `getStringForUser`, executes in the same process as the caller until a entry point is reached. Since `getStringForUser` runs in the same process as its calling app or service and because all apps and services of the Android system must be built using the current API (i.e., 27 for Android 8.1.0), the compatibility code segment of `getStringForUser` cannot be accessed by anything in the system. While mostly unimportant, the compatibility code segment does contain a call to the entry point `getString` of the `LockSettingsService`, which performs

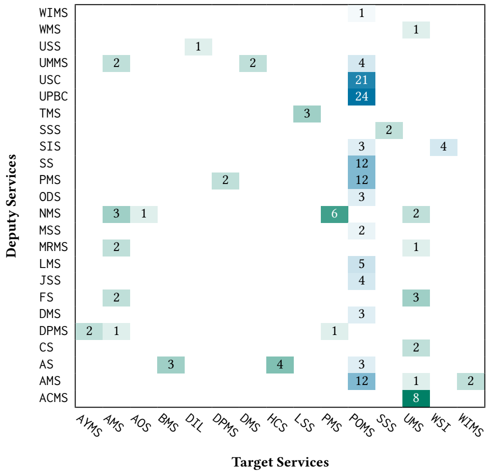

Figure 4: The deputy and target services of the paths identified with vulnerabilities.[1]

a number of permission checks. As a result, ARF generates paths for any deputy that reaches the target `getString` and is missing its permission checks, even though the compatibility code segment can never be executed. To eliminate these paths, ARF excludes all paths where the target is `getString` of the `LockSettingsService`.

5 EVALUATION

We evaluated ARF by performing an empirical analysis of the system services in AOSP version 8.1.0_r1 (i.e., API 27) built for a Nexus 5X device. Our analysis was performed on a machine with an Intel Xeon E5-2620 V3 (2.40 GHz), 128 GB RAM, running Ubuntu 14.04.1 as the host OS, and OpenJDK 1.8.0_171.

As described in Section 4, we used ACMiner [23] to mine the authorization checks of the AOSP 8.1.0 system services and iteratively applied ARF to these checks to discover all possible permission re-delegation instances. In both iterations of ARF, we manually analyzed the possible permission re-delegation paths to determine if the deputies are behaving correctly or represented vulnerabilities. Our evaluation is guided by the following research questions:

- **RQ1** *Does ARF improve a domain expert's ability to detect permission re-delegation vulnerabilities while reducing the manual effort of the domain expert?*
- **RQ2** *What is the time required by ARF to analyze all the possible permission re-delegation paths in the Android system?*
- **RQ3** *What services appear to be the most susceptible to permission re-delegation and what services are more likely to have their information or functionality improperly exposed?*
- **RQ4** *What are the major reasons for reported paths to not be permission re-delegation instances?*

We now highlight trends in our findings from our evaluation and then proceed to categorically discuss the vulnerabilities identified.

5.1 Evaluation Highlights

ARF reduced the number of paths that needed manual review during the first iteration from 15,483 down to just 2,816, a 82% reduction. The elimination techniques described in Section 4.3 further reduced this number to 419 single edge paths and 71 multiple edge paths, an overall reduction of 97%. As a result, ARF significantly enhances a domain experts ability to detect vulnerabilities caused by permission re-delegation in the Android system while minimizing the manual effort required (**RQ1**). Furthermore, ACMiner took about 1 hour and 25 minutes to output the authorization checks for the Android system of AOSP 8.1.0, with ARF adding an additional overhead of about 4 minutes to produce a list of possible permission re-delegation paths (**RQ2**). While optimizations of ARF may be possible, the time required for ARF to run is negligible compared to that of ACMiner and significantly more scalable than that of a fully manual analysis.

