Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework

Yuru Shao, Jason Ott†, Qi Alfred Chen, Zhiyun Qian†, Z. Morley Mao
University of Michigan, †University of California, Riverside
{yurushao, alfchen, zmao}@umich.edu, jott002@ucr.edu, zhiyunq@cs.ucr.edu

***Abstract*—The Android framework utilizes a permission-based security model, which is essentially a variation of the ACL-based access control mechanism. This security model provides controlled access to various system resources. Access control systems are known to be vulnerable to anomalies in security policies, such as inconsistency. In this work, we focus on inconsistent security enforcement within the Android framework, motivated by the recent work which discovered such vulnerabilities. They include stealthily taking pictures in the background and recording keystrokes without any permissions, posing security and privacy risks to Android users. Identifying such inconsistencies is generally difficult, especially in complicated and large codebases such as the Android framework.**

Our work is the first to propose a methodology to systematically uncover the inconsistency in security policy enforcement in Android. We do not assume Android's security policies are known and focus only on inconsistent enforcement. We propose Kratos, a tool that leverages static analysis to build a precise call graph for identifying paths that allow third-party applications with insufficient privilege to access sensitive resources, violating security policies. Kratos is designed to analyze any Android system, including vendor-customized versions. Using Kratos, we have conservatively discovered at least fourteen inconsistent security enforcement cases that can lead to security check circumvention vulnerabilities across important and popular services such as the SMS service and the Wi-Fi service, incurring impact such as privilege escalation, denial of service, and soft reboot. Our findings also provide useful insights on how to proactively prevent such security enforcement inconsistency within Android.

I. Introduction

Access control is a well-known approach to prevent activities that could lead to security breach. It is widely used in all modern operating systems. Linux inherits the core UNIX security model — a form of Discretionary Access Control (DAC). To provide stronger security assurance, researchers developed Security-Enhanced Linux (SELinux) [37], which incorporates Mandatory Access Control (MAC) into the Linux kernel. The fundamental question that access control seeks to answer is philosophical in nature: "who" has "what kind of access" to "what resources." It is from this single question that access control policies or security policies are derived.

NDSS '16, 21-24 February 2016, San Diego, CA, USA
 ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23046

Android OS employs a permission-based security model, which is a derivative of the Access Control List (ACL) based access control mechanism [14]. In this model, an application (commonly known as an *app*) or a user may request access to a set of resources that are governed by a set of permissions, exposed by the system or other apps.

Access control systems are known to be vulnerable to anomalies in security policies, such as inconsistency [36]. However, security policies can be inconsistent not only in their definitions, but also in the ways they are enforced [38]. A major challenge of supporting permission-based security models, as well as other access control systems, is to ensure that *all* sensitive operations on *all* objects are correctly protected by proper security checks in a consistent manner. If the proper security check is missing before a sensitive operation, an attacker with insufficient privilege may then perform the security-sensitive operation, violating user privacy or causing damage to the user or the system. For example, on Linux, multiple such examples have been discovered, which lead to unauthorized user account access [10], permanent data loss [38], etc. More recently, on the Android platform, attacks caused by inconsistent policy enforcement have also been found, e.g., stealthily taking pictures in the background [18] and stealing user passwords by recording keystrokes without the necessary permissions [46]. Therefore, to ensure a safe and secure platform for users and developers, it is critical to develop a systematic approach to identify inconsistencies in security policy enforcement.

To address this problem on MAC-based operating systems, Tan *et al.* present a method and tool, *AutoISES* [38], that can automatically infer security policies by statically analyzing source code and then directly using those policies to detect security violations. Its effectiveness has been demonstrated by experiments with the Linux kernel and the Xen hypervisor; however, AutoISES has several limitations that prevent it from being applied to the Android framework. First, it does not take into account inter-process communication (IPC) between different processes (or threads). In Android, remote method calls across process (or thread) boundaries are very common. Any static analysis that fails to consider this special feature would be hugely incomplete. Second, the Android framework consists of conflated layers: Java and C/C++, which is not currently supported by AutoISES. A fundamental limitation is that AutoISES assumes that a complete list of security check functions are given. While in Android, different types of security checks exist, making it extremely difficult to obtain a comprehensive list of them.

In view of these challenges, we propose *Kratos*, a static

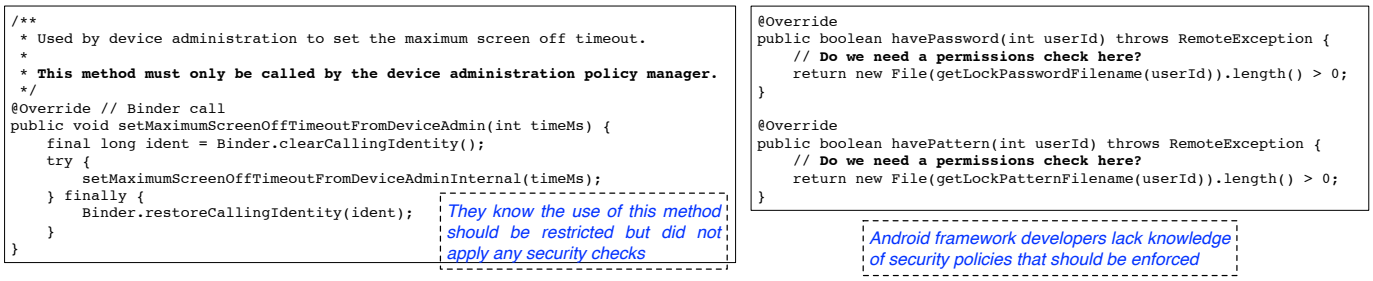

Fig. 1. Code snippets from `PowerManagerService.java` (left side) and `LockSettingsService.java` (right side) in AOSP 5.0.1. They show that even Google engineers do not have accurate knowledge of security policies that should be enforced.

analysis tool for systematically detecting inconsistent security enforcement in the Android framework. Kratos accepts Java class files and security enforcement checks as input, and outputs a ranked list of inconsistencies. It first builds a precise call graph for the codebase analyzed. This call graph comprises of all the execution paths available to access sensitive resources. Each node of the call graph is then annotated with security enforcement methods that are applied to that node. For a set of entry points, Kratos compares their sub-call graphs pairwise to identify possible paths which can reach the same sensitive methods but enforce different security checks (e.g., one with checks and the other without). Kratos can be applied to both the AOSP framework and vendor-specific frameworks.

Another thread of related work focuses on automated authorization hook placement to mediate all security-sensitive operations on shared resources. In work by Muthukumaran *et al.* [34], there is an implicit assumption made by the authors: they have perfect knowledge of which functions or pieces of code needs protection. They are able to automatically place the policy enforcement based on metrics such as the minimum number of checks needed. Not only do we lack this understanding, but also any understanding that might be inferred from the Android source code is further obfuscated by uncertainties introduced by developers, as illustrated in the code snippets in Figure 1.

Kratos makes no assumptions about or attempts to infer what resources or operations in Android framework are security-sensitive and should be protected. Instead, Kratos only identifies *where* existing security policy enforcement occurs based on observed security checks, and accurately infers security-sensitive operations by identifying inconsistency in the policy enforcement across different execution paths.

To implement Kratos, we overcome several engineering challenges. First, to maximize the completeness of our analysis, we need to cover as many system services as possible. However, system services are scattered throughout the Android framework with some implemented as private classes nested inside outer classes, making it difficult to include all service interfaces as analysis entry points. We address this by generating code that calls service interfaces and handles nested private services. Second, our analysis relies on a precise Android framework call graph, which is non-trivial to build. We tackle this challenge by resolving virtual method calls using Spark [28], a tool for performing Java points-to analysis, and applying IPC shortcuts. Third, due to the large size of code in Android framework, it is non-trivial to make the analysis efficient and scalable. We achieve high efficiency and scalability by optimizing the implementation and adopting a set of heuristics.

