ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware

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

Benjamin Andow
North Carolina State
University
beandow@ncsu.edu

Adwait Nadkarni
William & Mary
nadkarni@cs.wm.edu

Sunil Manandhar
William & Mary
sunil@cs.wm.edu

William Enck
North Carolina State
University
whenck@ncsu.edu

Eric Bodden
Paderborn University
eric.bodden@uni-paderbo
rn.de

Alexandre Bartel
University of Luxembourg
alexandre.bartel@uni.lu

ABSTRACT

Billions of users rely on the security of the Android platform to protect phones, tablets, and many different types of consumer electronics. While Android's permission model is well studied, the *enforcement* of the protection policy has received relatively little attention. Much of this enforcement is spread across system services, taking the form of hard-coded checks within their implementations. In this paper, we propose Authorization Check Miner (ACMiner), a framework for evaluating the correctness of Android's access control enforcement through consistency analysis of authorization checks. ACMiner combines program and text analysis techniques to generate a rich set of authorization checks, mines the corresponding protection policy for each service entry point, and uses association rule mining at a service granularity to identify inconsistencies that may correspond to vulnerabilities. We used ACMiner to study the AOSP version of Android 7.1.1 to identify 28 vulnerabilities relating to missing authorization checks. In doing so, we demonstrate ACMiner's ability to help domain experts process thousands of authorization checks scattered across millions of lines of code.

1 INTRODUCTION

Android has become the world's dominant computing platform, powering over 2 billion devices by mid-2017 [10]. Not only is Android the primary computing platform for many end-users, it also has significant use by business enterprises [33] and government agencies [34, 37]. As a result, any security flaw in the Android platform is likely to cause significant and widespread damage, lending immense importance to evaluating the platform's security.

While Android is built on Linux, it has many differences. A key appeal of the platform is its semantically rich application programming interfaces (APIs) that provide application developers simple and convenient abstractions to access information and resources (e.g., retrieve the GPS location, record audio using the microphone, and take a picture with the camera). This functionality, along with corresponding security checks, is implemented within a collection of privileged userspace services. While most Android security research has focused on third party applications [7, 13, 15, 18, 19, 35, 36, 39], the several efforts that consider platform security highlight the need for more systematic evaluation of security and access control checks within privileged userspace services (e.g., evidence of system apps re-exposing information without security checks [25, 43], or missing checks in the Package Manager service leading to Pile-Up vulnerabilities [44]).

To date, only two prior works have attempted to evaluate the correctness of access control logic within Android's system services. Both Kratos [38] and AceDroid [2] approximate correctness through consistency measures, as previously done for evaluating correctness of security hooks in the Linux kernel [12, 29, 41]. However, these prior works have limitations. Kratos only considers a small number of manually-defined authorization checks (e.g., it excludes App Ops checks). AceDroid considers a larger set of authorization checks, but these are still largely manually defined, primarily through observation. Kratos performs coarse-grained analysis using call-graphs, leading to imprecision. AceDroid's program analysis provides better precision, but oversimplifies its access control representation, making it difficult to identify vulnerabilities within single system images.

In this paper, we propose Authorization Check Miner (ACMiner), a framework for evaluating the correctness of Android's access control enforcement through consistency analysis of authorization checks. ACMiner is based on several novel insights. First, we avoid identification of protected operations (a key challenge in the space) by considering program logic between service entry points and code that throws a `SecurityException`. Second, we propose a semi-automated method of discovering authorization checks. More specifically, we mine all constants and names of methods and vari-

ables that influence conditional logic leading to throwing a `Security-Exception`. From this dataset, we identify security-relevant values (e.g., "restricted") and develop regular expressions to automatically identify those conditions during program analysis that mines policy rules from the code. Third, we use association rule mining to identify inconsistent authorization checks for entry points in the same service. Association rule mining has the added benefit of suggesting changes to make authorization checks more consistent, which is valuable when triaging results. By applying this methodology, ACMiner allows a domain expert (i.e., a developer familiar with the AOSP source code) to quickly identify missing authorization checks that allow abuse by third-party applications.

We evaluated the utility of ACMiner by applying it to the AOSP code for Android 7.1.1. Of the 4,004 total entry points to system services, ACMiner identified 1,995 with authorization checks. Of these entry points, the association rule mining identified inconsistencies in 246. We manually investigated these 246 entry points with the aid of the rules suggested by the association rule mining, which allowed us to identify 28 security vulnerabilities. ACMiner not only reduced the effort required to analyze system services (i.e., by narrowing down to only 246 entry points out of 4004), but also allowed us to rapidly triage results by suggesting solutions. Out of the 28 security vulnerabilities, 7 were in security-sensitive entry points that may be exploited from third-party applications, and an additional 12 were in security-sensitive entry points that may be exploited from system applications. The rest were in entry points with relatively low security value. All 28 vulnerabilities have been reported to Google. At the time of writing, Google has confirmed 2 of these vulnerabilities as "moderate severity."

This paper makes the following contributions:

- *We design and implement ACMiner, a framework that enables a domain expert to identify and systematically evaluate inconsistent access control enforcement in Android's system services.* Our results show that this analysis is not only useful for identifying existing vulnerabilities, but also inconsistencies that may lead to vulnerabilities in the future.
- *We combine program and text analysis techniques to generate a rich set of authorization checks used in system services.* This technique is a significant improvement over prior approaches that use manually-defined authorization checks.
- *We use ACMiner to evaluate the AOSP version of Android 7.1.1 and identify 28 vulnerabilities.* All vulnerabilities have been reported to Google, which at the time of writing has classified 2 as "moderate severity."

This paper describes how ACMiner can systematically analyze the consistency of the authorization checks in the system services. However, ACMiner may also be useful for other forms of analysis. For instance, ACMiner can aid regression testing, as it can be extended to highlight changes to the policy implementation on a semantic level. The information extracted by ACMiner can also be used to study the evolution of access control in Android, potentially discovering new vulnerabilities. Finally, since changes by OEMs have historically introduced vulnerabilities, OEMs can use ACMiner to validate their implemented checks against AOSP.

The remainder of this paper proceeds as follows. Section 2 provides background. Section 3 describes the challenges and provides

```
1   // Entry point with correct authorization checks
2   boolean hasBaseUserRestriction(String key, int userId) {
3       checkManageUsersPermission("hasBaseUserRestriction");
4       // Unique check without a SecurityException
5       if (!UserRestrictionsUtils.isValidRestriction(key))
6           return false;
7       ...}
8
9   // Entry point missing checkManageUsersPermission
10  boolean hasUserRestriction(String key, int userId) {
11      if (!UserRestrictionsUtils.isValidRestriction(key))
12          return false;
13      ...}
14
15  void checkManageUsersPermission(String message) {
16      if (!hasManageUsersPermission())
17          throw new SecurityException();}
18
19  boolean hasManageUsersPermission() {
20      int callingUid = Binder.getCallingUid();
21      return UserHandle.isSameApp(callingUid,
22          Process.SYSTEM_UID)
23      || callingUid == Process.ROOT_UID
24      || ActivityManager.checkComponentPermission(
25          "android.permission.MANAGE_USERS",
26          callingUid, -1, true) ==
27          PackageManager.PERMISSION_GRANTED;}
```

Figure 1: Vulnerability found in `UserManagerService` by our tool

an overview of ACMiner. Section 4 describes the design of ACMiner in detail. Sections 5 and 6 describe our analysis of the system services of AOSP 7.1.1. Section 7 describes the limitations of our approach. Section 8 discusses related work. Section 9 concludes.

2 BACKGROUND AND MOTIVATION

The Android middleware is implemented using the same component abstractions as third-party applications [16]: activities, content providers, broadcast receivers, and services. In this paper, we only consider service components, which provide daemon-like functionality. Apps interface with service components via the `Binder` inter-process communication (IPC) mechanism, which consists of sending *parcel* objects that indicate the target interface method being called via an integer. For the most part, Android's system services use the Android Interface Description Language (AIDL) to automatically generate the code that unmarshalles these parcels. Moreover, when interfacing with system services, third party apps rely on public APIs implementing a *proxy* to construct the parcel. When the parcel is unmarshalled by the service interface, the arguments are passed to a *stub* that calls the corresponding *entry point* method in the service component.