Through the manual analysis of the 490 paths identified by ARF, we discovered 170 paths containing vulnerabilities. Figure 4 summarizes all 170 paths by the services of their deputies and targets. The figure highlights services found to be more likely to improperly expose information or functionality, as well as the services more likely to have their data and functionality exposed (**RQ3**). For example, the `PowerManagerService` (`POMS`) has a high number of instances where its functionality is exposed by deputies. This is a result of a category of vulnerabilities we identified as allowing a caller to keep the device awake, a device state that is exclusively maintained by the `POMS`. Similarly, services `UiccPhoneBookController` (`UPBC`), `UiccSmsController` (`USC`), `ActivityManagerService` (`AMS`), `ShortcutService` (`SS`), and `PackageManagerService` (`PMS`) all have a relatively high number of paths with vulnerabilities exposing the `POMS` because they either have a number of entry points that access data from hardware, which requires the device to remain awake (i.e., `UPBC` and `USC`), or that perform operations requiring the screen to remain on (i.e., `SS`, `PMS`, and `AMS`). Apart from the `POMS`, the services with the highest number of permission re-delegation vulnerabilities are the core services `UserManagerService` (`UMS`), `AMS`, and `PMS`, which are commonly accessed by many other services to perform more complex actions. Lastly, for a variety of reasons discussed in Section 5.2, we found the `AccountManagerService` (`ACMS`), `AudioService` (`AS`), `DevicePolicyManagerService` (`DPMS`), `NotificationManagerService` (`NMS`), and `UiModeManagerService` (`UMMS`) particularly susceptible to permission re-delegation attacks. We have reported all 170 vulnerable paths to Google.

While ARF significantly reduces the number of paths a domain expert must manually verify, it does not remove all non-vulnerable paths. For AOSP 8.1.0, ARF contained 320 paths where the deputy is performing its role correctly (**RQ4**). These paths can be grouped into the following four categories. First, the deputy returns only a portion of the data (i.e., scrubbed data) provided by the target (5 paths). As such, the restrictive permission checks required by the target can be substituted for the less restrictive permission checks already in place in the deputy. Second, the deputy requires a caller to possess a special object (i.e., a token) that can only be obtained

[1] `AYMS=AccessibilityManagerService; ACMS=AccountManagerService; AMS=ActivityManagerService; AOS=AppOpsService; AS=AudioService; BMS=BluetoothManagerService; CS=ContentService; DIL=DeviceIdleController; DPMS=DevicePolicyManagerService; DMS=DreamManagerService; FS=FingerprintService; HCS=HdmiControlService; JSS=JobSchedulerService; LMS=LocationManagerService; LSS=LockSettingsService; MRMS=MediaResourceMonitorService; MSS=MediaSessionService; NMS=NotificationManagerService; ODS=OtaDexoptService; PMS=PackageManagerService; POMS=PowerManagerService; SS=ShortcutService; SIS=SipService; SSS=StorageStatsService; TMS=TrustManagerService; UPBC=UiccPhoneBookController; USC=UiccSmsController; UMMS=UiModeManagerService; USS=UsageStatsService; UMS=UserManagerService; WMS=WallpaperManagerService; WSI=WifiServiceImpl; -WIMS=WindowManagerService`

from another properly protected entry point (15 paths). Third, the paths reported by ARF are not actually traversable at run time but are instead produced as a result of noise in the call graph, a known limitation of ACMiner which ARF relies on for entry point relationship generation (63 paths). Fourth, for the remaining 237 paths, while a deputy does reach the target without checking for the required permissions of the target, the deputy is structured in such a way that the target's data and functionality is not reachable by a third-party app. All four categories described above illustrate hard problems often encountered when analyzing the Android system and are driving influences for future work.

Threats to Validity: We verified all 170 vulnerable paths through a manual inspection of the Android middleware code base. Though we have not created proof of concepts for these vulnerable paths, we are currently in talks with Google to determine the validity of the vulnerabilities and to resolve these weaknesses in platform.

5.2 Findings

Table 1 describes the vulnerabilities discovered through our analysis of Android 8.1.0 with ARF, except those involving the `WAKE_LOCK` permission which are detailed in Appendix A. By manually analyzing the the permission re-delegation instances reported by ARF, we discovered 170 vulnerable paths, which were distributed across 86 deputies. A majority of the paths with overlapping deputies actually only have a single vulnerability. Only the entry points `startManagedQuickContact`, `enableCarMode`, and `disableCarMode` have multiple vulnerabilities. For simplicity, we count each group of paths with the same deputy as a single vulnerability unless otherwise specified.

