AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection

Yousra Aafer*, Jianjun Huang*, Yi Sun*, Xiangyu Zhang*, Ninghui Li* and Chen Tian†
*Purdue University
†Futurewei Technologies
{yaafer, huang427, sun624}@purdue.edu, {xyzhang, ninghui}@cs.purdue.edu, Chen.Tian@huawei.com

***Abstract*—The Android framework has raised increased security concerns with regards to its access control enforcement. Particularly, existing research efforts successfully demonstrate that framework security checks are not always consistent across app-accessible APIs. However, existing efforts fall short in addressing peculiarities that characterize the complex Android access control and the diversity introduced by the heavy vendor customization. In this paper, we develop a new analysis framework *AceDroid* that models Android access control in a path-sensitive manner and normalizes diverse checks to a canonical form. We applied our proposed modeling to perform inconsistency analysis for 12 images. Our tool proved to be quite effective, enabling to detect a significant number of inconsistencies introduced by various vendors and to suppress substantial false alarms. Through investigating the results, we uncovered high impact attacks enabling to write a key logger, send premium sms messages, bypass user restrictions, perform a major denial of services and other critical operations.**

I. Introduction

Over the near-decade since its introduction in 2008, the Android operating system has been receiving an unprecedented success, overshadowing the market share of other competing mobile operating systems. According to [6], on average 1.5 million Android devices are being activated every day. However, this stunning success does not come at no cost. The number of identified vulnerabilities at various Android layers has soared in the recent years.

Of particular interest are the framework vulnerabilities which can allow attackers to easily access sensitive and privileged resources without proper access control. In fact, the Android framework has raised increased security concerns with regards to its access control enforcement. Several research works have questioned the effectiveness and consistency of the complex Android framework access control and demonstrated its weaknesses with detected vulnerabilities [33], [8]. The difficulty of determining if critical resources are sufficiently protected lies in the lack of an almighty oracle to determine the access control needed for a given resource. Therefore, a popular *approximate* solution is to compare the access control enforced across multiple instances of the same resource and report inconsistencies as potential vulnerabilities. For instance, Kratos [33] compares the set of explicit security checks (i.e., permissions, UIDs, package names and thread status) in multiple APIs leading to the same resource within the same image. DroidDiff [8] compares the security configurations employed by the different framework releases to detect vulnerabilities.

However, the simple modeling of access control in existing techniques [33], [8] does not reflect the nature of the problem and hence can hardly meet the challenges imposed by the increasing complexity of frameworks and the large number of vendor custimizations. *First*, most access control checks are essentially just conditional statements. There are often many different ways of composing the access control conditional statements for a given resource as long as they provide the same level of protection to the resource. These versions are not only syntactically different, but also semantically different in many cases (e.g., invoking different APIs). This is particularly true for vendor customizations. Developers from different vendors tend to choose diverse ways to implement equivalent protection for a resource. For instance, to check if a calling app is running with system privilege, the developer might compare its UID with the System UID (i.e., 1000), compare its signature with the *"android"* package signature, or compare its shared user id with *"android.uid.system"*. In addition to API calls that are explicitly related to access control, developers may use local variables and flag variables, whose relations to access control are implicit, further compounding the situation. *Second*, the access control to a resource may require multiple checks. Most existing techniques do not model the relations of these checks, but rather consider the entire set of checks as the demanded protection. However in practice, these checks are conjoined and/or disjoined in various fashions dictated by implementation. Such relations need to be precisely modeled in order to have legitimate comparison of access control. As such, a simple approach as that in Kratos [33], which collects and compares the set of explicit invocations to access control APIs in a path-insensitive fashion, does not model the essence of the problem and hence may miss vulnerabilities and produce many false positives (Section VI-F).

In this paper, we propose a normalization technique for access control checks. It identifies diverse security checks through comprehensive modeling. It further normalizes these checks and their correlations to a canonical form so that the different forms of access control checks that denote the same meaning can be correctly recognized. Particularly, we classify all the access control related framework modules and classes,

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23121
www.ndss-symposium.org

which contain properties that may be used in access control checks, to two main categories: the *app-specific* checks and *user-specific* checks. The former checks if the app that tries to access the resource has the needed credentials whereas the latter determines if the user of the app that tries to access the resource has a certain role (e.g., a *primary* user or a *guest* user). We define a small number of canonical values for each aspect. Any access control check can be normalized to one of these canonical values. For instance, checks by invoking different APIs and checks with syntactic differences may be normalized to the same value and hence considered equivalent. Through program analysis, disjunctions and conjunctions of security checks lead to the corresponding operations on canonical values so that program semantics can be faithfully modeled. At the end, our analysis produces a very concise and precise canonical security condition for each access to a sensitive resource. The conditions can be easily compared within a (framework) image or across different images.

We compared normalized access control for common resources *in/across* 12 images, including customized ones (by Samsung, Sony, HTC and LG) and running Android versions 5.0.1 to 7.0. Our analysis led to the discovery of a minimum of 73 unique true positive access control inconsistencies. To prove that our detected inconsistencies are security-critical, we picked 27 instances from the images that we had the physical device and carried out planned attacks targeting to exploit the vulnerabilities. Our results are alarming; we were able to exploit them through high impact attacks, allowing us to write a key logger and even to inject new touches in some LG devices, sending premium SMS messages and bypassing user restrictions on Samsung S6 and S7 Edge, injecting hard key events on Sony Xperia XA, disallowing SD Card mounting and wifi enabling on HTC devices, etc. We have filed security reports to the corresponding vendors. So far, LG, Samsung and Sony have all reproduced more than 20 of our reported vulnerabilities. Particularly, LG has classified two as *Critical* security level, while Samsung has classified three as *Medium*. Due to the lack of the corresponding physical devices, we have not confirmed the remaining cases with real attacks.

Our technique not only enables comparing access control across different framework images that cannot be achieved by existing works, due to their lack of support of detecting equivalent protection with diverse implementation, but also substantially improves the results of comparing access control within an image. We compared with Kratos, the state-of-the-art inconsistency detection framework in its original setting. Our technique has substantially improved the results of Kratos. Specifically, since AceDroid can model much more access control features and peculiarities, it detects on average 28 actual inconsistencies per image, whereas the simulated Kratos detects on average 16. Besides, our access control normalization has helped us suppress a substantial number of false positives. Due to the diverse security checks introduced by vendor customization, if we simply extend Kratos's approach to handle cross-image analysis, the detection will lead to a tremendous number of false positives (on average, 229 instances per image). Thanks to our proposed *normalization*, we are able to reduce this number to 13 instances per image (on average).

Contributions. The scientific contributions of the paper are outlined below:

- We provide a systematic categorization of access controls employed by the Android framework. We propose a path-sensitive modeling and normalization technique for access control checks.
- We develop a new analysis framework for inconsistency detection. We devise several approaches to improving precision in comparing vendor customized frameworks to reveal access control discrepancies.
- Our analysis uncovered high impact exploitable inconsistencies.

II. Motivation

Different from access control in regular kernels (e.g, Linux Kernel), Android framework access control features diversity, namely, there are many different ways to achieve the same level of protection. Some of them are even implicit, implemented by comparisons with local variables and flag variables, which are not much distinguishable from other conditional statements that have nothing to do with access control. Framework developers, especially vendor customization developers, do not have any gold standard to implement appropriate access control. Instead, they tend to compose their own version based on their personal preferences and understanding. As a result, access control implementation tends to be ad-hoc and error-prone. As mentioned earlier, comparing multiple access control instances of a same resource is the most important method to identify framework access control vulnerabilities. However, the diversity in implementation renders such comparison largely ineffective. Next, we use an example to illustrate such diversity, explain how the-state-of-art fails to handle it and how our technique works.

Diversity Caused by Different Implementations and Path-Sensitivity. Consider the two simplified code snippets (Figure 1) extracted from Samsung S6 Edge (6.0.1), both allowing to install a package (leading to the same sink). As shown in the code, the two APIs enforce different access control based on information related to the calling app / user. While the API `installPackageAsUser()` enforces multiple checks along the path to the sink, `installPackageForMDM()` enforces a subset of the same checks. A simplified flow of the multiple paths is depicted in Figure 2. As illustrated, to reach the sink from the API `installPackageAsUser()`, the calling app first needs to hold the system permission `INSTALL_PACKAGE`. Second, it needs to satisfy one of the checks aiming to make sure that the caller is privileged enough to install apps for other users. Specifically, the calling app needs to belong to either the `SYSTEM_UID` 1000 or `ROOT_UID` 0, or hold either one of the two system permissions `INTERACT_ACROSS_USERS_FULL` or `INTERACT_ACROSS_USERS`. Finally, the API enforces a user restriction `DISALLOW_INSTALL_APPS` to verify whether the calling user is restricted from installing apps.

As illustrated on the same figure, `installPackageForMDM()` enforces a more concise access control; a `SYSTEM_UID` check and a user restriction `DISALLOW_INSTALL_APPS` along the path to the same sink.