Android uses two broad techniques to enforce access control. For coarse-grained checks (i.e., at the component level), the Activity Manager Service (AMS) enforces policy specified in application manifest files. This paper focuses on fine-grained checks (i.e., at the service entry point level), which are enforced using hard-coded logic within the service implementation. This hard-coded logic includes variants of the `checkPermission` method, Unix Identifier (UID) checks, as well as many subtle checks based on service-specific state. Prior work [2, 38] has primarily relied on manual enumeration of these checks, which is error prone. To simplify discussion in this paper, we refer to such methods that return or check Android system state as *context queries*.

Figure 1 provides a motivating example for this paper, which contains a vulnerability discovered by ACMiner. The simplified

code snippet is from the User Manager Service, which provides core functionality similar to the Activity Manager and Package Manager Services. In the figure, there are two entry points: hasBaseUserRestriction and hasUserRestriction. The entry points perform very similar functionality, but have inconsistent authorization checks. Specifically, hasBaseUserRestriction throws a SecurityException if the caller does not have the proper UID or permission.

This example is particularly apropos to ACMiner, because hasUserRestriction does not call any of the context queries considered by prior work [2, 38]. It also does not throw a SecurityException. Without knowledge that isValidRestriction is an authorization check, no form of consistency analysis could have identified that hasUserRestriction has a missing check.

3 OVERVIEW

The goal of this paper is to help a domain expert quickly identify and assess the impact of incorrect access control logic in implementations of system services in Android. As with most nontrivial software systems, no ground truth specification of correctness exists. Rather, the "ground truth" resides largely within the heads of the platform developers. Prior literature has approached this type of problem by approximating correctness with consistency. The intuition is that system developers are not malicious and that they are likely to get most of the checks correct. The approach was first applied to security hook placement in the Linux kernel [12, 29, 41] and more recently the Android platform [2, 38].

Evaluating authorization check correctness via consistency analysis requires addressing the following challenges.

- *Protected Operations:* Nontrivial systems rarely have a clear specification of the functional operations that require protection by the access control system. Protected operations range from accessing a device node to reading a value from a private member variable. Axplorer [6] attempts to enumerate protected operations for Android; however, the specification is far from complete.
- *Authorization Checks:* What constitutes an authorization check is vague and imprecise. While some authorization checks are clear (e.g., those based on checkPermission and getCallingUid), many others are based on service-specific state and the corresponding authorization checks use a variety of method and variable names.
- *Consistency:* The granularity and type of consistency impacts the precision and utility of the analysis. While increasing the scope of relevant authorization checks increases the noise in the analysis, not considering all authorization checks (as in Kratos [38]) or using heuristics to determine relevancy (as in AceDroid [2]) raises the risk of not detecting vulnerabilities.

ACMiner addresses these challenges through several novel insights. First, ACMiner avoids the need to specify protected operations by considering program logic between service entry points and code that throws a SecurityException. Our intuition is that if one control flow path leads to a SecurityException, an alternate control flow path leads to a protected operation. Furthermore, the conditional logic leading to the SecurityException describes the authorization checks. However, we found that not all authorization denials lead to a SecurityException, therefore, we also include
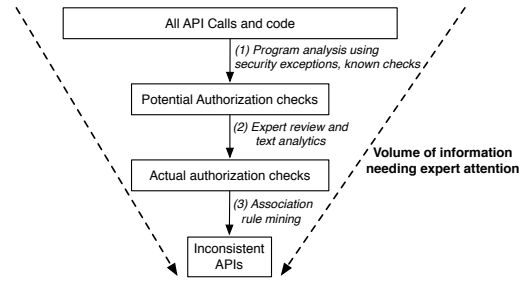

Figure 2: Overview of ACMiner. At each stage, ACMiner significantly reduces the information an expert needs to analyze.

entry points that contain known authorization checks. Second, ACMiner semi-automatically discovers new authorization checks using a combination of static program analysis and textual processing. More specifically, ACMiner identifies all of the method names, variable names, and strings that influence the conditional logic leading to a SecurityException. The security-relevant values are manually refined and used to generate regular expressions that identify a broader set of authorization checks within service implementations. Third and finally, ACMiner uses association rule mining for consistency analysis. For each entry point, ACMiner uses static program analysis to extract a set of authorization checks. Association rule mining compares the authorization check sets between entry points in the same service. The analysis produces suggestions (called "rules") of how the sets should change to make them more consistent. These rules include confidence scores that greatly aid domain experts when triaging the results. This general approach is depicted in Figure 2.

To more concretely understand how ACMiner operates, consider the discovery of the vulnerability shown in Figure 1. As a pre-processing step, ACMiner helps a domain expert semi-automatically identify authorization checks. First, ACMiner determines that the return value of isValidRestriction controls flow from the entry point hasBaseUserRestriction to a SecurityException. As such, this method name, along with many security irrelevant names are given to a domain expert. The domain expert then identifies security relevant terms (e.g. "restriction"), which ACMiner consumes as part of a regular expression. Next, ACMiner mines the policy of the User Manager Service, extracting a policy for both hasBaseUserRestriction and hasUserRestriction, with the policy for hasUserRestriction only being extracted because isValidRestriction was identified as an authorization check. For each entry point, the policy is then encoded as a set of authorization checks (e.g. isValidRestriction and ROOT_UID == getCallingUid()). Finally, ACMiner performs association rule mining to suggest set changes that make the policy more consistent. Such suggestions led us to discover the vulnerability in hasUserRestriction.

4 DESIGN

ACMiner is constructed on top of the Java static analysis framework Soot [30, 42] and has been largely parallelized so as to improve the run time of the complex analysis of Android's services. The design of ACMiner can be divided into three phases: (§4.1) Mining Authorization Checks, (§4.2) Refining Authorization Checks, and (§4.3) Inconsistency Analysis.
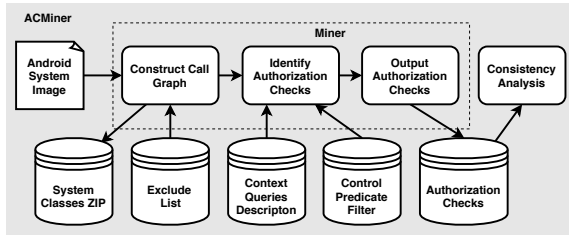
Figure 3: ACMiner's processing stages and input files.

4.1 Mining Authorization Checks

The first phase of ACMiner is mining authorization checks. This phase consist of the following program analysis challenges: (§4.1.1) Call Graph Construction, (§4.1.2) Identifying Authorization-Check Statements, and (§4.1.3) Representing Authorization Checks.

4.1.1 Call Graph Construction. Authorization checks are mined by traversing a call graph of the service implementation. ACMiner constructs call graphs using the following three steps.

Extracting Java Class Files: ACMiner extracts a *.jar* containing all the class files of the Android middleware in Soot's Jimple format from Android's *system.img*. This *.jar* containing Jimple files is then used on all subsequent runs of ACMiner. The implementation of this approach is detailed in the extended version [24].

Extracting System Services and Entry Points: Android's middleware is composed of isolated services that communicate through predefined `Binder` boundaries. This division allows ACMiner to analyze each service separately, by defining each service by the code reachable through its `Binder` entry points. ACMiner extracts system services and their entry points similar to prior work [2, 5, 38]. For implementation details, please see the extended version [24].

Reducing the Call Graph: ACMiner constructs a call graph representing all possible transitive calls from the entry points. ACMiner uses the Class Hierarchy Analysis (CHA) [11], which is guaranteed to provide an over-approximation of the actual runtime call graph. In contrast, Kratos and AceDroid use other potentially more accurate call graph builders (i.e., SPARK [32] and its WALA equivalent), which use points-to analysis to construct a less complete under-approximation of the runtime call graph. The loss of completeness occurs when constructing call graphs for libraries and other Java code without main methods. Therefore, it is important to note that unlike the prior work, ACMiner is more complete and guaranteed to include all paths containing authorization checks.

Since CHA call graphs are coarse over-approximations of the runtime call graph, ACMiner must apply heuristics to mitigate call graph bloat. When resolving targets for method invocations, CHA considers every possible implementation of the target method whose declaring type is within the type hierarchy of the call's receiver type. If the invoked method is defined in a widely used interface or superclass, the resolution may identify hundreds of targets for a single invocation. Thus, the resulting CHA call graph for the Android middleware is too large to be analyzed in a reasonable amount of time and memory [31].