We group the permission re-delegation vulnerabilities into the following 5 categories: (1) system state modifications, (2) information leaks, (3) disrupting usability, (4) user separation and restrictions, and (5) keeping the device awake. Additionally, when examining the paths reported by ARF, we discovered several vulnerabilities that were not related to permission re-delegation, but were weaknesses in the deputies. These deputies are categorized as (6) non-permission re-delegation vulnerabilities.

VC1: System State Modifications: As shown in Table 1, ARF identified 6 deputies with permission re-delegation vulnerabilities that enable a third-party application to cause changes in the state of the system. One such deputy is `whitelistAppTemporarily` of the `UsageStatsService`. As Figure 2 shows, the deputy directly exposes the target `addPowerSaveTempWhitelistApp` of the `DeviceIdleController` to being called by third-party applications, bypassing the target's check for the system permission `CHANGE_DEVICE_IDLE_TEMP_WHITELIST`. This missing check enables any application to be added to a whitelist allowing access to the network and permitting it to keep the device awake. Such permissions are normally only granted to an application by the user or the system, but using this deputy, any application can grant themselves access. Similarly, deputies 2⟶4, expose the functionality of the target `registerStateChangeCallback` in the `BluetoothManagerService`, allowing any application to monitor the bluetooth state changes (e.g., on and device connected) without the `BLUETOOTH` permission. Lastly, deputies 5 and 6 enable any application to adjust the HDMI volume without the permission `HDMI_CEC` by exposing the targets `setSystemAudioMute`, `sendKeyEvent`, and `setSystemAudioVolume` in the `HdmiControlService`.

```
/**... Note: no permissions are required when calling these
 * APIs for your own package or UID. However, requesting
 * details for any other package requires the permission
 * PACKAGE_USAGE_STATS, which is a system-level permission
 * that will not be granted to normal apps. ...*/
@SystemService(Context.STORAGE_STATS_SERVICE)
public class StorageStatsManager {...}
```

Figure 5: The comment above the manager for the `StorageStatsService` that states all entry points of the service must have a permission check for `PACKAGE_USAGE_STATS`.

VC2: Information Leaks: Missing permission checks in 2 deputies (i.e., 7 and 8 of Table 1) cause system information from a number of targets to be leaked to callers without the necessary permissions. For example, `isDeviceSecure` of the `TrustManagerService` leaks if the device is secure (i.e., has a password, lock pattern, or other lock screen mechanism) to any caller by exposing the combined data from `havePassword`, `havePattern`, and `getLong` in the `LockSettingsService`. All three targets normally require their caller to have the system permission `ACCESS_KEYGUARD_SECURE_STORAGE` to access the data exposed by `isDeviceSecure`. Similarly, `getFreeBytes` of the `StorageStatsService` leaks a combination of the data from the targets `getCacheBytes` and `isQuotaSupported` in the `StorageStatsService` by omitting a check for the permission `PACKAGE_USAGE_STATS`. Since all three entry points are in the same service, normally the permission check within the two targets would be performed on the same calling identity as the deputy. However, the deputy clears the calling identity before invoking the targets, allowing third-party applications to bypass the permission check. Moreover, omitting the permission check in the deputy violates the policy of the service defined in a comment (Figure 5) stating that all entry points of the service must check for the `PACKAGE_USAGE_STATS` permission.