We run Kratos on six different Android codebases, including 4 versions of Android frameworks, 4.4, 5.0, 5.1, and M preview, and two customized Android versions on AT&T HTC One and T-Mobile Samsung Galaxy Note 3 devices. After verifying the tool output manually, we find that all six codebases fail to ensure consistent policy enforcement with at least 16 to 50 inconsistencies discovered. For one, the number of inconsistencies are as high as 102. From these inconsistencies, we are able to uncover 14 highly-exploitable vulnerabilities spanning a wide range of distinct Android services on the 6 codebases; 12 of them are found to affect at least 3 codebases at the same time; 6 vulnerabilities have been patched in the latest or earlier releases but never been revealed to the public previously. All these exploits can be carried out with no permission or only low-privileged permissions (e.g., the `INTERNET` permission), and lead to serious security and privacy breaches such as crashing the entire Android runtime, terminating mDNS daemon to make file sharing and multiplayer gaming unusable, and setting up a HTTP proxy to intercept all web traffic on the device.

We have reported all of these vulnerabilities to the Android Security Team. Among the 11 that we have received feedback, all were confirmed as low severity vulnerabilities. This indicates the challenging nature of enforcing consistent policies in a complex system like Android, and the necessity of a systematic detection tool like Kratos. Due to the lack of an up-to-date Android malware dataset, we do not have statistics on how many of these vulnerabilities have already been exploited by malicious apps in the wild. More details about the inconsistencies reported by Kratos and the vulnerabilities we identified are available at our results website `http://tinyurl.com/kratos15`.

We summarize our key contributions as follows.

- Our work is the first to systematically uncover security enforcement inconsistencies in the Android framework itself, compared to previous work focusing on permission use in Android apps. We design and implement Kratos, a static analysis tool that can effectively identify inconsistent security enforcement in both the AOSP and customized Android. We tackle several engineering challenges, including automated entry point generation, IPC edges connection, and parallelization

to ensure accuracy and efficiency.

- We evaluate our tool on four versions of the Android framework: 4.4, 5.0, 5.1, and M preview, as well as two customized: AT&T HTC One and T-Mobile Samsung Galaxy Note 3, and find that all codebases fail to ensure consistent policy enforcement with up to 102 inconsistencies in a single codebase. Among the discovered inconsistencies, we are able to uncover 14 highly-exploitable vulnerabilities, which can lead to serious security and privacy breaches such as crashing the entire Android runtime, ending phone calls, and setting up a HTTP proxy with no permissions or only low-privileged permissions.
- Our analysis covers all app-accessible interfaces exposed by system services implemented in Java code. Many system service interfaces are by default invisible and undocumented for apps. Among the 14 vulnerabilities we identified, 11 of them are hidden interfaces that are rather difficult to detect. These findings suggest useful ways to proactively prevent such security enforcement inconsistency include reducing service interfaces and restricting the use of Java reflection (for accessing hidden interfaces).

II. Background and Motivation

In this section, we cover background on Android's system services and the different types of security enforcement adopted by the Android framework. We also present a motivating case that demonstrates inconsistent security enforcement in Android's Wi-Fi service.

A. System Services

System services implement the fundamental features within Android, including display and touch screen support, telephony, and network connectivity. The number of services has slowly increased with each version: growing from 73 in Android 4.4 to 94 for M Preview. Most system services are implemented in Java with certain foundational services written in native code. At runtime, system services are running in several system processes, such as `system_server`, `mediaserver`. To provide functionality to other services and apps, each system service exposes a set of interfaces accessible from other services and apps through remote procedure calls. For simplicity, we define users (either another service or an app) of a system service as its *clients*. From a client's perspective, calling remote interfaces of a system service is equivalent to calling its local methods.

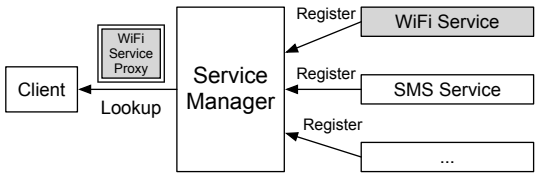

Fig. 2. System services register themselves to Service Manager and clients call their remote interfaces with proxies.

Fig. 2 depicts how system services are managed and used. When the Android runtime boots, system server registers system services to the Service Manager, which runs in an independent process `servicemanager` and governs all system services. When a client wants to call a system service, e.g., *Wi-Fi Service* in Fig. 2, it first queries the service index provided by Service Manager. If the service exists, Service Manager returns a proxy object, *Wi-Fi Service Proxy*, through which the client invokes Wi-Fi Service methods. The "contract" that both the Wi-Fi Service and the proxy agree upon is defined by Android Interface Definition Language (AIDL). Fig. 3 uses the Wi-Fi Service as an example and shows how the service and its proxy use the AIDL to define consistent interfaces.

During compiling, a class `IWifiManager` is automatically generated from the AIDL file, `IWifiManager.aidl`. It has two inner classes, i.e., `IWifiManager$Stub` and `IWifiManager$Stub$Proxy`. All interfaces defined in the AIDL file `IWifiManager.aidl` are also declared in `IWifiManager`, `IWifiManager$Stub` and `IWifiManager$Stub$Proxy`. The service extending `IWifiManager$Stub` is responsible for implement methods defined in `IWifiManager.aidl`. Clients who wish to access service functionality only need to obtain a reference to `IWifiManager` and invoke `IWifiManager`'s method. As a result, the corresponding implementation in the service will be called. The intermediate procedure is handle by Binder IPC [1] and completely transparent to clients and services.

```
1  // Client.java
2  public class Client {
3    IWifiManager mgr = IWifiManager.Stub.asInterface(
4      ServiceManager.getService("wifi"));
5    mgr.removeNetwork(netId);
6    mgr.disableNetwork(netId);
7    ...
8  }
9
10 // WifiManager.aidl
11 interface IWifiManager {
12   boolean removeNetwork(int netId);
13   boolean disableNetwork(int netId);
14   void connect();
15   void disconnect();
16   ...
17 }
18
19 // Decompiled from IWifiManager.class
20 public interface IWifiManager extends IInterface {
21   boolean removeNetwork(int netId);
22   boolean disableNetwork(int netId);
23   void connect();
24   void disconnect();
25   ...
26 }
27
28 // WifiService.java
29 public class WifiService extends IWifiManager.Stub {
30   @override
31   public boolean removeNetwork(int netId) {
32     enforceChangePermission();
33     ...
34   }
35   @override
36   public boolean disableNetwork(int netId) {
37     enforceChangePermission();
38     ...
39   }
40   ...
41 }
```

Fig. 3. Code snippet that demonstrates the usage of AIDL

In our analysis, we focus solely on system services. More specifically, we consider all remote interfaces exposed by system services as application-accessible interfaces. Note that some proxy interfaces are *invisible* to apps, either because the classes are excluded from the Android SDK or because the methods are labeled with the `@hide` or `@SystemApi` javadoc directive in the source code. However, they still exist in the runtime and apps can access them using Java reflection techniques.

B. Motivating Example of Inconsistent Security Enforcement

For convenience, we use the terms *security policy enforcement* and *security enforcement* interchangeably in this paper. Security enforcement consists of a set of security checks. *Security check* refers to a specific action which verifies whether the caller satisfies particular security requirements, e.g., holds a permission or has a specific UID. The Android framework employs several types of security enforcement: permission check, UID check, package name check, and thread status check, all explained below.