Listing 1. Simplified Code Snippet for installPackageAsUser ()

```
public void installPackageAsUser(..., int userId){
  enforceCallingOrSelfPermission(INSTALL_PACKAGE);
  uid = Binder.getCallingUid();
  ...
  if (uid != 1000 && uid != 0){
    try{enforceCallingOrSelfPermission(
       INTERACT_ACROSS_USERS_FULL);
    }catch(SecurityException se){
     enforceCallingOrSelfPermission(INTERACT_ACROSS_USERS);
    }
  }
  ....
  if(UserManager.getUserRestrictions(userId).getBoolean(
    DISALLOW_INSTALL_APPS, false)) return;
  Message msg = mHandler.obtainMessage(INIT_COPY);
       msg.obj = new InstallParams(...);
       mHandler.sendMessage(msg);
}                                                    sink
```

Listing 2. Simplified Code snippet for installPackageForMDM()

```
public void installPackageForMDM(..., int userId){
  if (Binder.getCallingUid() != 1000) throw new
     SecurityException("Unauthorized acccess only system is
     allowed");
  if(UserManager.getUserRestrictions(userId).getBoolean(
    DISALLOW_INSTALL_APPS, false)) return;
  ...
  Message msg = mHandler.obtainMessage(INIT_COPY);
       msg.obj = new InstallParams(...);
       mHandler.sendMessage(msg);
}                                                    sink
```

Fig. 1. Two APIs allowing to install a package on Samsung S6 Edge (6.0.1)

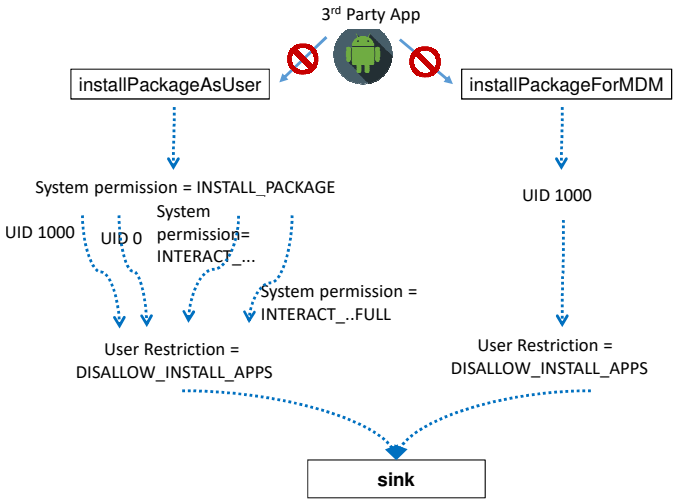

Fig. 2. Two paths allowing to install a package on Samsung S6 Edge

While the checks are syntactically different, their imposed protection is actually semantically equivalent from the perspective of a third-party app. A third-party app cannot obtain any of the UIDs enforced by the two APIs, nor can it hold the enforced system level permissions. Thus, this specific syntactic inconsistency is actually not exploitable by a third party app.

Kratos's Solution. Next, let us consider how the state-of-art tool Kratos [33] would handle this problem. Kratos does not reason about relations among access control checks (e.g., disjunction and conjunctions). It unions all the security checks from the entry point to the sink and considers the resulted set the security condition of the sink. It then compares these conditions across multiple instances of the same resource. Furthermore, Kratos only models a number of explicit security checks such as checks through permission APIs, UID and package name comparison. This is largely fine for in-image analysis on earlier AOSP images (less complex) for which Kratos was designed, because the security checks in those images are relatively simple and uniform. However, with newer features integrated into AOSP and more sophisticated functionalities added by customization parties, the framework includes very diverse security checks, such as checks on users (i.e. owner or guest, current or inactive), processes and package properties, app status (i.e., foreground or background), etc. Moreover, conjunctions and disjunctions are commonly used as well. The design of Kratos can hardly meet these challenges.

For the above example, Kratos determines the security condition for `installPackageAsUser` is {`permission =INSTALL_PACKAGE,UID=1000,UID=0,permission = INTERACT_ACROSS_USERS_FULL, permission= INTERACT_ACROSS_USERS`}. Observe that it unions most checks in the function body without modeling their correlations. Furthermore, it misses the `UserRestriction = DISALLOW_INSTALL_APPS` check because it does not model the user aspect. It will similarly determine the security condition for `installPackageForMDM` is {`UID=1000`}. Kratos will thus recognize these two APIs as inconsistent based on the syntactic differences of the two. We argue this is not an appropriate way to conduct inconsistency detection as the security checks imply a semantic equivalence from the perspective of a third party app, and thus cannot be exploited.

As we will show in our evaluation, the inability to model and normalize the semantics of security checks has led to a large number of false positives and false negatives for Kratos. Note that false negatives occur as some access control related comparisons are not modeled.

A plausible solution is to make the analysis path-sensitive such that the conjunctive and disjunctive relations can be modeled. For instance, we could derive the conditions for the individual program paths for `installPackageAsUser` as `permission= INSTALL_PACKAGE` $\wedge$ `UID=1000` $\wedge$ `UserRestriction=DISALLOW_INSTALL_APPS` (for the left-most path in Figure 2), and so on. However, comparing individual program paths across versions leads to combinatorial explosion and hence hardly scales. Furthermore, without normalization, there would be many paths that have different checks even though they enforce the same level of protection.

Our Solution. We propose a path-sensitive normalization technique that extracts and normalizes security checks to canonical values, which may be conjoined and/or disjoined following the program semantics. As such, we can derive a canonical condition value for each sensitive resource access. Comparison across accesses (that may be even from substantially different framework customizations) becomes both concise and precise after the canonicalization. Specifically, a canonical security condition consists of two orthogonal aspects, the *app aspect* and the *user aspect*. The former denotes the security enforcement for the calling app and the latter denotes the enforcement for the user (e.g., primary user or guest). Each aspect has a small number of canonical values, which is much smaller than the possible syntactic forms of the security checks. Each check is normalized to one of these canonical values. For instance, `permission=INSTALL_PACKAGE`, `UID=1000`,

`UID=0` and `permission=INTERACT..._FULL` are all normalized to a canonical value $System$ to denote they all imply system privilege. In contrast, `permission=INTERNET` or `permission= BLUETOOTH` are normalized to a canonical value $normal$.

These canonical values can be conjoined and disjoined. For instance, the four program paths in `installPackageAsUser` in Figure 2 denote disjunctive relation. Our analysis can strictly follow the program semantics and perform the corresponding operation on the canonical values. The disjunction of two canonical values yields the weaker value. Intuitively, the protection enforced by multiple paths is equivalent to the weakest protection enforced by any of them. For example, a disjunction of *System* and *Normal* yields *Normal*. In contract, the protection enforced by conjoined conditions is equivalent to the strongest protection enforced by any them; e.g., a conjunction of *System* and *Normal* yields to *System*.

At the end, we have the same overall canonical condition for the two APIs `installPackageAsUser` and `installPackageForMDM` as $App := [System] \wedge User := [Restriction = DISALLOW\_INSTALL\_APPS]$. Note that it precisely captures the meaning of the protections. Recall that Kratos would report it as inconsistency due to the lack of normalization.

We develop a program analysis to automatically extract security checks and perform normalization. The analysis is sophisticated such that it can also handle implicit checks who themselves do not seem to be security related, e.g., `if (x==10)`, but serve the functionalities of security checks. In our experiment, the analysis is applied to 12 complex frameworks and customizations that correspond to average 977030 LOC per-image.

Diversity Caused by Non-Standard Checks. Besides the traditional permission and UID checks, the framework developers (especially for custom frameworks) might rely on non-standard security features to enforce access control. For example, the package flags, signature and shared user ID might all be used to infer the privilege of the caller. A failure to account for such non-standard checks would lead to missing important inconsistencies.

Figure 3 demonstrates an example of an exploitable inconsistency that would go undetected if non-standards checks are not considered during analysis (e.g., as in Kratos). The figure depicts simplified code snippets of the API `DirEncryptService.getSDCardEncryptionPref()` across two Samsung device models. As shown, while Samsung Tab S 8.4 allows retrieving stored encryption preferences without any security checks, Samsung S6 Edge permits this operation only to platform signed apps. The API first retrieves the signatures of the calling package and that of the android platform based on the calling app's UID (line 6) and system process PID (line 7), respectively. It then compares the two signatures (line 8) and allows accessing the stored encryption preferences only if they are identical.

To address this, we have identified numerous features that are related to access control by inspecting central services responsible for maintaining information about installed apps, users and running processes. More details are discussed in (Section III).

Listing 3. Simplified Code from Samsung Tab S 8.4

```
1  public SDCardEncryptionPolicies getSDCardEncryptionPrefs(){
2    final long i = Binder.clearCallingIdentity();
3    ..
4    restorePrefs = this.mDep.restorePrefs();
5    Binder.restoreCallingIdentity(i);
6    return restorePrefs;                                  sink
```

Listing 4. Simplified Code from Samsung S6 Edge

```
1  public SDCardEncryptionPolicies getSDCardEncryptionPrefs(){
2    final long i = Binder.clearCallingIdentity();
3    ..
4    restorePrefs = this.mDep.restorePrefs();
5    Binder.restoreCallingIdentity(i);
6    Signature[] appSig =
       getPackageSettings(Binder.getCallingUid()).signatures;
7    Signature[] platformSig =
       getPackageSettings(Process.myUid()).signatures;
8    if (compareSignatures(platformSig, appSig) == 0)
9     return restorePrefs;
10   else return null;                                    sink
```

Fig. 3. DirEncryptService.getSDCardEncryptionPrefs() across two Samsung models

III. Android Access Control Modeling

In this section, we introduce our systematic modeling and normalization of Android access control.

A. Uncovering Access Control Criteria.

Android is a layered operating system with its app and framework layer built with Java sitting on top of a set of C libraries and the Linux Kernel. At the app layer are third party and system apps, preloaded by the vendors and other customization parties. The Android framework provides many high-level system services (e.g. `LocationManagerService` and `ConnectivityService`) implementing essential functionalities, and communicating with the Linux Kernel. Android system services execute in system processes and expose their privileged functionalities via a set of well-defined interfaces that are accessible by other apps and services through the Binder IPC.