VC3: Disrupting Usability: ARF identified 8 deputies (9⟶16 of Table 1) as permission re-delegation vulnerabilities that enable a caller to disrupt device usability and possibly render a device unusable. The deputy `dismissKeyguard` of the `ActivityManagerService`, for example, enables an application to render a device inaccessible by allowing the application to dismiss the window on the Android lock screen that asks for user input to unlock the device. This is a result of the deputy exposing the functionality of targets `dismissKeyguard` and `wakeUp` in the `WindowManagerService` and `PowerManagerService`, respectively, by lacking checks for the system permissions `CONTROL_KEYGUARD` and `DEVICE_POWER`. Deputy 10 has a similar vulnerability as there is a path that leads through `dismissKeyguard` to the two targets above. On the other hand, the deputies `enqueueToast` and `cancelToast` in the `NotificationManagerService` expose the target `setProcessImportant` in the same service, enabling a third-party application without the system permission `SET_PROCESS_LIMIT` to tell the system to keep any service in the foreground. Such an action can be used by malicious third-party applications for purposes such as performing CPU intensive operations, which will both drain the battery and hinder device performance. Deputies 13⟶16 suffer from the same vulnerability as they all have a path reaching the above target through `enqueueToast`. Lastly, deputies 15 and 16 contain an additional vulnerability that exposes the target `dream` of the `DreamManagerService` to third-party applications through the omission of a check for system permission `WRITE_DREAM_STATE`. This enables applications to cause the screen to flicker as a result of the device switching in and out of VR mode.

Table 1: A description of vulnerabilities, along with the services in which they are present.[1]

Deputy (Service)	Vulnerability Description
VC1: System State Modifications	
1. `whitelistAppTemporarily` (USS)	Missing check for `CHANGE_DEVICE_IDLE_TEMP_WHITELIST` grants any app network access and allows them to keep the device awake.
2. `startBluetoothSco` (AS)	Missing check for `BLUETOOTH` allows any app to monitor changes in the bluetooth hardware state (e.g., on, off, and device connected).
3. `stopBluetoothSco` (AS)	Missing check for `BLUETOOTH` allows any app to monitor changes in the bluetooth hardware state (e.g., on, off, and device connected).
4. `startBluetoothScoVirtualCall` (AS)	Missing check for `BLUETOOTH` allows any app to monitor changes in the bluetooth hardware state (e.g., on, off, and device connected).
5. `adjustSuggestedStreamVolume` (AS)	Missing check for `HDMI_CEC` allows any app to set the HDMI volume to any value.
6. `reloadAudioSettings` (AS)	Missing check for `HDMI_CEC` allows any app to set the HDMI volume to mute or unmute depending on the system default value.
VC2: Information Leaks	
7. `isDeviceSecure` (TMS)	Missing check for `ACCESS_KEYGUARD_SECURE_STORAGE` allows any app to determine if the device has a password, lock pattern, etc.
8. `getFreeBytes` (SSS)	Missing check for `PACKAGE_USAGE_STATS` allows any app determine cache size of any other app on a volume.
VC3: Disrupting Usability	
9. `dismissKeyguard` (AMS)	Missing check for `CONTROL_KEYGUARD` allows any app to prevent the device from being unlocked by denying access to the keyguard input.
10. `enterPictureInPictureMode` (AMS)	Missing check for `CONTROL_KEYGUARD` allows any app to prevent the device from being unlocked by denying access to the keyguard input.
11. `enqueueToast` (NMS)	Missing check for `SET_PROCESS_LIMIT` allows any app to keep a service in the foreground.
12. `cancelToast` (NMS)	Missing check for `SET_PROCESS_LIMIT` allows any app to keep a service in the foreground.
13. `enqueueNotificationWithTag` (NMS)	Missing check for `SET_PROCESS_LIMIT` allows any app to keep a service in the foreground.
14. `startManagedQuickContact` (DPMS)	Missing check for `SET_PROCESS_LIMIT` allows any app to keep a service in the foreground.
15. `enableCarMode` (UMMS)	Missing check for `WRITE_DREAM_STATE` enables any app to switch the device in and out of VR mode, causing the screen to flicker.
16. `disableCarMode` (UMMS)	Missing check for `WRITE_DREAM_STATE` enables any app to switch the device in and out of VR mode, causing the screen to flicker.
VC4: Multi-User Enforcement	
17. `getPermittedAccessibilityServicesForUser` (DPMS)	Missing checks for the multi-user permissions allow any user to get a list of the accessibility services enabled for another user.
18. `isPackageDeviceAdminOnAnyUser` (PMS)	Missing checks for the multi-user permissions allow any user to determine if an app is acting as a device administrator for another user.
19. `hasNamedWallpaper` (WMS)	Missing checks for the multi-user permissions allow any user to determine if another user has a wallpaper of a specific name.
VC6: Non-Permission Re-Delegation Vulnerabilities	
20. `getEnrolledFingerprints` (FS)	Missing checks for the multi-user permissions allow one user to get references to the fingerprints of any other user.
21. `getAuthenticatorId` (FS)	Missing check for `USE_FINGERPRINT` enables any app to obtain the id of the set of fingerprints associated with another app.
22. `cancelRequest` (CS)	Missing check for `WRITE_SYNC_SETTINGS` allows a caller to terminate the periodic syncing of data to cloud services.
23. `areNotificationsEnabledForPackage` (NMS)	Missing checks for the multi-user permissions allow any user to determine if another user has enabled notifications for an app.
24. `isSeparateProfileChallengeAllowed` (DPMS)	Missing checks for the multi-user permissions enable any user to check if another user has a work profile security lock (e.g., password).
25. `getPreviousName` (ACMS)	Missing check `isAccountManageByCaller` allows any caller to get the previous name for an account (e.g., Facebook) they are not managing.
26. `isCredentialsUpdateSuggested` (ACMS)	Missing check `isAccountManageByCaller` enables any non-managing caller to see if the credentials should be updated for an online account.
27. `updateCredentials` (ACMS)	Missing check `isAccountManageByCaller` enables any caller to update the credentials of an online account they are not managing.
28. `applyRestore` (NMS)	Missing checks for the multi-user permissions allow any user to set policy for the `NMS` that globally affects all users.
29. `getBackupPayload` (NMS)	Missing checks for the multi-user permissions allow any user to get the policy for the `NMS` that globally affects all users.
30. `addStatusChangeListener` (CS)	Missing checks for the multi-user permissions allow any user to monitor the managers used by other users to sync of data to cloud services.
31. `notifyResourceGranted` (MRMS)	Missing checks for the multi-user permissions allow any user notify others that a media resource was used through a system-wide broadcast.
32. `invalidateAuthToken` (ACMS)	Missing check `isAccountManageByCaller` enables any caller to invalidate a authorization token for an online account they do not manage.
33. `switchUser` (AMS)	Missing checks for the multi-user permissions allow any user to trigger a switch to another which requires physical user input to complete.