Permission checking is the most fundamental and widely used security enforcement in Android. Each app requests a set of permissions during installation The user must allow all permissions requested or choose not to install the app. When an app calls a method exposed by a system service, the service verifies that the app holds the required permission(s). If so, the app passes the permission check and continues executing. Otherwise, the service immediately throws a security exception; as a result, the app cannot access resources guarded by the service. As shown in Fig. 3 lines 31–34, `removeNetwork(int)` invokes `enforceChangePermission()` to check that the calling app has the `CHANGE_WIFI_STATE` permission. Permission checks are performed immediately after the code enters the service side. This is different from other systems such as SELinux, which places security checks right before accessing sensitive objects [32]. We believe these different approaches present tradeoffs in balancing the performance overhead and access control granularity.

Despite the permission check placement, we have observed inconsistent enforcement of permissions within the same service and between services with similar functionality. For example, in Fig. 4 we detail inconsistencies within a single service, i.e., the Wi-Fi Service. It exposes two interfaces to clients: `addOrUpdateNetwork()` and `getWifiStateMachineMessenger()`. Both of them can be leveraged by apps to update Wi-Fi configurations. Though they ultimately invoke the same underlying method `WifiConfigStore.addOrUpdateNetworkNative()`, the two paths they traverse are different. The method `addOrUpdateNetwork()` first calls `sendMessageSynchronously()`, through which it sends a `CMD_ADD_OR_UPDATE_NETWORK` message to the internal Wi-Fi state machine which is able to update Wi-Fi configurations according to current status. Meanwhile, apps can call `getWifiStateMachineMessenger()` to obtain the Wi-Fi state machine's `Messenger` object, with which they are able to directly send a `SAVE_NETWORK` message to the Wi-Fi state machine to update configurations. Surprisingly, permission checks differ along these

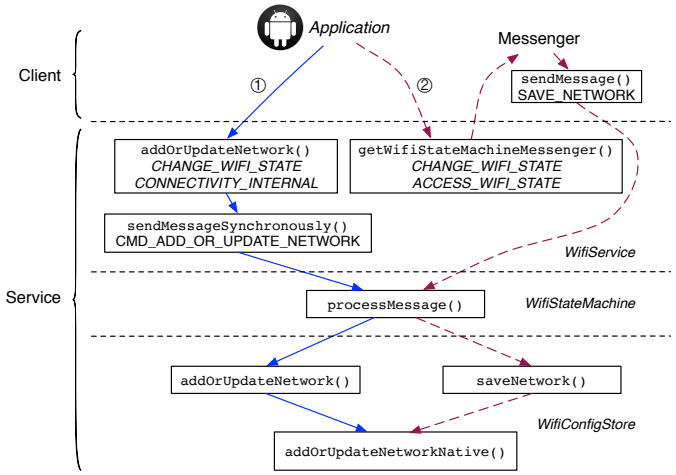

Fig. 4. A motivating case showing inconsistent security enforcement in the Wi-Fi Service of Android 4.4.4_r1. An app can use either of the two interfaces exposed by the Wi-Fi Service, `addOrUpdateNetwork()` (call chain connected by solid lines) `getWifiStateMachineMessenger()` (call chain connected by dashed lines), to update connection configurations to a Wi-Fi access point. However, they are enforcing different permissions. Note that actual call chains have been simplified to better demonstrate the example, with method parameters omitted.

two paths. `addOrUpdateNetwork()` checks two different permissions, i.e., `ACCESS_WIFI_STATE` and `CONNECTIVITY_INTERNAL`. However, only two of them are enforced in `getWifiStateMachineMessenger()`, i.e., `ACCESS_WIFI_STATE` and `CHANGE_WIFI_STATE`. Considering that `CONNECTIVITY_INTERNAL` is a system-level permission, it is impossible for third-party apps to acquire it. Thus, `addOrUpdateNetwork()` is protected from third-party app usage. Nevertheless, this enforcement can be completely bypassed if an app developer, or malware author, uses `getWifiStateMachineMessenger()` instead of `addOrUpdateNetwork()`. Similarly, Telephony Service and Telecom Service both provide methods to access telephony-related functionality. While these two services are different, they do have some overlapping functionality — they expose different methods which provide similar underlying functionality, see Section V for further discussion.

C. UID Check

Interfaces provided by a system service can be called by apps, as well as other system services. For some sensitive operations the system service only allows internal uses by checking the caller's UID. As aforementioned, service interfaces are invoked through the Binder IPC mechanism. For each AIDL method call, the system keeps track of the original caller's identity in order to check it within the service side. Fig. 5 shows a code snippet extracted from the Keyguard Service. Its `checkPermission()` method has two steps: First, it gets UID of the caller using `Binder.getCallingUid()` and verifies that the UID is equal to `SYSTEM_UID`. If so, the caller is the system and no further check is required. Otherwise, the permission check is performed.

```
void checkPermission() {
  if (Binder.getCallingUid() == Process.SYSTEM_UID)
     return;
  // Otherwise, explicitly check for caller permission
  if (checkCallingOrSelfPermission(PERMISSION)
        != PERMISSION_GRANTED) {
    ...
  }
}
```

Fig. 5. Permission check is performed if the UID check fails.

D. Package Name Check

The package name check is another means to restrict the capability of apps. For instance, in order to ensure that the client can only delete widgets that belong to itself, the App Widget Service checks whether the caller owns the given package, by using package name check shown in Fig. 6.

```
public void enforceCallFromPackage(String packageName) {
  mAppOpsManager.checkPackage(
        Binder.getCallingUid(), packageName);
}
```

Fig. 6. Check to verify the caller owns a given package.

E. Thread Status Check

Many malicious apps are reported to run in the background and stealthily jeopardize the security and privacy of end users. This is seen through the myriad of research: stealing sensitive photos [18], inferring keystrokes [44], discovering web browsing habits [29], understanding speech through the phones gyroscope [33], etc. To mitigate this, Android employs thread status checks, which are designed to ensure that certain sensitive operations cannot be performed by callers running in background. In this check, the system verifies that the caller is running in the foreground and visible to users, and considers only the operations from the foreground as performed by the user. One example of such checks in Android is implemented in the Bluetooth Manager Service. It ensures that only clients running in the foreground are able to manipulate the Bluetooth device, by checking their running status using a dedicated method named `checkIfCallerIsForegroundUser()`.

III. Methodology

In this section, we present our design of Kratos. We first give an overview of the design, followed by the key components to achieve the design goals.

A. Overview

Kratos' analysis flow consists of four phases, as shown in Fig. 7. In the first phase, we retrieve Java class files of the given Android framework, and process them to generate entry points of services for further analysis. Kratos is able to analyze both AOSP and customized Android versions. Therefore, Kratos takes Java classes as input as opposed to Java source code, since customized Android frameworks are usually closed source. In the second phase, Kratos constructs a precise call graph from the entry points generated from the previous phase. Third, we annotate the framework call graph

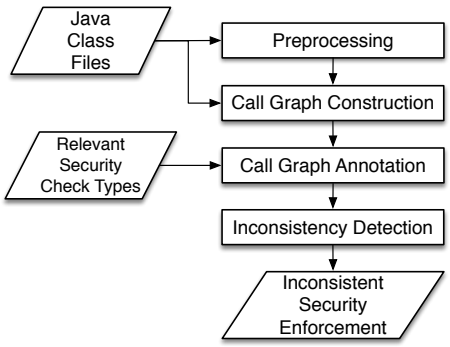

Fig. 7. Kratos analysis flow. There are four major phases: (1) Preprocessing, (2) Call Graph Construction, (3) Call Graph Annotation, and (4) Inconsistency Detection.

by considering different types of security checks of interest, e.g., permission check, UID check, package name check, and thread status check. Each node of the call graph is examined to determine which, if any, security checks exist within it. Finally, Kratos detects inconsistencies and outputs a prioritized list of security enforcement inconsistencies.