Upon receiving a Binder transaction (i.e., invocation of an exposed API), a system service determines the identity of the caller to allow / deny access to the underlying resource. Prominently, the Binder class defines the following three APIs allowing to retrieve the caller identity:

```
1  public class Binder implements IBinder {
2    public static final native int getCallingPid();
3    public static final native int getCallingUid();
4    public static final UserHandle getCallingUserHandle() {
5      return
         UserHandle.of(UserHandle.getUserId(getCallingUid()));}
```

The caller's PID and Linux UID can be used to uniquely identify the app process that sent the current transaction while the caller's `UserId` can be used to identify the user who initiated the transaction. It should be noted that the `UserId` is distinct from the calling app's UID in that it reflects the actual user of the device, which has multiple apps under it, each with their own UID.

TABLE I. ANDROID ACCESS CONTROL MODELING

Category	Perspective	Security Feature of Caller	Example of Usages
App-Specific	Privilege Check	UID	Comparison with a Privileged UID
		PID	Comparison with a Privileged PID
		Permission	Calling app has a permission
		Package Name	Comparison with constant String
		Process Name	Comparison with constant String
		Shared User ID	Comparison with constant String
		Signature	Comparison with a signature of another app
		Flags	Comparison with SYSTEM_FLAG
	Ownership Check	Calling App UID	Comparison with UID of input parameter
		Package Name	Comparison with input parameter
	Status Check	PID / Process Flags	Comparison with running PIDs and has a IMPORTANCE_FOREGROUND
User-Specific	Privilege Check	User Id	Comparison with a privileged User Id (e.g. 0)
		Flags (User Type)	Comparison with FLAG_PRIMARY , FLAG_GUEST FLAG_RESTRICTED, FLAG_ADMIN
	Ownership Check	User Id	Comparison with an input parameter
	Status Check	User Id	Comparison with current user
	Restrictions	Restriction Name	Calling user has a restriction

While the essence of Android access control is to employ these 3 identifiers to enforce *app-specific* and/or *user-specific* permissions, *the implementation of the access control checks is much more diverse than simple equivalence checks of these identifiers*. This is because there are a large number of features that are associated with these identifiers that can be used as their delegators in access control checks. Therefore, we have to identify, model and normalize all these access control checks in various forms in order to perform comparison.

All the central Android services responsible for managing app and user data and pertinent runtime information may contain features that can be used in access control. Therefore, we have to consider and model all of them, including 3 services: the `PackageManagerService`, `UserManagerService` and `ActivityManagerService` and 7 classes: `PackageInfo`, `PackageSetting`, `SharedUserSetting`, `ApplicationInfo`, `UserHandle`, `UserInfo`, and `RunningAppProcessInfo`. The criterion to determine if a feature is related to access control is whether it has association with one of the aforementioned three identifiers (e.g., package name can be retrieved with app UID). Note that these central Android services rarely change across framework versions and customizations, and hence serve as an ideal common basis for normalization. We observe some recent customizations occasionally use vendor added services in access control (e.g., LG's `IMdm` and Samsung's `EnterpriseDeviceManager` which enforce additional restrictions based on a custom user policy). They can be nonetheless modeled and normalized. However due to the small number of such instances, we leave them to our future work. We propose a classification of the access control related features in these services and classes, as shown in Table I. The classification is dictated by our threat model. In the following, we explain our threat model and then the classification/normalization.

Threat Model. We formally state our assumptions about the calling entity and the corresponding threat model, as follows: ① *a malicious third-party app aiming to exploit an inconsistency to perform a privileged operation with no/weaker requirements;* ② *a malicious user aiming to affect other users and/or bypass imposed restrictions.* In other words, access control checks using different features are considered equivalent if they allow/disallow the same set of behaviors from third-party apps and users.

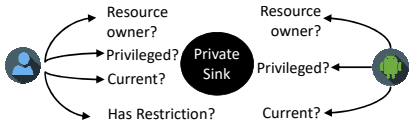

Fig. 4. Android Access Control Perspectives

Categorization of Android Access Control. At the highest level (the first column in Table I), we propose to classify all Android access control checks as two categories: *app-specific* and *user-specific* checks. For the app category, there are three unique perspectives (i.e., the first three entries in the second column): *privilege checks*, *ownership checks*, and *status checks*. The meaning of these checks can be intuitively explained in the right half of Figure 4. Specifically, privilege checks are to determine if the calling app owns certain privileges; ownership checks decide if the calling app is the owner of the resource; status checks validate if the calling app is the foreground app (as certain accesses are denied for background apps). As shown in the third column of Table I, checks in each perspective can be in various forms. For instance, (app) privilege checks can be carried out by comparing UID/PID, explicitly checking permissions, or even checking specific flag fields in relevant data structures. Many of these checks are semantically equivalent despite of their various syntactic forms.

Similarly, the user checks include *privilege checks* (i.e., privileges associated with specific users), *ownership checks*, *status checks*, and *restriction checks* that determine if a user is restricted from certain operations. The intuitive explanations are in the left half of Figure 4. Their details are explained later in this section.

Based on this understanding, we propose to model Android access control as follows:

$$\begin{aligned} GrantAccess &:= [App, User] \\ App &:= [Privilege, Ownership, Status] \\ User &:= [UsrPrivilege, Ownership, Status, Restriction] \end{aligned}$$

Specifically, each access control check is modeled as a pair consisting of the app and user aspects, each aspect itself being a vector of multiple perspectives. Later we will show that each perspective has a small number of canonical values. Our tool aims to transform all access control checks to canonical vectors.

Normalization. Although many of these checks are in different

forms, they have the equivalent semantics in terms of the protection they entail. Framework developers often leverage their domain knowledge of such implicit equivalence during customization and version upgrade. As a result, access control checks seem very diverse and sometimes even ad hoc across customizations and versions. The key challenge to consistency checking is hence to normalize these access control checks. *Our definition of equivalence is regarding the protection enabled by the checks, that is, the kind of malicious behaviors precluded by the checks.* For instance, enforcing that the calling package signature matches that of `"android"` and enforcing the calling process PID = `Process.myPid()` are syntactically inconsistent from the perspective of a preloaded app. However, neither the signature nor the PID can be acquired by a third party app and hence both checks disable exploit from a third party app and thus are semantically equivalent.

B. App-Specific Access Control

1) Privilege Perspective.: This perspective aims to verify whether the calling app is allowed to perform an operation because it holds a capability or is a privileged process / app (e.g., app is granted the permission `MASTER_CLEAR` or has a `SYSTEM_UID` thus can perform a factory reset). As depicted in Table I, this perspective can be enforced based on the caller's permissions, UID, PID, and others.

We formulate the syntax of privilege checks as follows.

$$\begin{aligned} AppPrivCheck\ P := {} & checkPermission(C) \mid \\ & UID = 0 \mid ... \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \end{aligned}$$

A privilege check can be a primitive check (e.g., $UID = 0$) or a compound check which is multiple primitive checks connected by disjunction or conjunction. While the syntax only lists two primitive checks, there are in fact many of such checks, some of which will be discussed later in the section. Despite of the different syntactic forms, the semantics of these checks can be normalized to the following canonical *Privilege* domain.

$$Privilege := \{System,\ Dangerous,\ Normal, \bot\}$$

The values in the domain are totally ordered. $System$ is the highest privilege level (acquired only by system apps and processes); $Dangerous$ is lower, indicating user confirmation is needed (for protecting private data or for operations affecting the user's data); $Normal$ indicates privileges that may be acquired by third party apps without user confirmation; $\bot$ implies no privilege.

We define a function $evalAppPriv$ to evaluate a privilege check to a canonical value. Its signature is the following.

$$evalAppPriv : AppPrivCheck \mapsto Privilege$$

While we will define the semantics of $evalAppPriv$ for individual primitive checks later, the semantics for compound checks are defined as follows.

$$evalAppPriv(P_1 \vee P_2) = Min(evalAppPriv(P_1), evalAppPriv(P_2))$$
$$evalAppPriv(P_1 \wedge P_2) = Max(evalAppPriv(P_1), evalAppPriv(P_2))$$

Intuitively, if a public API entry enforces two privilege requirements, we normalize it to the maximum privilege level of the two. Conversely, if the access control requires either one, we can safely map it to the minimum privilege of the two checks, as a calling app only needs one of them to access the resource.

Next, we describe a few primitive app privilege checks and their (normalized) semantics. We cannot include others due to the space limit.

Permission Checks. Android can invoke a permission validation call $checkPermission(C)$ to check whether a given app process has been granted a specific permission C. According to the Android specification, the protection level of a permission can be `Normal`, `Dangerous`, `Signature` or `SystemOrSignature`, depending on the resources protected. The normalized semantics of the checks is hence the following.

$$evalAppPriv(checkPermission(C)) =$$
$$\begin{cases} System, & PL(C) = SystemOrSignature \vee System \\ Dangerous, & PL(C) = Dangerous \\ Normal, & PL(C) = Normal \vee Undefined \end{cases}$$

The above normalization implies that two permissions are semantically equivalent if they hold the same protection level, denoted by the function $PL()$, regardless of their name. For example, $checkPermission$(`"android.permission.CONTROL_KEYGUARD"`) and $checkPermission$(`"android.permission.MANAGE_FINGERPRINT"`) are normalized to the same $System$ value. The protection levels *SystemOrSignature* and *System* are equivalent, since neither can be acquired by a third party app. Please note that the *Undefined* level refers to the cases where a permission is not defined in an image, or what has been referred to as a Permission Hare in [7]. Obviously, in such a case, the caller can define the permission with the lowest protection to pass the check.