To mitigate call graph bloat, we manually defined a list of classes and methods to exclude from the analysis, which become cutoff points in the call graph. We ensured that the class or method subject to exclusion did not contain, lead to, or was used in an authoriza-

tion check. While the exclude list may require revision for newer versions of AOSP or modifications made by vendors, the creation of the exclude list is a largely one-time effort. Please see the extended version [24] for a detailed description of the exclusion procedure and our website [1] for a full list of excluded classes/methods.

Finally, when analyzing an entry point, ACMiner treats all other entry points as cutoff points in the call graph. This decision further reduces call graph bloat. Unfortunately, it also introduces unsoundness into the call graph, which we discuss in Section 7.

4.1.2 Identifying Authorization Check Statements. Once ACMiner has the call graph for all entry points, the next step is to identify authorization checks. As described in Section 2, the complete set of authorization checks is unknown. ACMiner takes a two pronged approach to identifying authorization checks. First, it identifies all possible checks leading to a protected operation (this section). Second, it refines the list of possible authorization checks based on code names and string values (Section 4.2). To describe this process, we must first define a *control predicate*.

Definition 1 (*Control Predicate*). A conditional statement (i.e., an `if` or `switch` statement) whose logic authorizes access to some protected operation.

Identifying protected operations is nontrivial, as they may range from accessing a device node to returning a private member variable. However, even if we knew the protected operations, we would still need to determine which conditional statements are control predicates. ACMiner uses the key observation that Android frequently throws a `SecurityException` when access is denied. Therefore, ACMiner marks all conditional statements on the control flow path between entry points and the statement throwing the `SecurityException` as *potential* control predicates.

While throwing a `SecurityException` is the most common way Android denies access, it is not the only way. Some entry points deny access by returning false or even by returning empty data. Such denials are not easily identifiable. Fortunately, as shown in Figure 1, Android often groups authorization checks into methods to simplify placement. We refer to such groups of authorization checks as *context queries*.

Definition 2 (*Context Query*). A method consisting entirely of a set of *control predicates*, calls to other *context queries*, and/or whose return value is frequently used as part of a *control predicate*.

By identifying *context queries*, ACMiner can mark control predicates not on the path between a entry point and a thrown `SecurityException`, thereby making the authorization check mining more complete. As shown in Figure 3, ACMiner is configured with a input file that specifies context queries. Our method for defining this input is described in Section 4.2.

Using these insights, ACMiner identifies authorization checks with fairly high accuracy. The identification procedure is as follows. First, ACMiner marks all conditional statements inside a context query and the subsequent transitive callees as control predicates for the entry point. Next, ACMiner performs a backwards inter-procedural control flow analysis from each statement throwing a `SecurityException` and each context query invocation to the entry point. During this backwards analysis, ACMiner marks all condi-

tional statements on the path as potential control predicates. Finally, to capture control predicates that occur without a `SecurityException`, ACMiner performs a forward inter-procedural def-use analysis on the return value of a context query. During this analysis, AC-Miner marks any conditional statement that uses the return value as a potential control predicate. Note, ACMiner does not currently track the return value through fields, as this was found to be too noisy. However, ACMiner can track the return value through variable assignments, arithmetic operations, array assignments, and the passed parameters of a method invoke.

4.1.3 Representing Authorization Checks. ACMiner represents the authorization checks for each entry point as a set of context queries and control predicates. We initially represented authorization checks as boolean expressions representing the control flow decisions that lead to a thrown `SecurityException` or invoked context query. This representation would allow ACMiner to verify the existence, order, and the comparison operators of the authorization checks. However, for complex services (e.g., the Activity Manager) this representation was infeasible due to the large number of authorization checks. Additionally, we found that without more complex context-sensitive analysis, ACMiner could not extract authorization checks involving implicit flows. Therefore, we simplified our consistency analysis to only consider *the existence* of an authorization check for an entry point. This approach requires two reasonable correctness assumptions: (1) all authorization checks have been placed and ordered correctly, and (2) all control predicates have the correct comparison operator. ACMiner cannot detect violations of these assumptions.

The existence of authorization checks is easily represented as a set; however some processing is required. For each variable in a authorization check statement, ACMiner must substitute all possible values for that variable. More specifically, for each control predicates and context queries statement (i.e., conditional or method invoke statement), ACMiner must generate a list containing the product of all the possible variables and the values for each variable. To reduce redundant output, ACMiner only computes the product for context queries that do not have a return value or whose return value is not used in a control predicates.

For this expansion, ACMiner applies an inter-procedural def-use analysis to each variable used in a statement, thereby obtaining the set of all possible values for that variable from the entry point to this specific use site. It then computes the product of these sets to achieve the complete set of authorization checks for a single statement. If a variable is assigned a value from the return of a method call, ACMiner does not consider the possible return values of the method, but instead includes a reference to all the possible targets of the method call. Similarly, if a variable is assigned a value from the field of a class or an array, ACMiner includes only a reference to the field or array instead of all the possible values that may be assigned to the field or array. As such, while the list of values largely consists of constants, it may also contain references to fields, methods and arrays. The resulting combination of all the iterations of values for each control predicate and each required context query of an entry point makes up the final set of authorization checks output by ACMiner.

The resulting set has the potential to be exponentially large. To prevent this growth and to remove noise in ACMiner's output,

Table 1: Initial List of *Context Queries*

Classes	Methods
Context	enforcePermission
ContextImpl	enforceCallingPermission
ContextWrapper	enforceCallingOrSelfPermission
	checkPermission
	checkCallingPermission
	checkCallingOrSelfPermission
AccountManagerService	checkBinderPermission
LocationManagerService	checkPackageName
IActivityManager	checkPermission
ActivityManagerProxy	
ActivityManagerService	
ActivityManagerService	checkComponentPermission
ActivityManager	checkUidPermission
IPackageManager	checkUidPermission
IPackageManager$Stub$Proxy	checkPermission
PackageManagerService	

we apply several simplifications to the authorization checks (see the extended version [24]). These simplifications are designed to increase the number of authorization checks that are equivalent from a security standpoint and in no way affect the completeness of the authorization checks.

4.2 Refining Authorization Checks

The techniques in Section 4.1.2 identify *potential* control predicates; however, not all conditional statements are authorization checks. Performing consistency analysis at this point would be infeasible due to the excessive noise in the data. Therefore, ACMiner uses a one-time, semi-automated method to significantly reduce noise.

Our key observation is that Section 4.1 over approximates authorization checks on the path from entry points to a thrown `SecurityException` or context query. From this over-approximation, ACMiner can generate a list of all the strings and fields used in the conditional statements, a list of the methods whose return values are used in the conditional statements, and the methods containing the conditional statements. These values can be manually classified as authorization-related or not. The general refinement procedure is as follows: (1) a domain expert filters out values not related to authorization; (2) the refined list is translated into generalized expressions; (3) ACMiner uses the generalized expressions to automatically filter out values not related to authorization; (4) the generalized expressions are refined until the automatically generated list is close to that defined by the domain expert. While creating generalized expressions is time consuming, they can be used to analyze multiple Android builds with minimal modifications.

The specific refinement procedure is divided into two phases: (§4.2.1) identifying additional context queries, and (§4.2.2) refining control predicate identification.

4.2.1 Identifying Additional Context Queries. ACMiner uses context queries as indicators of the existence of authorization checks when no `SecurityException` is thrown. Our initial list of *context queries*, shown in Table 1, was very limited and only contained 33 methods. To expand this list we performed the following steps: (1) run ACMiner as described in Section 4.1 using the initial list of *context queries*, (2) from the marked conditional statements, generate a list of the methods containing these conditional statements and the methods whose return values are used in these conditional statements, (3) have a domain expert inspect the list and identify methods that match our definition of a context query, and add these to our list of context queries, and (4) repeat steps 1→3 until no new context queries are added to the list. For Android AOSP 7.1.1, this

```
(and (or (starts-with-package android.)
         (starts-with-package com.android.))
     (regex-name-words `^(enforce|has|check)\s
         ([a-z\s]+\s)?permission(s)?\b`)
     (not (equal-package android.test))...))
```

Figure 4: Example expressions to describe context queries.

procedure took about 48 hours and increased the number of *context queries* to 620 methods.