As we have illustrated in Fig. 4, system service interfaces with overlap in functionality may eventually invoke the same lower-level method(s) to complete their work. For instance, both of `addOrUpdateNetwork()` and `getWifiStateMachineMessenger()` exposed by the Wi-Fi Service can be used to update the configuration of the currently connected Wi-Fi network. Both methods invoke `WifiConfigStore.addOrUpdateNetworkNative()`, i.e., invoke the same lower-level sensitive method. Such behaviors are expected — while it is common that similar high-level functionality are provided for convenience, it is not necessary for the Android framework to implement their underlying functionality multiple times, hence the convergence at the lower-level method. Based on this observation, using a call graph to represent the execution path of a service interface, we can identify where those sensitive, or lower-level, methods are invoked by any two services. This is what we refer to as an "overlap." As a result, Kratos reduces the problem of detecting security enforcement inconsistency among system service interfaces into call graph comparisons.

B. Preprocessing

The Preprocessing step collects important information for further phases. Our analysis emphasizes system services whose implementations are scattered throughout the Android framework codebase. We must first obtain a comprehensive list of *app-accessible system services* and their corresponding Java classes, from which we can retrieve all interfaces they expose that could be invoked by apps.

As we mentioned in Section II, Service Manager manages all system services. Clients are required to obtain a proxy of the system service from Service Manager in order to invoke that service's interfaces remotely. System services that are visible for apps should be registered to *Service Manager*. In practice, besides a global service manager running in a dedicated process (`/system/bin/servicemanager`), there exist a

few local service managers. System services registered to local service managers are only accessible for other services running within the same process. Therefore, we only care about system services registered to the global Service Manager since only these are accessible by apps. By looking into a service's implementation and its corresponding AIDL definition, we easily distinguish which public methods of the service are publicly accessible AIDL methods. Although apps can invoke system services directly via low-level Binder IPC mechanism without passing through the AIDL interfaces, those Binder IPC endpoints that can be reached directly are exactly the same as those exported via the AIDL methods.

In addition to AIDL methods, we observe another type of interface exposed by system services that could be called by apps, i.e., *unprotected broadcast receivers* that are dynamically registered. Normally, system services dynamically register broadcast receivers in order to receive asynchronous messages from within the system. To defend against broadcast spoofing [19], either receivers should be protected by proper permissions or broadcast actions need to be protected. However, some broadcast receivers of system services are not protected at all. That means apps can also send crafted broadcasts to trigger certain method calls. Therefore, we also consider unprotected broadcast receivers in system services as app-accessible service interfaces.

Currently, we do not cover those system services whose main logic and security checks are all performed in native code, e.g., Camera Service. These native services are small in number: we manually checked the source code and only Camera Service, Media Player Service, Audio Policy Service, Audio Flinger and Sound Trigger Hardware Service were found in the Android 5.1 source code. While this may introduce false negatives, we believe the impact is minimal because most system services are implemented in the Java code.

C. Call Graph Construction

A precise call graph is the foundation of discovering inconsistent security policy enforcements in our approach. This phase computes the call graph for the entire Android framework. We rely on a context-insensitive call graph [27] which is light-weight and easier to build compared to a context-sensitive call graph, further discussion on the context-sensitivity may be found in Section VI.

To construct the call graph, we need to know, for each call-site, all of its possible targets. As is common for object-oriented languages, the target of a method call depends on the dynamic type of the receiving object. A polymorphic method, i.e., virtual method may have multiple implementations in descendant classes. The runtime has a dynamic dispatch mechanism for identifying and invoking the correct implementation. Unfortunately, it is impossible for static analysis to collect runtime information and identify with 100% accuracy the callees of a virtual method. To address this problem, we use a conservative way to compute possible methods that might be called at a call-site. In other words, it computes an over-estimation of the set of calls that may occur at runtime using context information. Additionally, we connect IPC callers and callees directly to improve the precision and conciseness of the call graph. This is necessary because IPCs would introduce imprecision into our call graph. They use abstract methods to send data across process boundaries and thus in that procedure, many levels of virtual method calls are involved.

We take advantage of the solution proposed by PScout [12] to resolve Binder IPC calls and Message Handler IPCs. However, PScout fails to take a few special cases into account. For example, not all system services have an AIDL file that defines their remote interfaces. For instance, instead of using a stub class auto-generated from AIDL file, Activity Manager Service relies on a manually implemented class, `ActivityManagerNative`, to define its remote interfaces. Activity Manager Service extends `ActivityManagerNative` and implements these remote interfaces. Therefore, system services like Activity Manager Service should be handled carefully with additional logic.

Moreover, PScout does not consider another important IPC that is widely used by system services — Messengers and State Machines. System services expose AIDL methods that allow callers to obtain Messenger objects of their internal state machines. With an Messenger, an app or a system service can send messages to the corresponding state machine. Although in essence the communication between Messengers and State Machines is built on top of Message Handler IPCs, we find that PScout is unable to deal with this. We identify and connect all such senders and receivers for messages sent through Messenger objects.

Entry points. Like Java programs, the Android framework has a main method `SystemServer.main()`, from which system services are initialized and started. However, we cannot use it as the entry point. All service interfaces are likely to be called by a client app, but the construction and initialization procedure of system services cannot cover all remote interfaces. As a result, the framework call graph would be incomplete if we use `SystemServer.main()` as our analysis entry point.

Preprocessing produces a list of app-accessible service interfaces. Since we would like to include all of them in the call graph, one possible approach is to build call graphs from each of the interfaces, then combine these call graphs together to form the framework's call graph. This is not efficient because many call-sites would be included and computed multiple times. To cope with this problem, for each system service we construct a *dummy main* method, in which we construct the service object and enumerate all its app-accessible interfaces. The implementation details are described in Section IV-A.

D. Call Graph Annotation

This phase annotates the framework call graph with security check information. More specifically, given the types of security checks that are of interest to Kratos, Kratos automatically determines which security checks are performed by which call graph nodes, or methods, and annotates the nodes with security enforcement information, e.g., permissions it enforces, UIDs it checks.

Identifying permission check methods. Android permissions are represented as string constants in the framework source code. When performing permission checks, a permission string is passed as an argument to a check method.

According to developer comments in the Android source code, `checkPermission` in Activity Manager Service is the only public entry point for permission checking. Therefore, methods that eventually call `checkPermission` are considered as performing permission checks. We also want to know which particular permission is checked. To achieve this, we keep track of permission string constants passed to permission check methods. We observe that a few naming patterns can indicate whether a method is a permission check method, such as `checkCallingPermission`, `enforceAccessPermission`, and `validatePermission`. Their names start with "enforce", "check" or "validate", and end with "Permission." Essentially, they are just wrappers of `checkPermission`, but we can leverage such patterns to make permission check method identification faster.

Identifying other security enforcements. Apart from permission checks, Kratos can also identify three other types of security checks automatically. For UID checks, they always get caller's UID using `Binder.getCallingUid()` and compare it with a constant integer. We use *def-use* analysis [39] to track which constant value the caller's UID is compared with. The Android framework reserves a list of UIDs and their values are defined in `android_filesystem_config.h`, from which we can identify the user that a given UID represents. Package name checks are more complicated. Besides the method shown in Fig. 6 that uses a similar approach to permission checks, there also exist package name checks that are conducted like UID checks. Similarly, we employ def-use analysis and examine if the package name returned from Package Manager Service is compared with a string. In summary, to detect package name checks, we use both approaches for identifying permission checks and UID checks. Currently there are no explicit hints that can instruct us to find a good way to identify thread status checks. Fortunately, their number is small, which allows us to identify them manually.

Annotating call graph nodes. After all security enforcement methods are identified, Kratos iterates call graph nodes and annotates them with security checks (labels) that are performed within. Labels are propagated toward the root of each sub-call graph with the union operator used to merge multiple labels at a node. This annotated call graph is then used in the next phase for detecting inconsistent security policy enforcement.