VC4: Multi-user Enforcement: ARF identified 3 deputies (17→19 in Table 1) involving weaknesses in Android's user separation and enforcement [21]. The first, Deputy 17 leaks data from `addClient` and `getInstalledAccessibilityServiceList` in the `AccessibilityManagerService`. This vulnerability enables an application running under one user to get a list of the accessibility services enabled for any other user as a result of the missing checks for the multi-user permissions (i.e., `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL`). Similarly, because of omitted checks for the multi-user permissions, Deputy 18 leaks data from `getDeviceOwnerComponent` and `packageHasActiveAdmins` in the `DevicePolicyManagerService`, allowing any user to determine the applications that are acting as device administrators for any other user. Lastly, by omitting the same permission checks, Deputy 19 reveals if a user has a wallpaper with a given name to other users.

VC5: Keeping the Device Awake: In certain instances, it is necessary to keep the device awake to perform operations (e.g., the processing of data and accessing of data from radios and sensors). In all such instances, an entity that will cause the device to stay awake must hold the `WAKE_LOCK` permission as the five entry points that control the device awake state require (i.e., `acquireWakeLock`, `releaseWakeLock`, and `updateWakeLockWorkSource` in the `PowerManagerService` or `acquireWifiLock` and `releaseWifiLock` in the `WifiServiceImpl`). However, when inspecting the output of ARF, we found 55 instances where the deputy fails to check for the `WAKE_LOCK` permission, despite having a path that leads to one of the five aforementioned entry points. As a result, there are many deputies that, through repetitive calling, allow a caller to keep the device awake without holding the `WAKE_LOCK` permission. As the list of deputies that can be used to keep the device awake is large, we omit them from Table 1 and instead include them in Appendix A.

```
/**... The part of Android Keystore which runs inside an
 * app's process invokes this method in certain cases but the
 * developer does not always demonstrate intent to use
 * fingerprint functionality. Thus, to avoid throwing an
 * unexpected SecurityException we do not check access. The
 * permission check should be restored once Android Keystore
 * no longer invokes this method from inside app processes.*/
public long getAuthenticatorId(String opPackageName) {...}
```

Figure 6: The comment for the `getAuthenticatorId` of the `FingerprintService` illustrating where developers compromised security to quickly fix a bug.