UID Checks. System services mediate access to certain privileged resources based on the caller's UID. To enforce this check, the system services compare the caller's UID with defined privileged UIDs (e.g., `ROOT_UID` and `SYSTEM_UID`). To evaluate the privilege requirement for these checks, we map all UID checks smaller than `FIRST_APPLICATION_UID` (the start of the range of UIDs reserved to apps) to the $System$ level:

$$\begin{aligned} evalAppPriv(&UID = c, \\ &with\ c < FIRST\_APPLICATION\_UID) = System \end{aligned}$$

PID checks are similarly modeled.

Package Properties. To allow finer grained privilege requirement checks, Android may rely on other properties of the calling app, which have strong association with its UID / PID. As shown in Table I, we have identified the features that can be used to infer the privilege of a calling app. The system service might retrieve any of these attributes from the calling app's `PackageInfo`, `ApplicationInfo` or its `RunningAppProcessInfo` and compare them with privileged ones (i.e., those assigned to system apps / processes).

Third party apps cannot obtain or share the privileged features because they should be signed with the same certificate as the defining app, or because Android enforces their uniqueness.

Let $Name$ denote a package, process or sharedUserId name of the caller. We normalize the privilege levels of $Name$ as follows:

$$evalAppPriv(Name = c, with\ c\ a\ sysapp) = System$$
$$evalAppPriv(Name = c,\ with\ c \in OtherName) = Normal$$

where $OtherName$ denotes names that can be claimed by any app.

Similarly, flags and signatures can serve the same purpose of checking privileges. We normalize them in a similar fashion.

2) Ownership Perspective.: This perspective aims to verify that the caller app is allowed to access a resource because it owns the resource or to perform an operation pertaining to itself (e.g. an app can only delete its own cache).

We define the function $evalAppOwner$ to evaluate an ownership check to a boolean value with *true* indicating that the app is the owner of the resource.

$$evalAppOwner : AppOwnerCheck \mapsto \{True, False, \bot\}$$

Back to our categorization in Table I, the ownership perspective can be generally enforced by comparing a security feature of the calling app (e.g., UID and package name) with a feature derived from the parameter to the public entry point. Consider the following example extracted from `ActivityManagerService`.

```
1 public boolean clearApplicationUserData(String packageName,
    IPackageDataObserver observer, int userId) {
2   uid = Binder.getCallingUid(); pm =
      AppGlobals.getPackageManager();
3   pkgUid = pm.getPackageUid(packageName, userId);
4   if (uid == pkgUid ||
      checkComponentPermission(CLEAR_APP_USER_DATA))
5     pm.clearApplicationUserData(packageName, observer,
        userId); ...
```

As shown, to clear an app's data (specified by its package name), the system service verifies that the calling process either owns this app or holds a system permission. To enforce the ownership check, the service first obtains the target app's UID (line 3) and then compares it against that of the calling app (line 4). The same check can be enforced through comparing the calling UID's package name with the supplied name. Let UID and $Name$ be an identifier of the calling app (e.g., `uid` in our example) and Arg an identifier (e.g., `pkgUid`) derived from some input parameter (e.g., `packageName`). We model an ownership check as follows.

$$evalAppOwner(UID = Arg) = True$$
$$evalAppOwner(Name = Arg) = True$$

3) Status Perspective.: This perspective aims to ensure that the calling app is running in the foreground UI that the user is currently interacting with. It enables the system service to accept operations that are directly initiated by the user, rather than by a background process.

We define a function $evalAppStatus$ to evaluate an app status check to a boolean value with *true* indicating that the app is the foreground app.

$$evalAppStatus : AppStatusCheck \mapsto \{True, False, \bot\}$$

As illustrated in Table I, status checks can be enforced by verifying attributes associated with the calling process (e.g.,`importance` flag) . The following example demonstrates this:

```
1 private boolean isForegroundActivity() {
2   uid = Binder.getCallingUid(); pid =
      Binder.getCallingPid();
3   List procs = ActivityManagerNative.getDefault()
      .getRunningAppProcesses();
4   for (int i = 0; i < procs.size(); i++) {
5     RunningAppProcessInfo proc = procs.get(i);
6     if (proc.pid==pid&&proc.uid==uid
        &&proc.importance==IMPORTANCE_FOREGROUND)
7       return true;
```

As shown, the system service first obtains the `RunningAppProcessInfo` corresponding to the calling app, and then checks whether its `importance` is equivalent to `IMPORTANCE_FOREGROUND` (line 6). The modeling details are elided.

C. User-Specific Access Control

The integration of multi-user and restricted-profile support was a major enhancement introduced in Android 4.2 and 4.3, respectively. Although it was initially advised to share devices with people that the owner trusts, the multi-user support gradually evolved towards less trustworthy environments (e.g., corporate). It has also evolved from being available only on tablets in Kitkat to phone models as well in Lollipop.

Clearly, Android has incrementally progressed towards a multi-user environment rather than having it designed-in from the beginning, a path which can potentially introduce vulnerabilities if user-specific checks are not enforced consistently in exposed framework APIs.

1) Privilege Perspective: Each Android user has distinct privileges. For example, a *primary* user has special privileges, e.g., it is always running in the background and has exclusive access to some security critical settings. Other users are less privileged. To normalize the privilege enforcement, we re-emphasize our threat model under the multi-user scenario: a new user added to the device, with a malicious intention, i.e., aiming to affect other users and/or bypass imposed restrictions. We hence define the following normalized user privilege domain.

$$UserPrivilege := \{Primary,\ Secondary,\ Guest, \bot\}$$

The elements in the domain are ordered: $Primiary > Secondary > Guest > \bot$. Where $Primary$ is the most privileged user (i.e., owning / administrating the device); it cannot be removed by other users and is always running in the background even when other users are running. $Secondary$ is less privileged (e.g., cannot access sms and telephony functions by default) while $Guest$ is the least privileged.

We further define a function $evalUsrPriv$ that evaluates a user privilege check to one of the privilege level.

$$evalUsrPriv : UsrPrivCheck \mapsto UserPrivilege$$

As shown in Table I, Android relies on the UserID and associated flags to enforce user privilege separation. For instance, the following code (from the `LocationManagerService`) shows how proximity alerts are only available to the owner of the device.

```
1 public void requestGeofence(LocationRequest request,
    Geofence geofence, ..) {
2  int uid = Binder.getCallingUid();
3  if (UserHandle.getUserId(uid) != UserHandle.USER_OWNER) {
4   Log.w(T, "proximity alerts are available only to primary
      user");
5   return;}
```

Some normalization rules of user privilege checks are as follows. Others are omitted due to space limit.

$$evalUsrPriv(UserID = USER\_OWNER) = Primary$$
$$evalUsrPriv(UserFlag = FLAG\_ADMIN) = Primary$$
$$evalUsrPriv(UserFlag = FLAG\_GUEST) = Guest$$

2) User Ownership and Status Perspectives.: Android leverages the user ID to ensure that a user cannot manipulate other user's settings or perform sensitive operations on behalf of others. Similar to checking the owner of an app (Section III-B2), enforcing the user ownership is generally performed by comparing a calling user ID with a user ID supplied as a parameter to the public API.

Besides, through user-switches, a user can still be running in the background when another user logs in. Thus, Android further checks the status of a user (i.e., *active* or *inactive*) to prevent inactive user from spying on active users or tamper with settings affecting them (e.g., an inactive user starts a camera recording session to spy on the active user). The check is usually done by comparing the calling user ID with that of the currently active user.

Since the modeling of these perspectives shares a lot of similarity with the corresponding perspectives for app checks. We omit the details.

3) Restrictions Perspective.: This perspective adds another layer of security in the multi-user environment. Each user has an associated user restriction list containing non-permissible operations. This list is created when the user account was created and can only be updated by the system process (or the device admin). For example, by default, secondary users cannot send, receive sms messages, or issue outgoing phone calls. Guest users have additional restrictions (cannot configure wifi or install apps from unknown sources).

Android enforces the user restrictions upon performing particular operations. For example, the `ConnectivityManagerService` ensures that users with `DISALLOW_NET WORK_RESET` restrictions cannot perform a network reset as shown in the following:

```
1 public void factoryReset() {
2  enforceConnectivityInternalPermission();
3  if (mUserManager.hasUserRestriction
     (UserManager.DISALLOW_NETWORK_RESET))
4   return; ...
```

Different from other domains such as $UsrPrivilege$ that can be normalized to a small number of abstract levels, restrictions are more fine-grained so that different restrictions have unique meanings. As such, the $Restriction$ domain is the universal set of all the defined restrictions. The semantics function $evalUsrRestriction$ evaluates a restriction check to the corresponding restriction.

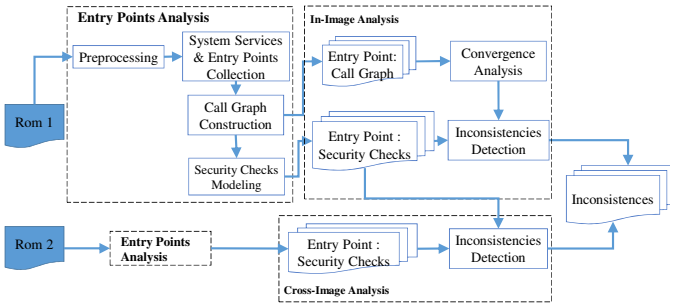

Fig. 5. Approach

IV. System Design

Our tool, built on top of WALA [23], follows the high-level work flow depicted in Figure 5. First, for each input Android ROM, we extract the framework class files and collect candidate system services. For each system service, we locate all publicly accessible entry points and build a call graph starting from each entry. We then model and normalize all security related checks appearing inside the call graph of a particular entry point.