E. Inconsistency Detection

Inconsistency Detection consists of three phases. First, for every service interface Kratos obtains its sub-call graph from the framework-wide call graph and does forward analysis on it to determine which security checks must be passed to reach each node. Next, Kratos compares service interfaces' call graphs in a pairwise fashion. Those pairs invoking the same method but with different security enforcements are considered as inconsistency candidates. Finally, Kratos applies three heuristics to rule out false positives.

With the annotated framework call graph, it is easy to obtain the sub-call graph of a service interface. For a sub-call graph, Kratos traverses all its nodes and summarizes the set of security enforcements required to reach each node from the root, by accumulating security enforcements along the path from root to that node. Note that in system services `clearCallingIdentity()` is frequently used to clear the original caller's identity, or UID, and set caller identity to the system service temporarily. As Fig. 8 depicts, after certain operations, the caller's identity is restored by calling `restoreCallingIdentity()`. If a method is called between `clearCallingIdentity()` and `restoreCallingIdentity`, all security checks are successfully passed because it appears as being called by the system service, which has elevated privileges. Thus, it is unnecessary for Kratos to perform any analysis between the two function calls.

Now we have annotated sub-call graphs for each service interface. Next, we use pairwise comparisons starting from their entry points to check if they ever invoke the same method, or converge on the same node. If such a convergence point exists for any two service interfaces, we believe they overlap in functionality and examine their paths that lead to the convergence point. Specifically, we compare security enforcements along the two paths to see if they are consistent, i.e., one path has a security check while the other does not.

Reducing false positives. Not all methods are used to access system resources or perform sensitive operations. If we use arbitrary convergence between call graphs to indicate they have similar functionality, there would be a large number of false positives. For example, many methods are frequently called, such as `equal()`, `toString()`, `<init>()`, `<clinit>()`, but they are not sensitive and do not reflect the caller's functionality. To reduce the number of false positives, we investigate service interface call graphs, as well as the Android source, and design three heuristic rules.

We observed that sensitive, low-level methods reside within the service side and are not accessible to apps. The runtime does not load them into an app's execution environment. Therefore, we can filter out methods that appear in an app's runtime. To achieve this, we classify classes imported by system services into three categories: (1) classes only used by system services, (2) classes used by both services and apps, and (3) classes only used by apps. Methods from the last two categories are believed to be insensitive and we discard them.

Second, we observe that many services have paired "accessor" and "mutator" methods, whose functionality is obviously different. For example, in Window Manager Service, `getAppOrientation` and `setAppOrientation` are used to get and set the app's orientation, respectively. Similarly, there exist other method pairs in which the two have opposite functionality, such as `addGpsStatusListener` and `removeGpsStatusListener`, `startBluetoothSco` and `stopBluetoothSco`. If such methods are found overlapping, we are confident that it is a false positive.

Third, we prioritize service interface pairs by calculating the sub-call graph similarity score of each pair. We also group together system services providing similar functionality, e.g., the Telephony Service and the Telecom Service. Overlapping service interfaces belong to the same group have higher priority to be manually examined. The rationale behind this is that if two services with no explicitly connection (e.g., the Power Manager Service and the SMS Service) are found overlapping,

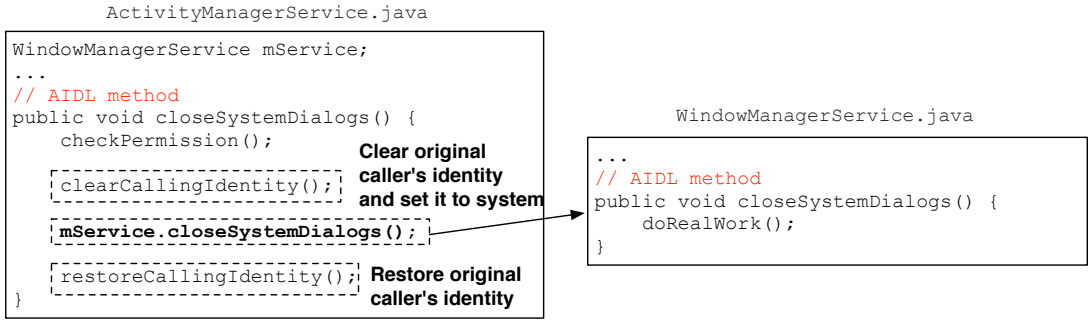

Fig. 8. Activity Manager Service calls Window Manager Service to do the real work. Since both interfaces are accessible for apps, a third-party app without any permission can call `WindowManagetService.closeSystemDialogs()` to close system dialogs, bypassing security checks in `ActivityManagerService.closeSystemDialogs()`. *code has been simplified for brevity*

it is highly likely a false positive.

IV. Implementation

We implement Kratos with around 15,000 lines of Java, Bash and Python. Based on the design described in Section III, in this section we elaborate our implementation choices of Kratos. To ensure efficiency and scalability, we make effort to parallelize the implementation.

Our implementation follows the logic shown in Fig. 7: (1) Preprocessing, (2) Call Graph Construction, (3) Call Graph Annotation, and (4) Inconsistency Detection.

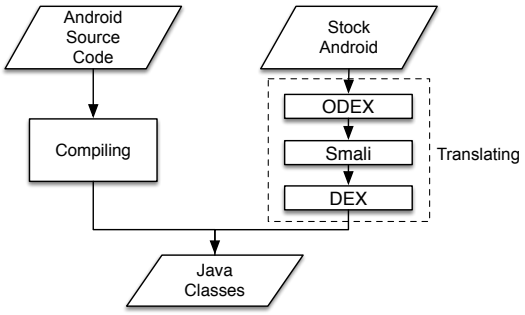

Fig. 9. Kratos can obtain Java class files from AOSP and customized Android versions

A. Preprocessing

In the Preprocessing step, we obtain the necessary class files for the particular Android framework version. In the case of analyzing AOSP, it is a matter of compiling the Android operating system from the source code and extracting the class files.

For a vendor specific version, it takes extra effort to obtain the class files. Because we do not have access to the source code of the customized framework, we must dump `odex` files from the device image, and translate them into corresponding Java `class` files. Fig. 9 shows the three steps involved in this process. We use *baksmali* [7] to convert `odex` into an intermediate format, `smali`. Then we employ *smali* [7] to assemble `smali` files into `dex`, and finally use *dex2jar* [8] to get JAR files, i.e., Java classes. We notice that since Android 5.0 the Dalvik runtime has been replaced by the Android runtime (ART) [2], in which `odex` files are no longer available. To deal with that, *Dextra* [3] can be used to dump `dex`s from ART's `oat` files.

Once the class files are obtained, we utilize the Soot framework [40], a Java decompiler and analysis tool, to parse any given class and its member bodies in order to identify which classes are app-accessible system services. More specifically, those services are identified by looking for invocations of `publishBinderService` and `addService`, two methods used for registering services to the global service manager. We exclude services registered by calling `publishLocalService`, as they are only available for system use. That means they are not accessible for third-party apps. We then distinguish app-accessible interfaces exposed by these app-accessible system services. For AIDL methods we look for their AIDL definitions, either in `aidl` files or in a public Java interface extending `IInterface`. Unprotected broadcast receivers can be identified by analyzing calls of `registerReceiver` and `registerReceiverAsUser`. If they do not have a `broadcastPermission` argument (or the argument is null) and intent actions the broadcast receiver listens to are not defined as `protected-broadcast` in the framework's `AndroidManifest.xml`, we consider this receiver as unprotected.

Once the identification of app-accessible system interfaces has finished, we take one last important step. We build an artificial single entry point for further analysis that uses Spark [28], a popular Java points-to analysis tool and call graph generator. It was designed to start at the program's single entry point, look for method calls there, then take all found callees, look at what callees call, and so on. This way, it builds a precise graph of what method is potentially called and identifies the methods which are reachable over all [6]. While the Android framework does have a static `main()` method in the System Server class, there is no guarantee that all methods will be called from that point of origin, as that is only responsible for instantiating system services. Thus, we must provide Spark a single static entry point into the Android framework for each class analyzed.