VC6: Non-Permission Re-Delegation Vulnerabilities: The goal of ARF is to identify permission re-delegation vulnerabilities. However, more generalized forms of missing checks (which were the focus of ACMiner [23]) still exist in the system. While investigating the output of ARF, we discovered 14 deputies (20→33 in Table 1) that have vulnerabilities resulting from missing authorization checks, but do not expose the information or functionality of the targets to third-party applications. For example, `getEnrolledFingerprints` of the `FingerprintService` allows an application with the permission `USE_FINGERPRINT` running under one user to obtain references to the fingerprints of any other user. This is yet another vulnerability in the multi-user enforcement of Android resulting from the missing checks for the multi-user permissions (i.e., INTERACT_ACROSS_USERS and INTERACT_ACROSS_USERS_FULL). In another example, the entry point getAuthenticatorId of the FingerprintService enables applications without the needed normal USE_FINGERPRINT permission to obtain the id of the set of fingerprints associated another app. These fingerprints would be those that are application specific (e.g., Google Pay) and not used to unlock the device. Regardless, they should be protected by a check for the USE_FINGERPRINT permission. Indeed, as shown in Figure 6, comments in the code state that the permission should be checked but further clarify that said check was removed to prevent "unexpected SecurityExceptions" from occurring because of design flaws.

6 LIMITATIONS

As ARF relies on the output of ACMiner [23] for its analysis, it retains many of its limitations. These limitations include the requirement of manually defined input, the inability to reason about authorization check ordering in a majority of cases, and the general limitations shared by tools that statically analyze Android (e.g., native code, runtime modifications, reflection, dynamic code loading, and Message Handlers). In addition to these limitations, ARF is unable to detect re-delegation vulnerabilities where the target contains only authorization checks that are not permission checks (e.g., the special callers from Section 4.3.4). This limitation is a result of ARF's narrowed focus on targets with permission checks. Furthermore, ARF omits paths where the deputy contains both system permission checks and third-party permission checks as it cannot distinguish which occur first, a limitation stemming from ACMiner. We plan to address both limitations future work.

7 RELATED WORK

Several prior works have addressed permission re-delegation issues in Android. However, all have focused on scenarios where the deputy is an application. For example, Davi et al. [8] exploited vulnerabilities in the native code environments of applications to gain access to another applications ability to send SMS messages without holding the appropriate permissions. Furthermore, Felt et al. [18] and Quire [9] both developed systems for the prevention of third-party application to application permission re-delegation attacks through the tracking of binder IPC calls and privilege reduction. Lastly, CHEX [30], Woodpecker [24], and SEFA [38] developed techniques for the detection of permission re-delegation vulnerabilities in applications. CHEX analyzed third-party applications while Woodpecker and SEFA treated system applications (e.g., settings) as the deputy, with Woodpecker focusing on vulnerabilities found in stock versions of system applications and SEFA looking a vulnerabilities introduced as a result of vendor modifications.

Prior works focusing on Android have also considered the consistency of the authorization logic of the Android system. Kratos [36] uses call graph comparison and a small, manually defined set of authorization checks to identify paths in the call graph where sensitive operations could be reached with two conflicting sets of authorization checks. AceDroid [1] expanded on Kratos by improving on the manually defined set of authorization checks and developing techniques to simplify the authorization logic through the combination of authorization checks that physically appear different but conceptually behave the same. Unfortunately, AceDroid greatly over-simplifies the authorization logic in order to perform its analysis. As such, both Kratos and AceDroid did not meet the needs of ARF, which relies on knowledge of authorization checks other than permission checks to eliminate non-vulnerable paths. Fortunately such information is provided by ACMiner [23], which semi-automatically identifies authorization checks. ACMiner also uses association rule mining to perform its consistency analysis.