A. Preprocessing

Given an Android image, AceDroid extracts its framework class files. As different versions or vendors might pack the code in different formats, we employ several existing tools to handle each format gracefully [2], [5], [18], [4].

System Services Collection. Identifying exposed system services in a decompiled Android ROM is not straightforward. System services registration is scattered throughout the framework code. The registration point might further vary across AOSP and custom Android images. To identify the services, we follow the key observation that the registration APIs are quite stable across Android customizations and upgrades. Particularly, the APIs `addService` and `publishBinderService` allow publishing a framework system service through registering in the service manager. AceDroid first pinpoints the invocation of these two APIs in the framework, then resolves the registered service class type.

Entry Point Identification. To collect public entry points, we first identify the exposed interface class of each service and then retrieve its declared public APIs. We do also mark the `onReceive` APIs in the identified services as public entry points. Please note that unlike Kratos which considers only unprotected receivers as public entry points, AceDroid extracts both protected and non-protected receivers. This design decision aims to detect the inconsistent cases where two receivers lead to the same sink point such that one is protected while the other is not. A protected broadcast action is evaluated to the highest privilege in our modeling.

Algorithm 1 Extracting Security Features.

Require: Inter-procedural Control Flow Graph *icfg*
Ensure: Security Features *sFeatures* of the corresponding public entry

```
1: function NORMALIZE(icfg)
2:     entryBB = icfg.entry
3:     return DFSTRAVERSEICFG(icfg, entryBB, null)
4: function DFSTRAVERSEICFG(icfg, bb, pathCheck)
5:     if bb == icfg.exit then
6:         sFeatures = sFeatures ∨ pathCheck
7:         return
8:     for all succ ∈ icfg.feasiblecontrolDepSuccessors(bb) do
9:         pathCheck' = COPYOF(pathCheck)
10:        instr = LASTINSTRUCTIONOF(bb)
11:        if isConditional(instr) and isTrueBranch(bb, succ) then
12:            check = CLASSIFYSECURITYFEATURE(instr)
13:            if check ≠ null then
14:                pathCheck' = pathCheck' ∧ check
15:        DFSTRAVERSEICFG(icfg, succ, pathCheck')
16: function CLASSIFYSECURITYFEATURE(cond)
17:    X = DEPTRACK(cond.fstOperand)
18:    Y = DEPTRACK(cond.sndOperand)
19:    if isAccessControl(X) then
20:        if isAppCheck(X) then
21:            return [evalAppPriv(cond), evalAppOwner(cond), evalAppStatus(cond)]
22:        else if isUserCheck(X) then
23:            return [evalUsrPriv(cond), evalUsrOwner(cond), evalUsrStatus(cond), evalUsrRestriction(cond)]
24:    else if isAccessControl(Y) then
25:        /*symmetric and elided*/
26:    else
27:        return null
```

B. Modeling Explicit Security Checks.

Explicit permission enforcements are those conducted by directly invoking security relevant APIs, such as `checkPermission()`. To model these checks, AceDroid traces back from the checks to the permission name strings passed as arguments. To get the security checks associated with the entry points of broadcast receivers (i.e., `onReceive`), AceDroid extracts the permissions and intent filter actions supplied to the registration APIs (e.g., `registerReceiver`) AceDroid then verifies whether the action filters are protected broadcasts within the analyzed images.

As discussed in Section III, we normalize android permissions based on their protection levels rather than their names. Hence for the identified permissions, we extract the corresponding protection levels from the framework configuration files (`framework-res/ AndroidManifest.xml`), custom files (e.g., `lge-res/AndroidMani-fest.xml` in LG) and from preloaded apps manifests.

C. Modeling Implicit Security Checks.

Implicit security checks are those that serve the purpose of access control but do not explicitly invoke security related APIs. It is more challenging to identify, extract and normalize these checks. The procedure is described in Algorithm 1. For each entry point, the algorithm is fed with an inter-procedural control flow graph (*icfg*) of the entry point. It normalizes the security related conditional statements to canonical values. Starting from the entry basic block of the *icfg*, depth-first search along the control dependence edges is employed to traverse all paths. In each path, when a conditional statement is met, we normalize it if it is security related (lines 11-12). Method `classifySecurityFeature()` is responsible for determining if a conditional statement is security related and for normalization. Within the method (line 17 and line 19), it checks if the first operand of the predicate is related to access control by tracking its dependencies backward, including both data dependencies and some special control dependencies as explained later, in order to decide if it originates from some predefined security feature(s) (e.g., PID, package signature, and user ID as defined in Section III). Similarly, we back track the second operand to a concrete value like 0 and 1000, or a parameter of the entry point (line 18). From lines 20 to 23, the algorithm further checks the type of security check. Depending on the type (app-specific or user-specific), different canonicalization methods are invoked. These methods are defined in Section III. Lines 24-25 handle the symmetric case, e.g., a constant is used as the first operand.

A canonicalized condition (i.e., vectors of canonical values) returned by `classifySecurityFeature()` is conjoined with the current canonical condition derived from all the preceding security checks along the path (line 14). We use conjunction because all access control checks have to be satisfied along this path. The canonical security condition for the entire entry point is the disjunction of those for individual paths (line 6). The semantics of conjunction and disjunction (of canonical vector values) are defined in Section III. Note that instructions, including predicates, are preprocessed to SSA form, in which a predicate contains only a simple comparison. Compound predicates (e.g., a conjunction of two primitive comparisons) are broken down to multiple predicates.

For example, in the following code snippet

```
1 int userId = UserHandle.getCallingUserId();
2 int uid = Binder.getCallingUid();
3 checkPermission(BLUETOOTH_ADMIN_PERM); // Normal Protection
4 if (uid == 1000 ||
    userId == ActivityManager.getCurrentUser()){...}
```

we get the canonical security condition for the first path (the true branch of the UID comparison) as follows.

$$\begin{aligned} App = & \ [evalAppPriv(checkPermission(B...) \wedge UID = 1000),\ \bot,\ \bot] \\ = & \ [Max(evalAppPriv(checkPermission(B...)), \\ & \ evalAppPriv(UID = 1000)), \bot, \bot] \\ = & \ [Max(Normal, System),\ \bot,\ \bot] \ = \ [System,\ \bot,\ \bot] \\ User = & \ [\bot, \bot, \bot, \bot] \end{aligned}$$

Recall an app-aspect canonical value is a triple containing the privilege, ownership, and status (Section III-B). The latter two do not apply here, and thus the computation is mainly for privilege. Since there are two privilege checks along the first path: `checkPermiss-ion(B...)` and `uid==1000`. The canonical value of the path is computed by applying *evalAppPriv()* to the conjunction of the two. And the condition for the second path is computed as follows.

$$\begin{aligned} App &= [evalAppPriv(checkPermission(BLUE...)),\ \bot,\ \bot] \\ &= [Normal,\ \bot,\ \bot] \\ User &= [\bot,\ \bot,\ evalUsrStatus(UserID = current),\ \bot] \\ &= [\bot,\ \bot,\ True,\ \bot] \end{aligned}$$

The canonical condition for the entry point is the disjunction of the conditions of the two paths, which is the following.

$$App = [Normal, \perp, \perp] \qquad User = [\perp, \perp, True, \perp]$$

Function DEPTRACK() tracks the dependencies of a given operand to its security related origin, if any. The tracking is mainly performed based on data dependence and a special kind of control dependence. Basically, if the origin of the operand is a concrete value, e.g., 0 or 1000, the function returns the value; if it is a parameter of the entry point, the function returns a flag indicating that it is a user specified value; and if it is originated from a certain pre-listed API method call such as `getCallingUID()`, the function returns the corresponding abstraction, e.g., `UID`. In addition to data dependencies, the function also tracks a special kind of control dependence that implies one-to-one mappings between variables and hence have a nature similar to data dependencies. Consider the following example.

```
1 boolean f = false;
2 if (uid == Binder.getCallingUid()) /*security check*/
3    f = true;
4 if (f) {/*access resource*/;}
```

Variable `f` at line 4 does not have any data dependence with security related feature although line 4 is clearly an access control check. Observe that `f=true` must imply `uid==Binder.getCallingUid()`. There is a one-to-one mapping between `f` and `uid`. This is similar to the nature of data dependence. For example, in `y=x+1`, the data dependence between `y` and `x` essentially denotes a one-to-one mapping. Observe that if we change the comparison at line 2 to `uid != Binder.getCallingUid()`. The mapping is no longer one-to-one as there are many possible values of `uid` that lead to `f=false`. Therefore, we track control dependencies caused by equivalence checks, which are the dominant kind of security checks.

V. INCONSISTENCY ANALYSIS

We applied AceDroid to detect security checks inconsistencies in various ROMs. We propose two analyses: *cross-image* and *in-image*.

Through the *cross-image* analysis, we aim to identify access control discrepancies along similar public entry points across two Android images. An intuitive approach to perform this analysis is to compare the security enforcement leading to common sinks in two images. This approach is heavyweight due to the sheer number of sinks and does not seem to be necessary: for common public entry points, we observe that vendors rarely alter the original sinks during customization, rather, they might alter the implementation to adopt it to custom hardware / functionalities. Intuitively, if vendors decide to add new resources, they would usually add new public APIs allowing the invocation of the corresponding sinks.