We use method and class instrumentation data structures provided by Soot to dynamically build "wrapper" classes with

a static main method. These wrappers are a necessity to meet Spark's requirement of static entry points for invoking method calls of a class. Kratos automatically builds the wrapper classes by inferring important attributes of service interfaces. Class access modifiers are one key piece of analysis. Once they are understood by Kratos, it decides how to build a wrapper. In the best case, the service is a public class; while in the worst case, the service is a private inner class.

B. Call Graph Construction

In this phase we utilize Spark to generate a context-insensitive call graph that encompasses all app-accessible service interfaces. We use the dummy main method as the single entry point. For Spark to generate the call graph, it must operate on one thread; thus we are unable to parallelize this phase. It is in this phase that Java virtual method resolution occurs, by leveraging "variable-type analysis" (VTA). We also enable the *on-the-fly* option, because it was reported that the most effective call graph construction method proceeds on-the-fly and builds the call graph at the same time as it computes points-to set [30].

C. Inconsistency Detection

In order to discover inconsistencies within security enforcements, we intelligently match two call graphs of different service interfaces in a pairwise fashion. Because this phase can run, at worst, in $O(n^2)$ time, we use a set of heuristics in order to reduce the total number of comparisons, which is outlined in Section III. Once two call graphs are paired, we look for a point of convergence — a method whereby both paths will intersect. Once we find an intersection, we use backward analysis to identify any other services that can reach the method. This allows Kratos to quickly identify additional services which may share that path in which security circumventions occur.

To prioritize the results for manual validation, we use fast belief propagation to measure node affinity, and then calculate sub-call graph similarity score with the Matusita distance. Denote the final affinity score matrix as S, we have $S = [I + \epsilon^2 D - \epsilon A]^{-1}$ where I is the identity matrix, D is the degree matrix, A is the adjacency matrix, and ϵ is a small number which we take the value of 0.02. Matusita distance d is defined as $d = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}(\sqrt{S_{1,ij}} - \sqrt{S_{2,ij}})}$ where $S_{1,ij}$ and $S_{2,ij}$ are entries of S for the subgraphs. And the similarity score $sim = \frac{1}{1+d}$ is then calculated.

V. Results

In this section, we evaluate Kratos' effectiveness, accuracy, and efficiency by applying it to six different Android frameworks. We also present vulnerabilities identified using Kratos and analyze some of them in detail. All our experiments are conducted on a desktop machine with a 3.60GHz 8-core Intel Core i7 CPU and 16GB memory, running 64-bit Ubuntu Linux 14.04.

Codebases. We target both AOSP Android and customized Android. Since the Android framework is evolving over time, in addition to inconsistencies within a particular Android framework codebase, we also track inconsistencies across different Android versions. Therefore, we choose four most recent releases, i.e., Android 4.4, 5.0, 5.1, and 6.0. Vendors and carriers often change existing or add new code to provide a unique and differing experience. Previous work [43], [26] reported security threats brought by such customizations. We believe that they may also lead to more inconsistencies as different parties are involved, and their engineers are likely to have different understanding of the security policy. Specifically, we analyze two customized Android frameworks, AT&T HTC One and T-Mobile Samsung Galaxy Note 3, both based on Android 4.4.2.

Table I summarizes the statistics of the six Android framework codebases in our evaluation. For the AOSP Android, the number of services increases dramatically from version 4.4 to 5.0, then remains unchanged in 5.1 and M preview. However, as the second column shows, the number of AIDL methods exposed by system services drops by 20 in M preview. It is obvious that Samsung and T-Mobile customize Android much more heavily than HTC and AT&T (mostly contributed by Samsung). Though both phones are based on the same version of AOSP codebase, Galaxy Note 3 has 89 more system services while HTC One only has 9 more. Moreover, customization also increases the number of service interfaces, as well as class files.

Tool Efficiency. We measure Kratos's efficiency and summarize the results in Table II. The Preprocessing phase only takes a few minutes. Call Graph Construction and Call Graph Annotation are very fast, each finishing within one minute. Even though Inconsistency Detection consumes the majority of processing time, we can analyze a framework codebase in less than 20 minutes.

A. Tool Effectiveness

Table V summarizes our overall analysis and detection results on all six Android framework codebases. The first column is the number of inconsistent security policy enforcement Kratos discovered. To evaluate true positive (TP) and false positive (FP), we manually examine all cases of enforcement inconsistency. The numbers of true positives and false positives are listed in column 2 and column 3, respectively. We also manually validate exploitable inconsistencies for each codebase, and show the results in the last column. We consider inconsistent enforcement cases which can be exploited by a third-party app as exploitable, as they are more likely to result in real-world attacks.

For the four AOSP codebases, Kratos reports more inconsistencies in newer versions. There are only 21 inconsistencies in Android 4.4 framework. However, this number drastically increases to 61 in the later version, Android 5.0. This is to be expected, as shown in Table V, Android 5.0 introduces 19 more system services. More interestingly, many of the new system services seem to have similar functionality to existing ones. For example, the RTT (round trip time) Service introduced in Android 5.0 can be used to measure round trip time of accessible Wi-Fi access points nearby. Incidentally, the Wi-Fi Service also provides similar functionality. Another example is Telecom Service, whose functionality overlaps with Telephony Service. Meanwhile, the large number of system services added by T-Mobile and Samsung undoubtedly introduce more inconsistencies.

TABLE I. STATISTICS OF THE SIX CODEBASES IN OUR EVALUATION. WE ONLY CONSIDER SERVICES IMPLEMENTED IN JAVA.

Codebase	# Services	# Service Interfaces		# Class Files
		# AIDL Methods	# Broadcast Receivers	
Android 4.4	70	1,010	26	14,901
Android 5.0	89	1,483	28	33,110
Android 5.1	89	1,510	31	33,433
Android M Preview	89	1,490	31	35,431
AT&T HTC One (Android 4.4.2)	85	1,868	35	17,879
T-Mobile Samsung Galaxy Note 3 (Android 4.4.2)	159	2,463	64	171,306

TABLE II. TIME CONSUMED IN EACH ANALYSIS STEP OF KRATOS (IN SECONDS)

Codebase	Preprocessing	CG Construction	CG Annotation	Inconsistency Detection
Android 4.4	95.4	23.4	8.6	470.3
Android 5.0	137.1	25.0	10.53	496.4
Android 5.1	209.0	22.2	14.6	445.9
Android M Preview	141.6	21.6	9.7	482.3
AT&T HTC One (Android 4.4.2)	110.8	29.1	16.0	655.8
T-Mobile Samsung Galaxy Note 3 (Android 4.4.2)	306.9	57.5	50.7	1273.7

TABLE III. OVERALL RESULTS OF KRATOS. THE NUMBERS OF EXPLOITABLE INCONSISTENCIES, TRUE POSITIVES AND FALSE POSITIVES ARE CONCLUDED BY MANUAL ANALYSIS.

Codebase	# Inconsistencies	# TP	# FP	Precision	# Exploitable
Android 4.4	21	16	5	76.2%	8
Android 5.0	61	50	11	82.0%	11
Android 5.1	63	49	14	77.8%	10
Android M	73	58	15	79.5%	8
AT&T HTC One (Android 4.4.2)	29	20	9	69.0%	8
T-Mobile Samsung Galaxy Note 3 (Android 4.4.2)	128	102	26	79.7%	10

True positive and false positive. For all codebases except the one from HTC One, Kratos can achieve more than 75% precision. We cannot measure false positive rate because we do not have other sources of data with ground truth of known inconsistencies. Therefore, it is not feasible for us to calculate the number of true negatives and false negatives. We further analyze false positive cases and try to understand why they occur. We find that most false positives are caused by the three limitations of Kratos. First, two service interfaces are not equivalent in functionality, yet they invoke the same underlying sensitive method, which is invoked with different arguments. Since Kratos uses path-insensitive analysis, it cannot discern the impact differing arguments has on the execution path. For example, Account Manager Service has two public interfaces: `getAccounts()` and `getAccountsForPackage()`. The former can list all accounts of any type registered on the device, while the latter returns the list of accounts that the calling packages is authorized to use. They eventually call `getAccountsAsUser()` with different arguments, and `getAccountsForPackage()` has one more security check — a UID check which ensures that the caller is an authorized user.