The complexity of the Android middleware motivated research that aids application analysis by creating simplified models of the authorization logic of API calls. For example, Stowaway [15] uses fuzzing to dynamically extract the permissions being checked when calling an API. As a result of the limitations of dynamic analysis, PScout [5] improves on Stowaway's permission check model using relatively simple static analysis. Axplorer [6] further refines this model by addressing complexities in Android that have made static analysis difficult. Most recently, Arcade [2] improves the precision of the model further by including other types of authorization checks as well logical relationships between checks.

In parallel to the above research, prior works have proposed many static and dynamic analysis techniques to analyze third-party applications. These works have targeted topics such as malware [14, 27, 40], privacy leaks and infringements [4, 10, 19, 20, 22, 26], and vulnerabilities [7, 11]. Furthermore, several prior works have tackled challenges related to inter-component communication (ICC) in Android (e.g., EPICC [31] and IccTA [29]). Moreover, through the review of the platform code, Xing et al. [39] discovered vulnerabilities that allowed privilege escalation on update, and Zhou et al. [42] revealed holes in the file access control policies of firmware images. While sharing goals with prior works, ARF uniquely provides a semi-automated approach for discovering permission re-delegation vulnerabilities within the Android system.

8 CONCLUSION

This paper builds upon recent efforts to more rigorously analyze authorization logic in Android's middleware. Specifically, we revisited the problem of permission re-delegation. In contrast to prior work that has studied permission re-delegation by third-party and system applications, we studied permission re-delegation within system services. We proposed ARF as an analysis framework to aid security analysts in efficiently identifying deputies in system services that improperly expose information or functionality to third-party applications. We applied ARF to Android AOSP version 8.1.0 and show that it can reduce the manual analysis effort from 15,483 paths between entry points down to just 490. We manually reviewed this set, identifying 170 paths that improperly exposed information or functionality, spanning 86 deputies. This effort demonstrates the continued need for researchers to study authorization logic within the Android middleware and to build novel tools to identify flaws that regularly evade manual code review.

Acknowledgements: This work was supported by the Army Research Office (ARO) grant W911NF-16-1-0299. Opinions, findings, conclusions, or recommendations in this work are those of the authors and do not reflect the views of the funders.

REFERENCES

[1] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS).*

[2] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API Protection Mapping Derivation and Reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[3] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. SoK: Lessons Learned from android Security Research for Appified Software Platforms. In *Proceedings of the IEEE Symposium on Security and Privacy.*

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

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security.* 217–228.

[6] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the USENIX Security Symposium.*

[7] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services.*

[8] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC).*

[9] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium.*

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

[11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proceedings of the USENIX Security Symposium.*

[12] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS).*

[13] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding Android Security. *IEEE Security & Privacy Magazine* 7, 1 (January/February 2009).

[14] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[15] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[16] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdata Akhawe, and David Wagner. 2012. How to Ask for Permission. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec).*

[17] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS).*

[18] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium.*

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

[20] Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST).*

[21] Google. 2018. Supporting Multiple Users. https://source.android.com/devices/tech/admin/multi-user. Accessed Sep. 11, 2018.

[22] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS).*

[23] Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY).*

[24] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the ISCO Network and Distributed System Security Symposium (NDSS).*

[25] N. Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.

[26] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[27] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the International Conference on Software Engineering (ICSE).*

[28] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java Program Analysis: A Retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS).*

[29] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2014. I Know What Leaked in Your Pocket: Uncovering Privacy Leaks on Android Apps with Static Taint Analysis. In *CoRR.*

[30] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[31] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication Mapping in Android with EPPIC: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium.*

[32] Oracle. 2019. Obtaining Names of Method Parameters. https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html. Accessed Jan. 15, 2019.