We rely on this key observation to devise a faster comparison of common public entry points. We compare the security enforcement across common public entries regardless of their invoked sinks.

However, although comparing common APIs (identified by a common method name and descriptor) covers the majority of APIs within an image, it does not allow reasoning about all cases, especially in vendor customized images. To generate more accurate results, we address the following important customization aspects in our cross-image analysis.

Renamed APIs. We observe that the signature of some custom public API (i.e., not appearing in the AOSP codebase) is not stable throughout version updates. For example, The API `reboot()` in Samsung S6 Edge (6.0.1)'s `DevicePolicyManager` is renamed to `rebootMDM()` in S7 Edge (7.0). To address this, AceDroid compares the call graphs of non-common APIs to identify the ones sharing the same implementation.

Exposed/ Non-Exposed APIs. Certain system service APIs are for exclusive system-use and thus are not exposed via the service's AIDL class. Surprisingly, we found out through our inspection of vendor added APIs that sometimes, vendors do expose APIs which are internal in their counter AOSP version. This practice is quite risky because of the following: intuitively, since these APIs are meant for framework use, access control is not needed; however, if vendors decide to expose them, they should remedy the exposure with strong access control checks. A failure to do so might expose important privileged resources. To address these cases, we try to match vendor added public APIs inside a service (by signature comparison) with the private APIs of the counter service. (or those registered in a local service, within the same class). If a match is found, we compare their imposed access control checks, please note that we map an `unexposed` feature to the canonical value with the highest security.

The in-image analysis compares canonical conditions for multiple accesses to a same resource. We followed the taxonomy proposed by [11] to identify the sinks. We further focused on vendor added methods in this analysis as inconsistencies in other methods would be detected by the cross-image analysis. Details are omitted due to space limit.

VI. EVALUATION

To evaluate the effectiveness of access control modeling and normalization, we applied AceDroid to detect inconsistencies in 12 factory images. Our results show that modeling and normalization are critical to cross-image analysis due to their capabilities of handling implementation differences caused by customization; they have also substantially improved the state-of-the-art in-image analysis.

A. Collected Images.

In our research, we collected 12 factory images from online repositories [3], [1] and physical devices. These images are customized by 5 distinct vendors and operate Android versions from 5.0 to 7.0. We selected our images carefully to allow comparison through versions upgrades, different vendors and different models (e.g., Tablet versus Phone).

B. Runtime Overhead.

Table II shows the details of our collected system services and identified entry points. The 2nd column reports the number of collected services while the 3rd column reports the number of detected public APIs and registered receivers. Please note that some of the vendor added services were not correctly

TABLE II. STATISTICS SUMMARY

Image	# of Services	# Exposed Methods & Receivers	In-Image Time (min)	Max Cross-Image Time (min)
Nexus 5.0.2	85	1491	44	17
Nexus 6.0	87	1715	53	21
Nexus 6.0.1	87	1727	54	21
S6 Edge 6.0.1	124	3605	99	41
Tab S 8.4 (6.0.1)	89	2187	73	32
S7 Edge 7.0	119	3138	112	41
LG G3 5.0.2	86	1693	59	27
LG G4 6.0	89	1917	59	27
HTC M8 5.0.2	85	1556	53	26
HTC M8 6.0	87	1882	62	29
Sony Xperia XA 6.0	92	2003	75	24
Sony Xperia XZ 7.0	93	2032	79	24

decompiled because of some limitations in pre-processing the ROMs.

The last two columns of Table II show a summary of the time consumed by AceDroid to conduct the in-image and cross-image analyses. As shown, the in-image analysis takes on average 65.2 minutes. Since this is a one time effort, the time is acceptable.

The cross-image analysis time varies based on the number of common entry points between two given images. Thus, we report the time consumed in the comparison between images sharing the maximum number of common entries. As illustrated, comparing S6 Edge and S7 Edge incurs the longest time, since they share the largest number of common entries.

C. Inconsistencies Landscape.

AceDroid discovered that all analyzed images contain security enforcement inconsistencies. Table III shows the details and reads as follow: each row / column corresponds to a unique image. The intersection of a row and a column shows the inconsistencies discovered through comparing the two images, depicted by the column / row name. The in-image analysis results are presented in the intersection of the same name column / row.

To measure the True Positive (TP) and False Positive (FP) rate, we manually inspected each reported inconsistency. The white cells in Table III depict the discovered TP and the Total # of reported inconsistencies. The total number of unique TP vulnerabilities found in these 12 images is 73. Note that a vulnerability can be reported by multiple inconsistency analyses. Through the normalization process, AceDroid avoids detecting cases where two accesses apply different security checks, yet expressing the same protection from the perspective of a malicious app / user. That is why, the reported inconsistencies in the white cells all represent cases of accesses applying actual different checks.

To demonstrate the usefulness of our normalization process, we report the number of false inconsistencies detected if we extend a simpler approach to handle cross-image analysis without sophisticated analysis. The results are depicted in the shaded cells. Due to the diverse security checks introduced by vendor customization, the detected instances were quite high, e.g., reaching a tremendous number of 523 false alarms when comparing Samsung S7 Edge 7.0 and Nexus 5.0.2 (on average 229 instances). After the normalization, we were able to reduce this number to 12 false inconsistencies (on average 13 instances). This clearly demonstrates the importance of our normalization process.

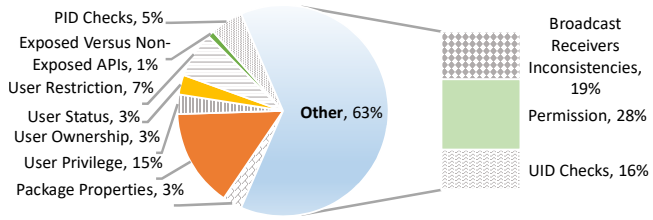

Fig. 6. Inconsistencies Breakdown

In-Image Results. As shown in the table, AceDroid detects a significant number of inconsistencies through the in-image analysis, where Nexus images exhibit the smallest number of inconsistencies. Sony and Samsung introduce the highest inconsistencies. A possible reason is that these vendors perform an extensive customization of the AOSP code bases (as shown in Table II).

Cross-Image Results. The problems are also pervasive across different device manufacturers: For instance, comparing Nexus (6.0.1) with Samsung S6 Edge(6.0.1) leads to 21 actual inconsistencies (e.g., Samsung's `setStreamVolume()` in `AudioService` does not include a user restriction check found in Nexus (6.0.1)). Interestingly, even within different models from the same vendor and OS version, security enforcements are different: 12 true inconsistencies were detected across Samsung's S6 Edge (6.0.1) and Tab S 8.4 (6.0.1). For example, the custom API `setMultipleScreenStateOverride` in `PowerManagerService` enforces a system permission in Tab S while it enforces no checks in the counter S6 Edge. These findings further indicate the decentralized nature of vendor customization.

We also observed that version updates cause a significant number of inconsistencies. For example, the APIs `setOemUnlockEnabled` in `PersistentDataBlockService` and `mount` in `MountService` both add a user-based check (primary user and a restriction check) in the images 6.0. Naturally, this could be attributed to the fact that through version updates, vendors patch previously unprotected accesses.

D. Inconsistencies Breakdown.

We further analyzed our reported inconsistencies detected through both analyses, and classified each case based on the security features described in Section III. We found out that inconsistencies are caused by the absence or alteration of various attributes. The detailed breakdown is depicted in Figure 6. As depicted, 63% of the reported cases are due to permissions, UID, and broadcast receivers inconsistencies and hence may be detected by an approximate solution that only models explicit permission checks. However, we want to point out that such true positives would be substantially overshadowed by the large number of false positives due to the lack of normalization and path-sensitive analysis (i.e., a few hundreds per-image as shown in Table III).

E. False Positives.

As illustrated in the results, not all our reported inconsistencies are actually TPs. Our average FP rate is 31.4%. The in-image analysis particularly introduces more FPs because of specific limitations that cannot be automatically handled. Our

TABLE III. INCONSISTENCIES LANDSCAPE

Image	Nexus 5.0.2	Nexus 6.0	Nexus 6.0.1	Samsung S6 Edge 6.0.1	Samsung Tab S 8.4 6.0.1	Samsung S7 Edge 7.0	LG G3 5.0.2	LG G4 6.0	HTC M8 5.0.2	HTC M8 6.0	Sony Xperia XA 6.0	Sony Xperia XZ 7.0
Nexus 5.0.2	21/32	13/17	17/19	38/47	35/45	39/51	7/9	28/36	9/12	24/32	26/35	36/53
Nexus 6.0	101	15/29	6/9	27/37	24/35	34/38	20/26	15/19	22/28	11/15	13/18	33/47
Nexus 6.0.1	133	55	12/26	21/28	18/26	32/40	24/28	21/28	24/31	15/22	19/27	19/36
S6 Edge 6.0.1	546	446	410	36/64	12/16	26/33	40/51	35/48	37/49	26/40	28/43	36/52
Tab S 8.4 (6.0.1)	503	422	379	212	30/53	29/35	34/48	31/46	34/47	22/37	23/39	31/46
S7 Edge 7.0	562	457	498	289	314	39/68	43/51	35/52	37/49	26/40	28/43	36/52
LG G3 5.0.2	115	96	188	338	305	468	23/41	19/26	13/17	31/41	28/39	33/43
LG G4 6.0	209	198	222	403	378	313	215	28/41	28/37	26/32	23/32	28/36
HTC M8 5.0.2	68	183	198	331	298	325	233	401	30/47	23/33	21/31	35/46
HTC M8 6.0	186	87	119	268	243	366	274	333	264	29/46	26/41	24/32
Xperia XA 6.0	183	89	123	284	252	369	274	340	271	312	32/48	16/21
Xperia XZ 7.0	246	186	221	410	389	491	305	238	294	213	247	34/54

manual inspection reveals that the FPs are mainly caused by the following limitations. First, not all identified sinks through the in-image analysis are privileged; We followed several heuristics to reduce insensitive sinks; still, there are cases that are not important. Second, some of our modeled checks might seldom serve a non-security purpose. For example, the user's restriction list might be consulted before attempting to add a new entry to it. Third, some user checks reflect specific customization needs. For example, Samsung adds additional user checks `UserId == 100`, `UserId == 200` for implementing a custom component. Similarly, HTC and LG add custom user-based checks to enforce their device administration. Through cross-vendor analysis, AceDroid cannot recognize those custom checks.