The second limitation is the inaccuracy of the overlapping service interfaces reported by Kratos. As we have mentioned in Section III, the over-estimated call graph could introduce false positives. Spark utilizes point-to analysis, which makes every effort to resolve virtual method calls according to context. Nevertheless, it cannot resolve all virtual methods with 100% accuracy.

The third limitation also comes from service interfaces with similar but not equivalent functionality. One might be more capable than the other one, and the service with more capability is guarded by stricter security enforcement. For example, `deleteHost()` and `deleteAllHosts()` from App Widget Service are able to delete host records. They both call `deleteHostLocked()`. The difference is `deleteHost()` calls `deleteHostLocked()` only once, while `deleteAllHosts()` calls it multiple times in a loop. The latter appears to be more powerful, as it can delete all host records while the former can only delete one record per call. Compared to `deteleHost()`, `deleteAllHosts()` checks a caller's UID. Kratos is not able to recognize method body semantics in order to evaluate a service interface's capability.

Not all inconsistencies are exploitable. Note that among all true inconsistency cases, only a small portion of them (18.3%) are exploitable by a third party. The reason is threefold. First, they may both require system-level permissions. Our attack model assumes that an attacker builds a third-party app and manages to have it installed on a victim's Android device. While it is possible for the methods to be invoked, system-level permissions are inaccessible by third-party apps. Two services in the "HTC One" code base provide telephony functionality, i.e., HtcTelephony Service and HtcTelephonyInternal Service. They both expose an interface `setUserDataEnabled()` for enabling and disabling cellular data connection, and both invoke `Phone.setUserDataEanbled()` to finish the request. Kratos reports that permissions used in the enforcement are different. HtcTelephony only enforces `APP_SHARED`, but HtcTelephonyInternal enforces `APP_SHARED` together with `CHANGE_PHONE_STATE`. We cannot exploit this inconsistency, because `APP_SHARED` is a system-level permission.

The second reason that an identified inconsistency is not exploitable stems from the difficulty to construct valid arguments for calling a service interface without a required permission, even though the interface has a

weaker security enforcement than another service interface providing the same functionality. For instance, Connectivity Service exposes `isActiveNetworkMetered()` and Network Policy Management Service defines `isNetworkMetered(NetworkState)` to allow callers to query if the active network is metered. Kratos reports that `isActiveNetworkMetered()` enforces a permission `ACCESS_NETWORK_STATE`, but `isNetworkMetered(NetworkState)` does not. This is a true inconsistency. However, to invoke `isNetworkMetered(NetworkState)`, one must obtain or instantiate a `NetworkState` object, which requires the `ACCESS_NETWORK_STATE` permission. In the end, the same permission is required in order to successfully invoke these two interfaces.

Third, the existence of feature checking logics makes it difficult to reach to particular methods. Some resources could be accessed only when a certain feature is satisfied. Sometimes, a security checking function is not directly called, and instead an object (could either be a flag or instance of another class) is verified where the object itself will only be valid if a security check is passed.

Characteristics of the vulnerabilities. Interestingly, we find many vulnerabilities are discovered only when we analyze hidden interfaces. In fact, 11 of them are exploitable through hidden interfaces that are not directly visible to apps. Theoretically, these hidden interfaces are not expected to be used by developers, but Android does not restrict apps to access them through Java reflection. This finding suggests that hidden interfaces are not carefully scrutinized. Perhaps disabling reflection would be one way to reduce such attack surface.

In addition, we find 3 vulnerabilities are discovered by analyzing two different services which performed the same sensitive operation, which shows that functionalities are sometimes redundant across services. Besides, we find 4 vulnerabilities where a system permission is bypassed, allowing a third-party app to perform operations that are absolutely disallowed by Android.

In summary, these results demonstrate that although human efforts are indispensable, Kratos is effective at automatically detecting a variety of inconsistent security enforcement. Based on the cases identified, Kratos is able to uncover previously unknown vulnerabilities.

B. Case Studies

By analyzing security enforcement inconsistencies reported by Kratos, we have discovered 14 vulnerabilities, summarized in Table IV. Items in Column 1 labeled with † indicate the inconsistency occurs between two services. Permissions labeled with $*$ are system permissions that cannot be used by third-party apps. Specific UIDs that can be bypassed are parenthesized. N/A means the vulnerability only exists in customized Android and does not affect other frameworks.

We have filed 8 security reports regarding these vulnerabilities to the Android security team. All of the vulnerabilities we reported have been acknowledged and confirmed. Among them, the mDNS daemon vulnerability was originally classified as a high severity vulnerability, but then rated as low severity. The ones in Wi-Fi Service and Power Manager Service had been fixed before we reported it. Note that we are very conservative about the results, which means those confirmed in code but have not been validated in real devices are not counted.

In this section we select several vulnerabilities shown in Table IV and explain them in detail. Due to space constraints, we cannot provide code-level details, and refer to the reviewers to visit our anonymized result website **http://tinyurl.com/kratos15** [4] for more information.

Starting/Terminating mDNS daemon (denial of service). The multicast Domain Name System (mDNS) provides the ability to perform DNS-like operations on the local area network in the absence of any conventional Unicast DNS server [5]. Android starts an mDNS daemon `mdnsd` when the system boots up. This daemon is used and controlled by the Network Service Discovery (NSD) service, which allows an app to identify other devices on the local network that support the services it requests [9]. It is useful for a variety of peer-to-peer apps such as file sharing and multiplayer gaming. NSD Service exposes an interface that is able to start and terminate `mdnsd`. Considering the importance of the mDNS daemon, that interface is protected by a system-level permission, `CONNECTIVITY_INTERNAL`, which cannot be acquired by third-party apps. Therefore, by design, all attempts made by third-party apps to start or terminate the mDNS daemon is thwarted.

However, another interface exposed by the NSD Service, `getMessenger()`, could be used to achieve the exact same functionality. By calling `getMessenger()`, the caller obtains a reference of the Messenger object from NsdStateMachine that manages the communication with `mdnsd`, then can send messages to it. Compared to the interface protected by the `CONNECTIVITY_INTERNAL` permission, this interface only checks the `INTERNET` permission, which is one of the most frequently requested permissions [42]. Considering that the `INTERNET` permission is so commonly used and low-privilege, apps that request it do not raise a user's attention. NsdStateMachine distinguishes different types of incoming messages by examining their `what` field. A malicious app with only `INTERNET` permission can easily craft a message, set its `what` field to `NsdManager.DISABLE`, and send it to NsdStateMachine, to terminate the mDNS daemon. As a result, users can no longer use apps that rely on the NSD Service.

Ending phone calls (privilege escalation). Both Telephony Service and Telecom Service provide a method `endCall()` that allows caller apps to reject incoming phone calls and end ongoing phone calls. According to Android's source code, their corresponding wrappers, `TelephonyManager.endCall()` and `TelecomManager.endCall()`, are annotated with `@hide` and `@SystemApi`, respectively. That means both should only be used by the system. In fact, Telecom Service's `endCall()` indeed enforces a system permission, `MODIFY_PHONE_STATE`, ensuring that only the system is able to use it. However, Telephony Service's `endCall()` only checks the `CALL_PHONE` permission, which can be acquired by third-party apps. More interestingly, a component of Telephony Service registers a broadcast receiver in which

TABLE IV. SUMMARY OF INCONSISTENT SECURITY ENFORCEMENT THAT CAN LEAD TO SECURITY POLICY VIOLATIONS.