To make this list reusable, we translate it into a set of generalized expressions that describe context queries across different Android versions. The expressions consist of regular expressions and string matches for the package, class, and name of a method, and also include conditional logic. An example expression is shown in Figure 4. Overall, we defined 49 generalized expressions to describe context queries for Android AOSP 7.1.1, which took <10 hours. The expressions enabled ACMiner to identify an additional 255 methods, resulting in a total of 875 context queries. Please see the extended version [24] for details on the translation procedure and our website [1] for the expression-list.

4.2.2 Refining Control Predicate Identification. To refine the over-approximation of authorization checks, ACMiner again uses a semi-automated method of refinement, this time for control predicates. The process begins by running ACMiner with the refined context queries from Section 4.2.1. The expert then inspects lists of strings, fields, and methods for the *potential* control predicates, removing those not related to authorization.

From our exploration, we discovered a number of different categories of control predicates. Some we were aware of such as those involving *UID*, *PID*, *GID*, *UserId*, *AppId*, and package name. However, even within these categories, we discovered new fields, methods, and contexts in which checks are performed. We also discovered previously unknown categories of control predicates including those: (1) involving `SystemProperties`, (2) involving flags, (3) performing permission checks using the string equals method instead of the standard check permission methods, (4) checking for specific intent strings, and (5) checking boolean fields in specific classes. Finally, we discovered that a significant amount of noise was generated by the conditional statements of loops and sanity checks such as *null* checks. Using all of the information gained from the exploration of elements related to possible *control predicates*, we defined a filter that refines control predicate and reduces the overall noise.

Overall, the exploration took about 56 hours. We defined a 41-rule filter in about 16 hours (see our website [1] for the actual filter and the extended version [24] for the specification process). The application of the filter for Android AOSP 7.1.1 reduced what AC-Miner considered to be control predicates from 25808 to 3308. Such a significant reduction not only underscores the need for a filter but also makes the consistency analysis (Section 4.3) more feasible.

4.3 Consistency Analysis

The final step of ACMiner is consistency analysis of authorization checks for each entry point. In this paper, we perform consistency analysis of all entry points within a service. However, the methodology may work across multiple services, or even across different OEM firmwares. ACMiner performs consistency analysis by automatically discovering underlying correlative relationships between sets of authorization checks. Specifically, ACMiner adopts a targeted
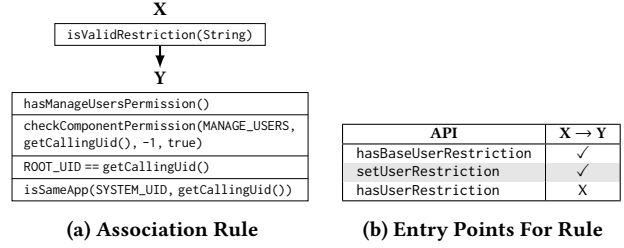
X

isValidRestriction(String)

Y

hasManageUsersPermission()
checkComponentPermission(MANAGE_USERS, getCallingUid(), -1, true)
ROOT_UID == getCallingUid()
isSameApp(SYSTEM_UID, getCallingUid())

API	X → Y
hasBaseUserRestriction	✓
setUserRestriction	✓
hasUserRestriction	X

(a) Association Rule   **(b) Entry Points For Rule**

Figure 5: (a) shows an association rule generated from the code in Figure 1 and (b) illustrates how the first 2 entry points satisfy the rule, while `hasUserRestriction` does not, indicating it contains one or more inconsistent authorization checks.

approach for association rule mining by using constraint-based querying. It then uses these association rules to predict whether an entry point's authorization checks are consistent. The results are presented to a domain expert for review.

Figure 5a shows an example association rule generated by AC-Miner from the code in Figure 1. X and Y are sets of authorization checks found in entry points. The rule states that if an entry point has check(s) from the set X, then it must also have the check(s) in set Y. ACMiner then uses these generated rules to identify potential vulnerabilities by reporting entry points that violate the learned rules (i.e., if an entry point has all of the checks in X, but it is missing checks in Y, then a violation occurs). For instance, Figure 5b shows the three entry points that match X for the rule in Figure 5a, out of which `hasUserRestriction` fails to satisfy the rule (it does not contain `checkManageUsersPermission`). On closer inspection, we discovered that all three entry points either set or get information about user restrictions. Moreover, the functionality of `hasUserRestriction` is nearly identical to `hasBaseUserRestriction`, which suggests `checkManageUsersPermission` is needed. As seen in these examples, ACMiner allows an expert to only compare entry points that are close in terms of their authorization checks, which is more precise than comparing all entry points to one another.

The remainder of this section discusses ACMiner's approach to efficiently discover these association rules and how ACMiner uses them to detect inconsistencies in authorization checks.

4.3.1 Association Rule Mining. Association rule mining discovers correlative relationships between sets of authorization checks, $A = \{i_1, i_2, \cdots, i_n\}$, across a set of entry points, $E = \{t_1, t_2, \cdots, t_m\}$ where each entry point in E contains a subset of the items in A. An association rule takes the form $X \implies Y$ where X (antecedent) and Y (consequent) are sets of authorization checks and X and Y are disjoint, i.e., $X \subseteq A$ and $Y \subseteq A$ and $X \cap Y = \emptyset$.

To avoid an excessive number of association rules, ACMiner uses two statistical constraints (support and confidence) to remove candidate association rules that are less than the thresholds minimum support (*minsup*) and minimum confidence (*minconf*). Let $\alpha(I)$ represent the set of entry points in E that contain the authorization checks $I \subseteq A$, i.e., $\alpha(I) = \{t \in E \mid \forall i \in I, i \in t\}$. The support of an association rule $X \implies Y$ is the probability that a set of authorization checks $Z = X \cup Y$ appears in the set of transactions E, i.e., $\sigma(Z) = \frac{|\alpha(Z)|}{|E|}$. The confidence of an association rule is an estimate of the conditional probability of the association rule $P(Y|X)$ where

$X \implies Y$ and can be calculated as $conf(X \implies Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$.

While association rule mining has been applied to similar problems by prior work [28], the large transaction size (i.e., number of authorization checks in an entry point) in our problem domain makes general association rule mining algorithms infeasible due to their exponential complexity. Therefore, ACMiner uses two main optimizations to reduce the complexity to polynomial time.

First, ACMiner only generates a subset of the association rules called closed association rules [40]. An association rule $X \implies Y$ is closed if $X \cup Y$ is a frequent closed itemset. A frequent closed itemset is a set of authorization checks $C \subseteq A$ where the support of C is greater than *minsup* and there does not exist a superset C' that has the same support as C. C is closed iff $\beta(\alpha(C)) = C$ where $\beta(T)$ represents the largest set of common authorization checks present in the entry points T where $T \subseteq E$, i.e., $\beta(T) = \{i \in A \mid \forall t \in T, i \in t\}$. Note that only mining frequent closed itemsets is lossless, because all frequent itemsets can be generated from the set of frequent closed itemsets, as proven by Zaki and Hsiao [45]. Our proof that closed association rules also do not result in information loss can be found in the extended version [24].

Second, ACMiner generates closed association rules in a targeted manner by placing constraints on the authorization checks that appear in the antecedent of the association rule. Since the goal of consistency analysis is to predict whether an entry point's implementation of authorization checks is consistent, we are only interested in generating association rules where the antecedent of the association rule is a subset of the entry point's authorization checks (i.e., $X \subseteq A_j$ where A_j is the authorization checks of entry point e_j). For example, consider $A_j = \{i_1, i_2, i_3\}$ and we have two frequent closed itemsets $\{i_1, i_2, i_3, i_4\}$ and $\{i_5, i_6, i_7\}$. The association rule $\{i_1, i_2, i_3 \implies i_4\}$ is useful, as it could potentially hint that the authorization checks in A_j is inconsistent and should also contain i_4. However, all of the association rules from the set $\{i_5, i_6, i_7\}$ do not provide additional information on the consistency of authorization checks in A_j, as the two sets are disjoint.

Further, assuming that the authorization checks that are present within an entry point are correct, we can force the antecedent to be constant. In particular, when generating association rules from a frequent closed itemset I for an entry point A_j, we set $X = A_j \cap I$ and can generate association rules by varying the items in Y. If we reduce the authorization checks in X, then we are making the rule less relevant to the consistency of the entry point A_j while keeping X constant only produces the most relevant association rules.