F. Improving Kratos in In-Image Analysis

In this section, we aim to show how normalization can improve the state-of-the-art system Kratos. We simulate the results of Kratos since the tool is not public. Please note that our simulation of Kratos (*SimKratos*) is our best effort to conduct the tool's analysis and might not lead to the exact results (e.g., we might miss non-explicit package-based checks that the authors annotate manually). Recall that Kratos considers the set of explicit checks for an access as the protection for the access, without normalization or considering their relations. We report the results in column 4 in Table IV `inc⋆`. In contrast, we conduct AceDroid 's in-image analysis, perform our proposed modeling and normalization and report the detected inconsistencies in column 2 in Table IV `inc⋆`. Our reported TP is the set of cases that are only semantically different (i.e., with different protections).

As shown in Table IV, SimKratos reports a larger number of inconsistencies, where on average 22% are actual inconsistencies. In comparison, AceDroid detects more TPs, an average increase of 63%. Besides, by suppressing the semantically equivalent checks, we reduce SimKratos' FP by 65%. This illustrates the importance of our technique.

TABLE IV. COMPARISON WITH SIMKRATOS

Image	AceDroid		Simkratos		TP	FP
	# Inc*	TP	# Inc*	TP	%↑	%↓
Nexus 5.0.2	32	21 (65.6 %)	53	13 (24.5 %)	62	73
Nexus 6.0	29	15 (51.7 %)	47	7 (14.9 %)	114	65
Nexus 6.0.1	26	12 (46.2 %)	45	7 (15.6 %)	71	63
S6 Edge 6.0.1	64	36 (57.8 %)	98	26 (26.5 %)	42	63
Tab S 8.4 6.0.1	53	30 (56.6 %)	92	21 (22.8 %)	43	68
S7 Edge 7.0	68	39 (57 %)	103	18 (17.5 %)	56	74
LG G3 5.0.2	41	23 (57 %)	71	16 (22.1 %)	44	68
LG G4 6.0	41	28 (63.3 %)	71	17 (23.6 %)	43	64
HTC M8 5.0.2	47	30 (63.8 %)	71	18 (25.4 %)	67	68
HTC M8 6.0	46	29 (63 %)	68	18 (26.5 %)	61	66
Xperia XA 6.0	48	32 (66 %)	69	18 (26.1 %)	72	69
Xperia XZ 7.0	54	34 (62 %)	75	20 (26.7 %)	70	64

TABLE V. CONFIRMED ATTACKS

Security Impact	Description	Victim Device(s)
Privilege Escalation	Eavesdropping on input events such as screen taps	LG G4 6.0
Privilege Escalation	Intercepting and injecting input events such as screen taps	LG G4 6.0
Privilege Escalation	Sending SMS messages including premium messages	S6 Edge 6.0.1
DoS	Denying receiving of SMS messages	S6 Edge (6.0.1) HTC M8 6.0
Privilege Escalation	Enabling Bluetooth Quietly	S6 Edge (6.0.1) LG G4 6.0
Privilege Escalation	Persist Bluetooth Settings	LG G4
Privilege Escalation	Bypassing and Forging User Restrictions	S6 Edge 6.0.1 S7 Edge (7.0)
Privilege Escalation	Injecting Hard Key Events such as Volume Up, Power Off, Screen Off	Sony Xperia 6.0
Privilege Escalation	Rebooting the phone into Recovery Mode	Sony Xperia 6.0
Privilege Escalation	Phone Shutdown	Sony Xperia XA 6.0
Privilege Escalation	Turning Radio On / Off	LG G3 5.0.2
DoS	Unmounting SD Card persistently	HTC M8 6.0
DoS	Turning-Off Wifi persistently	HTC M8 6.0
DoS	Turning-Off Bluetooth persistently	LG G3 5.0.2
Privilege Escalation	Manipulating Network Firewall Rules	Xperia XA 6.0

VII. FINDINGS

We analyzed the reported consistencies and found 73 actual inconsistencies. We picked the instances for which we had the physical device and confirmed 27 actual vulnerabilities that can be exploited to conduct different attacks. Table V summarizes some of the confirmed vulnerabilities. We reported the high profile attacks discovered through our analysis to the corresponding vendors (Samsung, LG, HTC, Sony). More than 20 of our reported vulnerabilities have been acknowledged and reproduced by the vendors. Other cases are still being investigated by the vendors. Due to space limitations, we discuss 7 attacks, where 2 are ranked as Critical by the corresponding vendors.

Triggering a System Shutdown in Sony. AceDroid discovered several exploitable inconsistencies in Sony devices allowing to trigger a system shutdown; an operation always reserved to the system/ preloaded apps. A notable inconsistency discovered through comparing two Sony devices is depicted in Figure 7. As illustrated in Sony Xperia XA's simplified program paths (on the left), the vendor introduces a new public entry point (i.e., `DevicePolicyManager.reboot()`) enforcing disjoint privilege checks: The caller needs to satisfy either one of three UID checks (1000, 1001,..) or possess the normal level permission

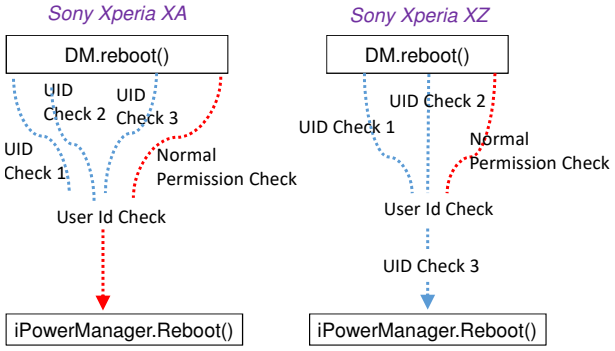

Fig. 7. `DevicePolicyManager.Reboot()` across two Sony devices

(`com.sonymobile.permission.ENTERPRISE_API`). Additionally, the calling app should satisfy the user privilege check (UserId = 0). The API then invokes the API `reboot()` with the `PowerManagerService` after escalating to system privilege. Clearly, this API is vulnerable as a calling app can simply meet the normal permission requirement (and the userId = 0) to trigger the system shutdown. On the contrary, AceDroid found out that the same custom API on Sony Xperia XZ is well protected. As illustrated on the right side of the same figure, `reboot()` enforces a conjoint UID requirement at the end of the execution (which was a disjoint requirement in Xperia XA). As a result, a calling app cannot exploit the normal permission requirement to trigger the reboot as it needs to satisfy the additional UID check.

Although the main strength of our path-sensitive analysis and normalization lies in suppressing false positives, this specific case proves that it is also powerful for discovering important inconsistencies that cannot otherwise be found without sophisticated analysis; e.g. Kratos would fail to discover this case as the union of the access control checks would look exactly the same for the two instances.

Eavesdropping Screen Taps on LG. We discovered through our cross-image analysis that LG G4 (6.0)'s `WindowManagerService` has exposed a sensitive API, which is internal in other images (e.g., Nexus 6.0), without taking any security measures. Specifically, the exposed API allows retrieving an `InputChannel` instance of a given input device, and thus can be leveraged to monitor screen tap coordinates received by the window manager. Since the API does not enforce any status check for the calling app, we were able to invoke this API in a background service and successfully eavesdrop on the user's input taps coordinates (e.g., soft key strokes to infer typed text, etc).

Injecting Input Events on LG. Similarly, AceDroid revealed through the cross-image analysis that LG G4 further exposes another critical API, which is private on other images. The API allows registering an `InputFilter` in the custom LG's `WindowManagerService`, without any security checks. The vulnerable API can be exploited similarly to intercept all screen tap events. More importantly, the `InputFilter`'s callback method, further allows to inject new input events. Given the capabilities it allows, an attacker can leverage the exposed API to write a powerful tool to achieve critical operations; e.g., inferring the user's password in a banking app, triggering a money transfer order, etc.

Bypassing and Tampering with User Restrictions. Interestingly, we discovered through our analysis that the user restrictions (employed to restrict user capabilities as discussed in Section III-C3) can be manipulated by a non-privileged caller in several Samsung devices. Specifically, our inconsistency results for the `UserManagerService` shows that the internal API for updating the user restrictions can be reached through multiple entry points enforcing different security checks. One entry point verifies whether the caller is System UID, root UID, or has the the signature permission `MANAGE_USERS` before updating the user restrictions. The second entry point, added by Samsung, invokes the internal API without enforcing any security checks, enabling a non-privileged app to maliciously manipulate the restrictions. First, an attacker can exploit the new entry point to bypass existing restrictions set by the device owner or administrator (e.g. access restricted SMS operations through setting `DISALLOW_SMS` to false). More dangerously, the vulnerability can cause a major denial of service by disabling all operations that can be controlled through the user restrictions list (e.g., disallowing to perform outgoing calls, to configure system wide settings, etc). The attacking app can even prevent its uninstallation through setting the restrictions `DISALLOW_UNINSTALL_APPS` and `DISALLOW_SAFE_BOOT` to true.