[33] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the USENIX Security Symposium.*

[34] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[35] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. *Droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Computing Surveys (CSUR)* 49, 3 (Oct. 2016).

[36] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS).*

[37] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - A Java Bytecode Optimization Framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research.*

[38] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The Impact of Vendor Customizations on Android Security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* 623–634.

[39] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. 2014. Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[40] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[41] Yury Zhauniarovich and Olga Gadyatskaya. 2016. Small Changes, Big changes: An Updated View on the Android Permission System. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID).*

[42] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proc. of the IEEE Symposium on Security and Privacy.*

A DEPUTIES TO KEEP THE DEVICE AWAKE

Table 2 provides a list of all deputies that were found to have paths to the targets that control the device awake state (i.e., acquireWakeLock, releaseWakeLock, and updateWakeLockWorkSource in the PowerManagerService or acquireWifiLock and releaseWifiLock in the WifiServiceImpl).

Table 2: The deputies that can be used to keep the device awake.

Service	Method Names
ActivityManagerService	activityPaused activitySlept addAppTask setTaskResizeable setVoiceKeepAwake startLockTaskModeById
AudioService	handleBluetoothA2dpDeviceConfigChange setBluetoothA2dpDeviceConnectionState setWiredDeviceConnectionState
DreamManagerService$BinderService	finishSelf startDozing stopDozing
JobSchedulerService$JobSchedulerStub	cancel cancelAll
LocationManagerService	locationCallbackFinished removeUpdates sendExtraCommand
MediaSessionService$SessionManagerImpl	dispatchMediaKeyEvent
OtaDexoptService	prepare
PackageManagerService	performDexOptMode performDexOptSecondary registerDexModule runBackgroundDexoptJob
ShortcutService	addDynamicShortcuts disableShortcuts enableShortcuts getDynamicShortcuts getManifestShortcuts getPinnedShortcuts getRemainingCallCount removeAllDynamicShortcuts removeDynamicShortcuts reportShortcutUsed setDynamicShortcuts updateShortcuts
SipService	close open3
UiccPhoneBookController	getAdnRecordsInEf getAdnRecordsInEfForSubscriber getAdnRecordsSize getAdnRecordsSizeForSubscriber updateAdnRecordsInEfByIndex updateAdnRecordsInEfByIndexForSubscriber updateAdnRecordsInEfBySearch updateAdnRecordsInEfBySearchForSubscriber
UiccSmsController	copyMessageToIccEfForSubscriber disableCellBroadcastForSubscriber disableCellBroadcastRangeForSubscriber enableCellBroadcastForSubscriber enableCellBroadcastRangeForSubscriber getAllMessagesFromIccEfForSubscriber updateMessageOnIccEfForSubscriber
UiModeManagerService$6	disableCarMode enableCarMode
WindowManagerService	updateRotation

B CONTEXT QUERIES RESTRICTING ACCESS TO SPECIAL CALLERS

Table 3 provides a complete list of all context queries we discovered that restrict a caller to either the system or other special callers (e.g., administrative apps).

Table 3: The context queries restricting access to deputies.

Service	Method Names
System Restricting Context Queries	
AppOpsService	checkSystemUid checkPackage
DevicePolicyManagerService	enforceShell isCallerWithSystemUid
NotificationManagerService	checkCallerIsSystem checkCallerIsSystemOrSameApp checkCallerIsSystemOrShell
PackageManagerService	enforceSystemOrRoot enforceSystemOrPhoneCaller
UserManagerService	checkSystemOrRoot
ShortcutService	verifyCaller enforceSystem enforceSystemOrShell
LockSettingsService	ensureCallerSystemUid
SmsUsageMonitor	checkCallerIsSystemOrPhoneOrSameApp
Special Caller Restricting Context Queries	
AccountManagerService	permissionIsGranted isAccountManagedByCaller canUserModifyAccounts canUserModifyAccountsForType
DevicePolicyManagerService	getActiveAdminWithPolicyForUidLocked
LocationManagerService	canCallerAccessMockLocation
NetworkScoreService	isCallerActiveScorer
NotificationManagerService$7	verifyPrivilegedListener