4.3.2 Inconsistency Detection and Output Generation. ACMiner uses the association rules discussed in Section 4.3.1 to analyze the consistency of an entry point's authorization checks. To minimize the amount of manual effort required to verify the presence of an inconsistency, we ensure the output only contains high confidence rules by setting *minconf* to 85%. Moreover, as we want the authorization checks in the consequent of an association rule to be formed by at least 2 entry points, we set the *minsup* to $\frac{2}{|E|}$.

While generating the association rules, we mark an entry point's authorization checks as consistent if there exists a frequent closed itemset that contains the exact same authorization checks as the entry point, as this hints that the entry point's authorization checks are consistent with another entry point's authorization checks. In particular, entry point e_j's authorization checks A_j is consistent iff $\exists C \in A | C = A_j \wedge \beta(\alpha(C)) \wedge \sigma(C) \geq \frac{2}{|E|}$. In contrast, we mark an entry point's authorization checks as potentially inconsistent if an association rule exists where the entry point's authorization checks are the antecedent of the rule and the consequent is not empty (i.e., $\exists X \implies Y | X \subseteq A_j \wedge Y \neq \emptyset$).

ACMiner outputs an HTML file for the domain expert to review for each association rule representing a potentially inconsistent entry point. The HTML file contains the set of supporting authorization checks for the association rule (i.e. the antecedent), the set of authorization checks being recommended by the association rule (i.e. the consequent), and the 3 or more entry points that contain the authorization checks of the association rule. This group of entry points can be subdivided into two sets: the target entry point and the supporting entry points. The target entry point is the entry point the association rule has identified as being inconsistent, i.e., the entry point the association rule is recommending additional authorization checks for. The supporting entry points are the 2 or more entry points where the recommended authorization checks occur. Note that the supporting authorization checks occur in both the target and the supporting entry points. To aid the review, the HTML file also contains the set of all authorization checks from the target entry point that do not occur in the supporting authorization checks and for all authorization checks, the method in the Android source code where the checks occur.

To reduce the manual effort required to confirm inconsistencies, we perform two post-processing techniques. First, we remove association rules where |*recommended authorization checks*| >= 5 * |*supporting authorization checks*|, since association rules that contain 100 authorization checks which imply 500 authorization checks is improbable. Second, we remove any remaining association rules that have over 100 recommended authorization checks as such association rules are unlikely to indicate inconsistencies, and the domain expert may not be able to evaluate such rules in a reasonable amount of time.

5 EVALUATION

We evaluated ACMiner by performing an empirical analysis of the system services in AOSP version 7.1.1_r1 (i.e., API 25) 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, OpenJDK 1.8.0_141, and Python 2.7.6.

We used ACMiner to mine the authorization checks of all the entry points from this build of AOSP and perform consistency analysis, as described in Section 4. Finally, we manually analyzed the inconsistencies using a systematic methodology to identify high risk (i.e., easily exploitable) and high impact vulnerabilities, and developed proof-of-concept exploits to validate our findings. Our evaluation is guided by the following research questions:

RQ1 *Does ACMiner reduce the effort required by the domain expert in terms of the entry points that need to be analyzed?*

RQ2 *Do the inconsistencies identified by ACMiner help a domain expert in finding security vulnerabilities?*

RQ3 *What are the major causes behind inconsistencies that do not resolve to security vulnerabilities?*

RQ4 *Is ACMiner more effective than prior work at detecting vulner-*

```
High Impact  Low Impact

┌──────────────────────────────┐   ┌──────────┬──────────┐
│ 4004 Total Entry Points      │   │ 7 Entry  │ 1 Entry  │ High Risk
│ ┌──────────────────────────┐ │   │ Points   │ Point    │
│ │ 1995 Entry Points with   │ │   ├──────────┼──────────┤
│ │ Authorization Logic      │ │   │ 12 Entry │ 8 Entry  │ Low Risk
│ │ ┌──────────────────────┐ │ │   │ Points   │ Points   │
│ │ │ 246 Entry Points     │ │ │   └──────────┴──────────┘
│ │ │ with Association     │ │ │
│ │ │ Rule(s)              │ │ │
│ │ └──────────────────────┘ │ │
│ └──────────────────────────┘ │
└──────────────────────────────┘
    (a) Entry Point Reduction          (b) Risk vs. Impact
```
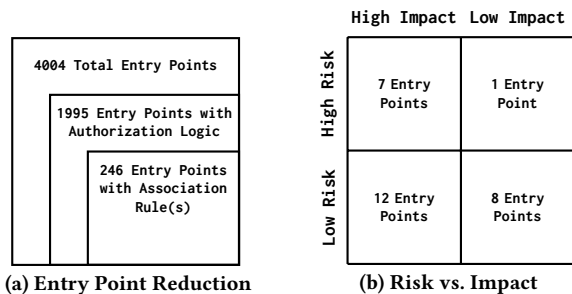
Figure 6: (a) shows how ACMiner is able to reduce the scope of the AOSP 7.1.1 system code a domain expert needs to evaluate and (b) breaks down the 28 vulnerabilities in terms of risk and impact.

> abilities in system services?
>
> **RQ5** *What is the time required by ACMiner to analyze all the system services in a build of Android?*

We now highlight the salient findings from our evaluation, followed by the categorization of the discovered vulnerabilities. The categorization of non-security inconsistencies, developed via a systematic manual analysis of our results, is described in Section 6.

5.1 Evaluation Highlights

As shown in Figure 6a, ACMiner reduces the total number of entry points that need to be manually analyzed down to just 246 entry points with inconsistent authorization checks, a 94% reduction (**RQ1**). As a result, ACMiner significantly enhances a domain expert's ability to evaluate the consistency of access control enforcement in the Android system by minimizing the effort required. Further, ACMiner took approximately 1 hour and 16 minutes to mine the authorization checks of all entry points from the system image of the AOSP build, and spent an additional 30 minutes producing the HTML files for the association rules that represent potentially vulnerable entry points. While ACMiner could be optimized further, time taken by ACMiner is a feasible cost, given its scalability benefits over a fully manual analysis (**RQ5**).

On manually analyzing the 246 entry points, we discovered a total of 28 entry points containing security vulnerabilities (**RQ2**). As Figure 6b illustrates, these 28 vulnerabilities were then classified in terms of their risk (i.e., the ease of exploiting a vulnerability) as well as the impact (i.e., the gravity of the consequence of an exploited vulnerability). Using this criteria, we found 7 vulnerabilities that were high risk as well as high impact, 1 vulnerability that was high risk only, 12 vulnerabilities that were high impact only, and 8 vulnerabilities that were low in terms of both risk and impact. All 28 vulnerabilities have been submitted to Google. So far, 2 of our vulnerabilities have been assigned a "moderate" Android Security Rewards (ASR) severity level, which is generally awarded to bypasses in access control mechanisms (e.g., restrictions on constrained processes, or general bypasses of privileged processes [22]). In Section 5.2, we categorize these 28 vulnerabilities according to their effect; however, we only discuss a few of these vulnerabilities in depth, due to space constraints.

ACMiner is significantly more effective than prior work at identifying inconsistent authorization checks. For instance, ACMiner is able to identify 875 unique context queries using the semi-automated approach described in Section 4.2.1, a drastic 2552% improvement over the original 33 context queries that encompass a majority of the context queries considered by Kratos [38]. Further, while AceDroid [2] is more comprehensive than Kratos in its identification of Android's authorization checks, it relies on a manually defined list of context queries, which is insufficient. That is, as described in Section 4.2.1, our thorough attempts at identifying context queries through manual observation alone resulted in the identification of only 620 context queries, 71% of the total context queries that ACMiner is able to find using its semi-automated approach. Thus, while AceDroid does not provide quantitative information on its set of context queries, we can certainly say that it is not as complete as ACMiner in its identification of Android's authorization checks. Indeed, the context query isValidRestriction in Figure 1 is an example of a context query that neither AceDroid nor Kratos was able to identify, and in fact, one that we missed in our manual definition of Android's authorization checks. However, through the general expressions, ACMiner was able to identify isValidRestriction as a context query and the vulnerability outlined in Figure 1. Moreover, neither AceDroid nor Kratos makes any mention of the App Ops restrictions in their definition of Android's authorization checks. Yet ACMiner is able to identify 2 vulnerabilities relating to the App Ops restrictions (see Section 5.2). While a full empirical comparison with Kratos and AceDroid is infeasible due to the lack of source code access, our evaluation demonstrates that ACMiner makes significant advancements to existing work in terms of the coverage of the authorization checks, making the consistency analysis as complete as possible (**RQ4**).