Forging Premium SMS messages. The SMS Manager allows sending SMS messages through the exposed API `sendText()`. Figure 8 shows the flow of this API's implementation. As illustrated, once an app invokes `sendText()`, the service verifies that the calling app is granted the `SEND_SMS` permission and that the calling user does not have any restrictions on sending SMS messages (i.e., restriction `DISALLOW_SMS`). It then calls the internal `sendText()` in `SMSDispatcher`. AceDroid discovered that Samsung S6 edge (6.0) registers a new broadcast receiver in the same class, allowing to invoke the internal `SendText()` API. Surprisingly, this broadcast receiver is not protected, enabling any app to send SMS messages without the required `SEND_SMS` permission and allowing users to bypass the `DISALLOW_SMS` restriction. Even worse, this receiver can be further exploited to send premium SMS messages without requiring user confirmation. Once the request is received, the service inspects the destination phone number of the SMS message before invoking `sendSMS()` that performs the actual sending of the SMS. If the destination number is a premium short code, the service verifies whether the caller has the system permission `SEND_RESPOND_VIA_MESSAGE`, otherwise it prompts the user to confirm sending the premium SMS as it imposes monetary charges. Clearly, enforcing the permission `SEND_RESPOND_VIA_ MESSAGE` inside the context of the broadcast receiver is not effective at all, as it implies checking whether the SMS service itself holds the permission.

SMS Receipt DoS. We discovered that another component related to the SMS functionalities is customized through additional broadcast receivers. Notably, the `SmsStorageMonitor` keeps track of available SMS storage space through listening to the protected broadcasts `ACTION_DEVICE_STORAGE_FULL` and `ACTION_DEVICE_STORAGE_NOT_FULL`. If the former broadcast is received, the monitor sets the field `mStorageAvailable` to *false*, otherwise to

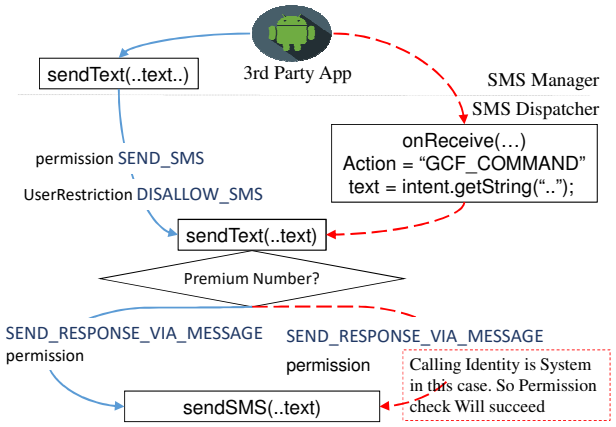

Fig. 8. Sending SMS Paths

true. Upon receiving a new SMS from the RIL, the `InboundSMSHandler` consults this field to check if there is enough memory to store the incoming SMS and subsequently dispatches it to the SMS app. Our analysis reveals that Samsung and HTC added two non-protected broadcasts updating the field `mStorageAvailable` in a similar manner. Non-system apps can exploit these broadcasts to stop users from receiving SMS messages.

Enabling Bluetooth Quietly. The `Bluetooth ManagerService` saves the bluetooth state across boot-ups in the `Settings.Global` provider. Our analysis reveals that updating the bluetooth entry in the provider can be reached through different paths. The API `disable` sets the state to Off after verifying that the calling app is either SYSTEM, or the user is in the background and the app holds `BLUETOOTH_ADMIN` permission. LG and Samsung add other paths updating the state without user status or app privilege checks. Consequently, an app belonging to a non-active user can alter the bluetooth settings without any permissions.

VIII. Related Work

Security risks in Android customization. The vendor customizations have been proven to be problematic in prior studies. ADDICTED [37] finds under-protected Linux drivers on customized ROMs by comparing them with their counterparts on AOSP images. Harehunter [7] reveals the Hanging Attributes References vulnerability caused by the under-regulated Android customization. The Hare vulnerability happens when an attribute is used on a device but the party defining it has been removed. A malicious app can then fill the gap to acquire critical capabilities, by simply disguising as the owner of the attribute. Another prominent research work analyzes pre-installed apps and reports the presence of known problems such as over-privileged apps, permission re-delegation [15], [14]. Gallo et al [16] analyzed five different custom devices and concluded that serious security issues such as poorer permission control grow sharply with the level of customization. More recently, Zhang et al. analyzed ION related vulnerabilities caused by the customization it undergoes through different devices [35].

Vulnerability detection on Android. The high flexibility of Android's security architecture demands a complete understanding of the permission model. Stowaway [14] and PScout [10] lead the way by mapping individual APIs to the required permission. Recently, Axplorer [11] produces improved mappings based on an accurate static analysis of the framework. This understanding has inspired us to conduct our normalization analysis. It has inspired other researchers to identify vulnerabilities at apps and framework. Prominent examples include the re-delegation problem [15], content provider leaks [20], issues in push-cloud messaging [27], in the app uninstallation process [36], crypto misuse in apps [12], [24] and others [13]. In addition, Whyper [30] and AutoCog [31] check the inconsistency between an app's permissions and its description. AsDroid [22] leverages the inconsistency between the code and GUI to detect malicious behaviors. AAPL [28] examines inconsistent behaviors within similar functionalities of similar apps to detect privacy leaks. Our work is fundamentally different from all these efforts, as we do not focus on apps nor specific services. Rather, we aim to analyze the whole framework with regards to the consistency of security enforcements. A closely related work to our user-based modeling and resulting inconsistencies is the work [32] aiming to evaluate Android's multi-user framework from several aspects. The work reveals important vulnerabilities; however, it is based on hypotheses and manual experiments and thus cannot be applied for our purpose.

Static analysis on Android. Static analysis techniques have been proposed to address the special characteristics of Android platform. Particularly, FlowDroid [9], DroidSafe [19], AndroidLeaks [17], Amandroid [34] and BidText [21] have employed static taint analysis on Android apps for tracing information flow and detecting privacy leaks. Other tools such as Epicc [29], Didfail [25] and IccTA [26] handle other particular challenges of Android's ICC.

IX. Conclusion

Given the complexity of Android access control enforcement, it is evident that inconsistencies will be introduced when new functionalities are integrated into the AOSP code base through version updates or vendor customization. In this paper, we provide a systematic categorization of access controls employed by Android system services and propose a path-sensitive modeling and a normalization technique to address specific challenges characterizing various checks. We employed our tool to detect framework security inconsistencies in 12 Android images. Through our conduct analyses within and across images, we uncovered substantial inconsistencies, some leading to high impact security breaches.

Acknowledgement

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1748764, 1409668 , and 1320444, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Ninghui Li's work was supported in part by the United States National Science Foundation under Grant No. 1314688, and the United States ARO grant W911NF-16-1-0127. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] "Android revolution mobile device technologies," http://android-revolution-hd.blogspot.com/p/android-revolution-hd-mirror-site-var.html, last Accessed: May 13, 2017.

[2] "Baksmali: a disassembler for Android's dex format," https://code.google.com/p/smali.

[3] "Samsung updates: Latest news and firmware for your samsung devices!" http://samsung-updates.com/, accessed: 05/02/2017.

[4] "sdat2img: Convert sparse android data image (.dat) into filesystem ext4 image (.img)," https://github.com/xpirt/sdat2img.

[5] "Smali: an assembler for Android's dex format," https://github.com/JesusFreke/smali.

[6] "70 amazing Android statistics and facts(april 2017)," http://expandedramblings.com/index.php/android-statistics/, 2017.

[7] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015.

[8] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android roms via differential analysis," in *Proceedings of the 25th USENIX Conference on Security*, ser. SEC'16, 2016.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, New York, NY, USA, 2014.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228.

[11] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016.

[12] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[13] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, *Hey, You, Get Off of My Clipboard*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–161.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011.

[15] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[16] R. Gallo, P. Hongo, R. Dahab, L. C. Navarro, H. Kawakami, K. Galvão, G. Junqueira, and L. Ribeiro, "Security and system architecture: Comparison of android customizations," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15. New York, NY, USA: ACM, 2015.

[17] C. Gibler, J. Crussell, J. Erickson, and H. Chen, *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer, 2012.

[18] Google, "ART compiler," https://source.android.com/devices/tech/dalvik/, 2017.

[19] M. I. Gordon, D. Kim, J. Perkins, L. Gilhamy, N. Nguyenz, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *NDSS*, 2015.

[20] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[21] J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016.

[22] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014.

[23] IBM, "WALA: T.J. Watson Libraries for Analysis," http://wala.sourceforge.net, 2017.

[24] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 659–668. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516706

[25] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014.

[26] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis," *arXiv preprint arXiv:1404.7431*, 2014.

[27] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, "Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014.

[28] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting." in *the 2015 Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[29] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013.

[30] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, Berkeley, CA, USA, 2013.

[31] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014.

[32] P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang, and W. Du, "A systematic security evaluation of Android's multi-user framework," in *Mobile Security Technologies (MoST) 2014*, ser. MoST'14, San Jose, CA, USA, May 17 2014.

[33] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *Proc. of ISOC NDSS*, 2016.

[34] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014.

[35] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. ACM, 2016.

[36] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du, "Life after app uninstallation: Are the data still alive? data residue attacks on android," ser. NDSS '16.

[37] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA*.