Finally, ACMiner produced 453 association rules denoting inconsistent authorization checks in 246 entry points. Some entry points had more than one inconsistency. Furthermore, while some inconsistencies were indeed valid security vulnerabilities (30/453), others were a result of irregular coding practices in Android (25/453) or indicative of ACMiner's limitations in terms of analyzing the semantics of the authorization checks (**RQ3**). The limitations identified via our analysis point to hard problems in analyzing Android's access control logic and motivate future work.

5.2 Findings

Table 2 describes the vulnerabilities discovered through our analysis of Android 7.1.1 with ACMiner. On manually analyzing the inconsistent entry points produced by ACMiner, we discovered 28 entry points that represent security vulnerabilities. While most of these entry points represent one vulnerability each, two entry points (i.e., getLastLocation and setStayOnSetting, vulnerabilities 15 and 16 in Table 2 respectively) each led us to clusters of multiple identically vulnerable entry points, as described later in this section. For simplicity, we count each cluster as a single vulnerability.

We group the vulnerabilities into the following 3 categories: (1) user separation and restrictions, (2) App Ops, (3) and pre23. This categorization is based on the subsystems affected by the vulnerabilities (e.g., App Ops), as well as the characteristics they have in common (e.g., pre23). Additionally, some vulnerabilities that are hard to classify have been categorized as (4) miscellaneous. **VC1: Multi-user Enforcement:** As shown in Table 2, a majority (i.e., 14) of the vulnerabilities affect Android's separation among users (i.e. user profiles in Android's multi-user enforcement [23]).

Table 2: Description of vulnerabilities, along with the services in which they are present

Associated Entry Point (Service)	Vulnerability Description
VC1: Multi-user Enforcement	
1. getInstalledApplications (PMS)	Missing the enforceCrossUserPermission check, allowing any app on one user profile to discover apps installed on other profiles.
2. getPackagesHoldingPermissions (PMS)	Missing enforceCrossUserPermission, allowing any app on one user profile to get sensitive permission information about other profiles.
3. hasUserRestriction (UMS)	Missing the hasManageUsersPermission check, which checks for the permission MANAGE_USERS, is missing, allowing any user to discover the restrictions on their own and other user profiles.
4. checkUriPermission (AMS)	Missing the handleIncomingUser check that verifies if a user can operate on behalf of another, allowing any user access to content provider URIs belonging to another user, so long as the app making the request has access to the content provider.
5. grantUriPermission (AMS)	Missing the handleIncomingUser check, with similar implications as checkUriPermission.
6. killPackageDependents (AMS)	Missing the handleIncomingUser check, allowing any user to kill the apps and background processes of another user.
7. setUserProvisioningState (DPMS)	Missing the enforceFullCrossUsersPermission check, enabling any user to change another user profile's state.
8. setDefaultBrowserPackageName (PMS)	Missing the enforceCrossUserPermission check, enabling any user to set the default browser of any other user.
9. updateLockTaskPackages (AMS)	Missing handleIncomingUser, enabling any user to modify the apps that may be permanently pinned to the screen in a kiosk like venue.
10. installExistingPackageAsUser (PMS)	Does not check if the target user exists, allowing any user to install apps on user profiles that may be created at a later time.
11. setApplicationHiddenSettingAsUser (PMS)	Does not check if the target user exists, allowing any user to hide apps on user profiles that may be created in the future.
12. setAlwaysOnVpnPackage (CS)	Does not check for the no_config_vpn user restriction, allowing a managed user to set its always on VPN to another application.
13. setWallpaperComponent (WPMS)	Missing the two checks isSetWallpaperAllowed and isWallpaperSupported, allowing a managed user to change their wallpaper.
14. startUpdateCredentialsSession (ACMS)	Missing checks canUserModifyAccounts and canUserModifyAccountsForType, allowing a user to trigger an update for the credentials of online accounts like Google and Facebook even when restricted.
VC2: App Ops	
15. noteProxyOperation (AOMS)	Missing the verifyIncomingUid check, which checks for a signature permission, allowing non-system apps to call this entry point.
16. getLastLocation (LMS)	A majority of the entry points in the LMS use the AppOpsManager check checkOp, which is not intended for security, instead of the security check noteOp. getLastLocation uses the correct check.
VC3: Pre23	
17. setStayOnSetting (POMS)	On systems with API 23 or above, the pre23 protection level allows any permission to be automatically granted to non-system apps built targeting the API 22 or below. This vulnerability allows non-system apps to access 6 additional entry points protected by the WRITE_SETTINGS *signature* permission, as WRITE_SETTINGS also has the pre23 protection level.
VC4: Miscellaneous	
18. unbindBackupAgent (AMS)	Missing check for if caller is performing a backup, allowing any app to disrupt the backup process of another app.
19. getPersistentApplications (PMS)	Missing system UID check, allowing any non-system app to discover what apps and services permanetly run in the background.
20. logEvents (MLS)	Incorrect check for permission CONNECTIVITY_INTERNAL, should check DUMP when writing sensitive data to logs.
21. getMonitoringTypes (GHS)	Missing check checkPermission, allowing a caller access both fine and coarse levels of geofence location data.
22. getStatusOfMonitoringType (GHS)	Missing check checkPermission, with similar implications as getMonitoringTypes.
23. setApplicationEnabledSetting (PMS)	Missing isPackageDeviceAdmin check, allowing an app to disable an active administrator app.
24. setComponentEnabledSetting (PMS)	Missing isPackageDeviceAdmin check, allowing an app to disable components of an active administrator app.
25. convertFromTranslucent (AMS)	Missing check for enforceNotIsolatedCaller, allowing a isolated process to affect the transparency of windows.
26. notifyLockedProfile (AMS)	Missing check for enforceNotIsolatedCaller, allowing an isolated process to trigger a retrun to the home screen.
27. setActiveScorer (NSS)	Missing BROADCAST_NETWORK_PRIVILEGED permission check which is always paired with the SCORE_NETWORKS permission check.
28. getCompleteVoiceMailNumberForSubscriber (PSIC)	Incorrect check for permission CALL_PRIVILEGED instead of the READ_PRIVILEGED_PHONE_STATE results in coarse-grained enforcement.

AMS=ActivityManagerService; AOMS=AppOpsManagerService; CS=ConnectivityService; DPMS=DevicePolicyManagerService; LMS=LocationManagerService; PMS=PackageManagerService; POMS=PowerManagerService; UMS=UserManagerService; WPMS=WallpaperManagerService; PSIC=PhoneSubInfoController; ACMS=AccountManagerService; MLS=MetricsLoggerService; GHS=GeofenceHardwareService; NSS=NetworkScoreService

These can be further divided into four subcategories based on how they may be exploited: (1) leaking user information across users, (2) operating across users, (3) modifying user settings before a user exists, and (4) allowing users to bypass restrictions.

1. Leaking Information to Other Users: In 5 entry points (i.e., 1→5 in Table 2), the lack of checks leads to potential leaks of security-sensitive information to other users. For instance, using the vulnerable entry point getInstalledApplications in the PackageManagerService, any user can learn of the applications another user has installed, as the entry point does not enforce any checks other than checking if the user being queried exists. Similarly, the entry point hasUserRestriction in the UserManagerService, previously used as the motivating example, is not protected with the *signature* level permission MANAGE_USERS, which is present in the similar hasBaseUserRestriction entry point. This omitted authorization check allows a user to know of the restrictions placed on other users, which is security-sensitive information that should not be public. The entry points getPackagesHoldingPermissions, checkUriPermission and grantUriPermission similarly leak sensitive information.

We experimentally confirmed the existence of both the vulnerabilities in hasUserRestriction and getInstalledApplications in Android 7.1.1 as well as Android 8.1. We have submitted bug reports to Google and received "moderate" ASR severity level for both the bugs. Further, we confirmed that the vulnerability in getPackagesHoldingPermissions was fixed in Android 8.1. As a result, we could not submit it to Google's bug program, which only considers bugs affecting the latest version of Android. All remaining vulnerabilities have been reported to Google.

2. Operating Across Users: Missing authorization checks in 4 entry points (i.e., 6→9 in Table 2) allow users to bypass multi-user restrictions and perform sensitive operations on behalf of other users. For example, we discovered that the entry point killPackageDependents takes in a *userId* as an argument but does not actually verify whether the calling user is allowed to perform operations on behalf of the supplied *userId*. This allows *any user to kill the apps and background processes of any other user*. A similar flaw in entry point setUserProvisioningState enables any user to set the state of any other user profile to states such as "managed", "unmanaged", or "finalized". Such changes may be dangerous. For instance, a user may be able to set their managed enterprise profile to unmanaged, releasing it from the administrator's control.

Fortunately, all four entry points described in this category can only be called from apps installed on the system image (i.e., are protected by authorization checks that ensure this). This indirectly mitigates some damage, by making the vulnerabilities difficult to exploit from a third-party app. However, capability leaks in privileged

```
1   /** Do a quick check for whether an application might be
2    * able to perform an operation. This is not a security
3    * check; you must use noteOp or startOp for your actual
4    * security checks, which also ensure that the given uid
5    * and package name are consistent. ... */
6   int checkOp(int op, int uid, String packageName) {...}
```

Figure 7: The comment above `checkOp` from the class `AppOpsManager` that states it should not be used as a security check.

apps may allow such vulnerabilities to be exploited by third-party apps, as prior work has demonstrated [25, 43]. All of these vulnerabilities have been reported to Google.

3. Modifying User Settings Before A User Exists: Both the entry points `installExistingPackageAsUser` and `setApplicationHiddenSettingAsUser` do not perform the authorization check `exists`, which verifies if a the *userId* passed into the entry points represents a valid user. Without this check, it is possible for a caller to install an app for a non-existent user or hide an app from a non-existent user. Thus, when the user for whom this change was made is actually created, these settings will already be in place. These entry points are only callable from systems apps; however, system apps may be compromised or may leak capabilities, and the *exists* check needs to be in place to prevent system apps from being tricked into allowing users to install apps in profiles that have yet to be created (e.g., installing apps in a future enterprise profile). We have submitted these vulnerabilities to Google.

4. Allowing Users to Bypass Restrictions: Vulnerabilities in entry points 12→14 from Table 2 allow a user to perform operations despite the restrictions placed on the user profile. For instance, the entry point `setAlwaysOnVpnPackage` does not check for the restriction `no_config_vpn`, allowing a managed user to set the always on VPN for the user profile to another application, effectively switching VPN connections. The entry points `setWallpaperComponent` and `startUpdateCredentialsSession` have similar vulnerabilities. All of these vulnerabilities have been reported to Google.

VC2: App Ops: ACMiner identified weaknesses related to App Ops. One such vulnerability lies in the `noteProxyOperation` of the `AppOpsService`. The entry point makes a note of an application performing some operation on behalf of some other application through IPC. However, unlike other entry points in the `AppOpsService`, `noteProxyOperation` is missing the authorization check `verifyIncomingUid` which includes a check for the *signature* level permission `UPDATE_APP_OPS_STATS`. Without `verifyIncomingUid`, it is possible for any non-system app to use `noteProxyOperation` to query the restrictions a user has placed on other apps, thus retrieving information that should not be available to non-system apps.

We discovered a set of identical vulnerabilities in App Ops through our analysis of the `getLastLocation` entry point in the `LocationManagerService`, which ACMiner pointed out as having inconsistent authorization checks. The `getLastLocation` entry point calls the authorization check `reportLocationAccessNoThrow` which performs the check `noteOpNoThrow` from the `AppOpsManager`, a wrapper for the `AppOpsService`. ACMiner correctly identified the use of `noteOpNoThrow` as an inconsistency since a majority of the entry points (9) in `LocationManagerService` use the authorization check `checkLocationAccess` which performs the check `checkOp` from the `AppOpsManager`. However, as Figure 7 shows, the comment above the `checkOp` method clearly states that `checkOp` should not be used

```
<permission android:name="android.permission.WRITE_SETTINGS"
  android:protectionLevel="signature|preinstalled|appop|pre23" />
```

Figure 8: The permission protection levels of `WRITE_SETTINGS` in the `AndroidManifest.xml` file [3]

to perform a security check. Instead, one of the various forms of `noteOp` should be used. This implies that all 9 entry points using the context query `checkLocationAccess` suffer from a vulnerability, and that the use of `reportLocationAccessNoThrow` in `getLastLocation` is actually appropriate. This instance demonstrates an interesting outcome of the use of consistency analysis in ACMiner. That is, our use of consistency analysis in ACMiner is also useful in identifying instances, where the majority of the related entry points are vulnerable. As described previously, for simplicity, we count this cluster of vulnerable entry points as a single vulnerability, which has been submitted to Google.

VC3: Pre23: ACMiner identified a group of vulnerabilities related to Android's *pre23* permission protection level. The entry point `setStayOnSetting` in the `PowerManagerService` uses the authorization check `checkAndNoteWriteSettingsOperation`, which checks if an app has the *signature* level permission `WRITE_SETTINGS`. Permissions with the *signature* protection level can only be granted to system apps (i.e., apps that were originally packaged with the system image). However, as shown in Figure 8, `WRITE_SETTINGS` has an additional protection level of *pre23*, which has an interesting effect on Android versions 6.0 or above (i.e., API 23 or above). It allows permissions marked as *pre23* to be granted to non-system apps that target API 22 or below [21]. Thus, as a result of the improperly defined permission protection levels for `WRITE_SETTINGS`, the `pre23` grants non-system apps access to a `signature` level permission.

The damage resulting from the *pre23* vulnerability is not restricted to the entry point `setStayOnSetting`. A simple search for the use of the permission `WRITE_SETTINGS` in the authorization checks ACMiner mined for all entry points in the system revealed 13 additional entry points checking for the permission `WRITE_SETTINGS`, 6 of which can be called from a non-system app using the *pre23* vulnerability (i.e., these 6 entry points are not protected with any other *signature* permission). Of the 6, the following 5 entry points deal with tethering and are located in the `ConnectivityService`: `setUsbTethering`, `stopTethering`, `startTethering`, `tether`, and `untether`. The last `setWifiApEnabled` was located in the `WifiServiceImpl` and allows a caller to set some WIFI access point configuration, causing the device to connect or disconnect from any WIFI access point the caller provides. These entry points are clearly more important to protect than `setStayOnSetting`, and an adversary may be able to do considerable damage by exploiting them. We do not count these entry points in our initial list of 28 vulnerabilities. All entry points affected by the *pre23* vulnerability have been submitted to Google.

VC4: Miscellaneous Vulnerabilities: ACMiner also identified 11 vulnerabilities related to information leaks, denial of service, disabling of administrator apps, and a mixture of other minor vulnerabilities. All of these vulnerabilities have been reported to Google.

6 NON-SECURITY INCONSISTENCIES

ACMiner identified 423 inconsistencies (i.e., rules) that did not represent vulnerabilities. Aside from the 20 rules that were caused by easily fixed bugs in ACMiner, we resolve these non-security incon-

Table 3: Non-security Inconsistencies

Type of Inconsistency	Number of Rules
1. Shortcuts to Speed-Up Access	7
2. Fixing Access Bugs	2
3. Potential Vulnerabilities	16
4. Difference in Functionality	189
5. Checks With Different Arguments	66
6. Noise in Captured Checks	53
7. Restricted to Special Callers	37
8. Semantic Groups of Checks	23
9. Equivalent Checks	10

sistencies to their likely causes, and classify them into 9 categories, shown in Table 3 (**RQ3**). The first three categories point to irregular coding practices, i.e., (1) inconsistent application of short-cuts to speed up access, (2) access bugs or discrepancies in how the permission should be used as per the documentation, or (3) inconsistent application of hard-coded checks that would potentially lead to vulnerabilities on future updates. The remaining 6 categories point to issues that could be corrected by engineering improvements to ACMiner, such as considering semantic equivalence between authorization checks, or the integration of call graph comparison and method-name comparison to mitigate the analysis of functionally different entry points. We provide additional details on all of the 9 categories in the extended version [24].

7 LIMITATIONS

While ACMiner is effective at discovering inconsistencies that lead to vulnerabilities, consistency analysis has a general limitation, i.e., it may not discover vulnerabilities that are consistent throughout code. Further, for precision, ACMiner does not handle the invocation of secondary entry points, i.e., calls to entry points from within other entry points. ACMiner omits the *ordering* of the authorization checks and hence does not identify improper operator use in *control predicates*, which we plan to explore in the future. Moreover, ACMiner's semi-automated analysis requires the participation of domain experts. However, as Section 5 demonstrates, our design significantly reduces manual effort in contrast with the manual validation of system services. As we have already analyzed AOSP version 7.1.1, only minor input is needed to analyze newer versions or vendor modifications. Finally, ACMiner shares the general choices made by Android static analysis techniques for precision, i.e., it does not consider native code, or runtime modifications (e.g., reflection, dynamic code loading, Message Handlers).

8 RELATED WORK

ACMiner addresses a problem that has conceptual origins in prior work on authorization hook validation for traditional systems. Early investigations targeted the Linux Security Modules (LSM) hook placement in the Linux kernel, using techniques such as type analysis using CQUAL [47], program dominance [48], and dynamic analysis to create authorization graphs from control flow traces [12, 29]. As the lack of ground truth is a general challenge for hook validation, prior work commonly uses consistency analysis [12, 29, 41]. Closest to our work is AutoISES [41], which infers security specifications from code bases such as the Linux kernel and Xen. However, AutoISES assumes a known set of security functions or security-specific data structures, whereas ACMiner identifies a diverse set of authorization checks in the Android middleware.

The closest to our approach is prior work on authorization

hook validation in the Android platform, i.e., Kratos [38] and AceDroid [2]. ACMiner distinguishes itself from Kratos and AceDroid through its deep analysis of Android's system services, and its significantly improved coverage of Android's authorization checks.

Kratos [38] compares a small subset of Android's authorization checks across entry points of the same system image to look for inconsistent checks between different system services. ACMiner does not analyze for consistency across services. Instead, we hypothesize that entry points within a single service are similar in purpose, and hence, analyze the consistency of the authorization checks by comparing the entry points of every system service with other entry points in the same service. Further, ACMiner's semi-automated approach for identifying authorization checks results in a 2552% improvement over Kratos' manually-curated list (Section 5).

Similarly, AceDroid [2] evaluates the consistency of the authorization checks among different vendor-modified Android images, and hence differs from ACMiner in terms of its objective. AceDroid makes key improvements over Kratos, as it considers various non-standard context queries not considered by Kratos. However, AceDroid also relies on a manually-defined list of context queries, which may produce only approximately 71% of the context queries that ACMiner is able to find (Section 5).

Finally, recent literature is rich with static and dynamic program analysis of third-party Android apps targeted at privacy infringement [13, 20, 26], malware [17, 27, 46], as well as vulnerabilities [9, 14]. As the Android platform and apps use similar programming abstractions, researchers have applied these tools and techniques to the platform code, e.g., for providing a mapping between APIs and corresponding permissions [5, 8, 18] or automatically identifies privacy-sensitive sources and sinks [4]. Moreover, prior work has also studied the platform code, to analyze OEM apps for capability leaks (e.g., Woodpecker [25] and SEFA [43]), discover privilege escalation on update vulnerabilities (e.g., Xing et al. [44]), or uncover gaps in the file access control policies in OEM firmware images (e.g., Zhou et al. [49]). While ACMiner shares a similar objective, unlike prior work, ACMiner provides an automated and systematic investigation of core platform services.

9 CONCLUSION

This paper provides an approach for the systematic and in-depth analysis of a crucial portion of Android's reference monitor, i.e., its system services. We design ACMiner, a static analysis framework that comprehensively identifies a diverse array of authorization checks used in Android's system services, and then adapts the well-founded technique of association-rule mining to detect inconsistent access control among service entry points. We discover 28 security vulnerabilities by analyzing AOSP version 7.1 using ACMiner, and demonstrate significantly higher coverage of checks than the state of the art. Our work demonstrates the feasibility of in-depth analysis of Android's system services with ACMiner, as it significantly reduces the number of entry points that must be analyzed, from over 4000 with millions of lines of code to a mere 246.

are those of the authors and do not reflect the views of the funders.

REFERENCES

[1] 2019. ACMiner Project Website. https://wspr.csc.ncsu.edu/acminer/.
[2] 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)*.
[3] AndroidXref. 2019. WRITE_SETTINGS permission in AndroidManifest.xml. http://androidxref.com/7.1.1_r6/xref/frameworks/base/core/res/AndroidManif est.xml#1865. Accessed Jan. 10, 2019.
[4] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*.
[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] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android.. In *USENIX Security Symposium*.
[8] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2014. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android. *IEEE Transactions on Software Engineering (TSE)* 40, 6 (June 2014).
[9] 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*.
[10] Andrew Dalton. 2019. Android powers 2 billion devices around the world. https://www.engadget.com/2017/05/17/android-powers-2-billion-devices-around-t he-world/. Accessed Jan. 10, 2019.
[11] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.
[12] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. 2002. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[13] 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*.
[14] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proceedings of the USENIX Security Symposium*.
[15] 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)*.
[16] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding Android Security. *IEEE Security & Privacy Magazine* 7, 1 (January/February 2009).
[17] 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)*.
[18] 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)*.
[19] 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*.
[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. 2019. protectionLevel. https://developer.android.com/reference/android/ R.attr#protectionLevel. Accessed Jan. 10, 2019.
[22] Google. 2019. Security Updates and Resources. https://source.android.com/secur ity/overview/updates-resources. Accessed Jan. 10, 2019.
[23] Google. 2019. Supporting Multiple Users. https://source.android.com/devices/tec h/admin/multi-user. Accessed Jan. 10, 2019.
[24] 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. http://arxiv.or g/abs/1901.03603. (Jan. 2019). arXiv:1901.03603

[25] 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)*.
[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. As-Droid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[28] JeeHyun Hwang, Tao Xie, Vincent Hu, and Mine Altunay. 2010. Mining likely properties of access control policies via association rule mining. *Data and Applications Security and Privacy XXIV* (2010), 193–208.
[29] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. 2004. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *Transactions on Information and System Security* 7, 2 (May 2004), 175–205.
[30] 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)*.
[31] Ond∨rej Lhoták. 2007. Comparing Call Graphs. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*.
[32] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC 03)*. Springer Berlin Heidelberg, Warsaw, Poland, 153–169.
[33] Travis McCoy. 2019. How the World Bank is mobilizing their workforce with Android. https://www.blog.google/topics/connected-workspaces/how-world-b ank-mobilizing-their-workforce-android/. Accessed Jan. 10, 2019.
[34] Mark Milian. 2019. U.S. government, military to get secure Android phones. http://www.cnn.com/2012/02/03/tech/mobile/government-android-phones/in dex.html. Accessed Jan. 10, 2019.
[35] Adwait Nadkarni and William Enck. 2013. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[36] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. Ad-Droid: Privilege Separation for Applications and Advertisers in Android. In *Proc. of the ACM Symposium on Information, Computer and Communications Security*.
[37] Steve Ranger. 2019. The world's most secure smartphones – and why they're all Androids. http://www.zdnet.com/article/the-worlds-most-secure-smartphones -and-why-theyre-all-android/. Accessed Jan. 10, 2019.
[38] 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)*.
[39] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. 2014. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of the USENIX Operating Systems Design and Implementation (OSDI)*.
[40] Laszlo Szathmary. 2006. *Symbolic Data Mining Methods with the Coron Platform*. Ph.D. Dissertation. Université Henri Poincaré-Nancy I.
[41] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *Proceedings of the USENIX Security Syposium*.
[42] 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*.
[43] 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.
[44] 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*.
[45] Mohammed J Zaki and Ching-Jui Hsiao. 2002. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *Proceedings of the 2002 SIAM International Conference on Data Mining*.
[46] 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)*.
[47] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. 2002. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the USENIX Security Symposium*.
[48] Xiaolan Zhang, Trent Jaeger, and Larry Koved. 2004. Applying Static Analysis to Verifying Security Properties. In *Proceedings of the Grace Hopper Celebration of Women in Computing Conference (GHC)*.
[49] 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*.