Service	Affected Framework						Description	Security Implication	Bypassed Security Enforcement
	AT&T HTC	T-Mobile Samsung	4.4	5.0	5.1	M Preview			
SMS	✓	✓	✓	✓	✓	✗	Clear all SMS notifications showing in the status bar	Privilege escalation	Package Name (SMS)
Wi-Fi	✓	✓	✓	✗	✗	✗	Set up an HTTP proxy that works in PAC mode	Privilege escalation	CONNECTIVITY_INTERNAL*
NSD	✓	✓	✓	✓	✓	✓	Enable/Disable mDNS daemon with only INTERNET permission	DoS	CONNECTIVITY_INTERNAL*
RTT	✗	✗	✗	✓	✓	✓	Crash the Android runtime	Soft reboot	ACCESS_WIFI_STATE
Wi-Fi Scanning	✗	✗	✗	✓	✓	✓	Crash the Android runtime	Soft reboot	ACCESS_WIFI_STATE
GPS	✓	✓	✓	✓	✓	✗	(1) Send raw data to GPS's native interface (2) Crash the Android runtime	Privilege escalation, Soft reboot	ACCESS_FINE_LOCATION
GPS	✓	✓	✓	✓	✓	✓	Get GPS providers that meet given criteria	Privilege escalation	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION
Input Method Management	✓	✓	✓	✓	✓	✓	Dismiss input method selection dialog	DoS	UID (SYSTEM)
Telephony/Telecom†	✗	✗	✗	✓	✓	✓	End phone calls without any permissions	Privilege escalation	MODIFY_PHONE_STATE* CALL_PHONE
Telecom	✗	✗	✗	✓	✓	✓	Get phone state without any permissions	Privilege escalation	READ_PHONE_STATE
Activity Manager/ Window Manager†	✓	✓	✓	✓	✓	✓	Close system dialogs	DoS	UID (SYSTEM)
Power Manager/ Persona Manager†	✓	✓	✓	✓	✗	✗	Set maximum screen timeout	Draining battery	UID (ADMIN, SYSTEM)
Device Info	N/A	✓	N/A	N/A	N/A	N/A	Save MMS to audit database	Privilege escalation	UID (PHONE)
Phone Interface Manager Ext	N/A	✓	N/A	N/A	N/A	N/A	Send raw request to radio interface layer (RIL)	Not clear	MODIFY_PHONE_STATE*

phone calls are ended when a specific broadcast comes in. Unfortunately, this broadcast receiver is not protected at all, making it possible for an app without any permissions to end phone calls. This inconsistent security enforcement allows an attacker to create denial of service attacks against compromised devices. It could also be exploited by ransomware to make victims' phones unusable.

Dismissing all SMS notifications (privilege escalation). A `MessagingNotification` object instantiated in SMS Service registers a broadcast receiver to listen to message deleting broadcast. If such broadcast is received, it clears all SMS notifications showing in the status bar. Kratos reports that this receiver is not protected by any permission, and it has similar functionality to Notification Service. By design, notifications can only be dismissed by their owners or the system, enforced by package name check. However, we can bypass this enforcement by sending the broadcast to SMS Service. This bug only affects Android 5.1 and earlier versions because "the MMS app no longer ships with latest versions of the OS."

Crashing the Android runtime (soft reboot). Wi-Fi Scanning Service provides a way to scan the Wi-Fi universe around the device. Similar functionality is provided by Wi-Fi Service as well. They both leverage the `WifiNative` class that is responsible for communicating with the native binary `wpa_supplicant`, from which Wi-Fi scanning results can be obtained. Kratos reports that Wi-Fi Service checks `ACCESS_WIFI_STATE` permission, while Wi-Fi Scanning Service has no permission enforcement. We attempt to exploit this inconsistency to query Wi-Fi scanning reports without declaring any permissions. It turns out due to implementation issues of the Wi-Fi Scanning Service, it causes crash of the entire Android runtime.

We further analyze the source code and crash log. In fact, we could successfully trigger the invocation of `WifiNative.startScanNative`. Since we have to stop the scanning before reading the results, we attempt to call `WifiNative.stopScanNative`, in which a runtime exception is thrown out at line 65 of `art/runtime/check_jni.cc`. The exception is not handled, therefore the runtime crashes, causing a soft reboot.

Setting maximum screen timeout (draining battery). In T-Mobile's Samsung Galaxy Note 3, Kratos uncovered two service interfaces with exactly the same name but different security enforcement for calling `setMaximumScreenOffTimeoutFromDeviceAdmin()`. One is exposed by Power Manager Service from the AOSP codebase based on which customization was made; another is exposed by Persona Manager Service from customized portion. These two methods implement the same functionality but their security enforcement is inconsistent. Power Manager Service does not apply any security checks on its `setMaximumScreenOffTimeoutFromDeviceAdmin()`, however, Persona Manager Service checks the caller's UID.

The name of the method implies that it should only be used by the device administrator, but Kratos did not find any checks. We further analyze the AOSP source code and confirm that it is a real inconsistency in security enforcement. In the comments, the developer made it very clear that this method should only be called by a device administrator (as shown in left side of Fig. 1). But surprisingly, they did not apply any security checks to secure it. By invoking Power Manager Service's `setMaximumScreenOffTimeoutFromDeviceAdmin()`, an app without the proper permissions can set the screen timeout to a very large value in order to drain the battery. Past studies [45], [16], [35] have shown that display is a major contributor to the battery consumption of smartphone users.

In this case, the inconsistency occurs between two codebases — the original AOSP code and customization code. This case also demonstrates that though customization is often blamed for introducing more security threats, it is also possible that customizations are more secure than AOSP.

Sending raw requests to RIL. We found two system services in the Samsung Galaxy Note 3 that provide very similar telephony-related functionality. These two services are implemented in two classes, `PhoneInterfaceManager` and `PhoneInterfaceManagerExt`. Kratos reports that the app-accessible interface `invokeOemRilRequestRaw()` exposed by PhoneInterfaceManager and another interface `sendRequestRawToRIL()` exposed by PhoneInterfaceManagerExt mirror in functionality. Specifically, they both invoke `Phone.invokeOemRilRequestRaw()` to send raw requests to radio link layer (RIL).

Nevertheless, their security enforcement is different. `invokeOemRilRequestRaw()` checks the `CALL_PHONE` permission, while `sendRequestRawToRIL()` has no security checks. As a result, using `sendRequestRawToRIL()`, an attacker can send arbitrary data to RIL without requesting any permissions. Note that attackers can only control data, but cannot alter the request type. `sendRequestRawToRIL()` restricts request types to `RIL_REQUEST_OEM_HOOK_RAW`. We monitor all RIL requests sent by a test-phone and confirm that this request type is used. We have not managed to craft malicious data to take control of RIL or attack base stations, because of the difficulty involved in reverse engineering the protocol. However, we believe this unprotected service interface can be exploited by sophisticated attackers who have more knowledge of cellular networks, especially the use of `RIL_REQUEST_OEM_HOOK_RAW` request.

Interestingly, our further analysis of AOSP Android 5.0 source code reveals that `invokeOemRilRequestRaw()` is protected by a system permission `MODIFY_PHONE_STATE`, which is higher-privileged than `CALL_PHONE` that this T-Mobile Samsung phone's `invokeOemRilRequestRaw()` enforces. This implies that vendors/carries and Google engineers do have different understanding of how to protect certain sensitive operations.

VI. Discussion and Limitations

False negatives. Similar to previous static analysis work, our approach can miss security enforcement inconsistencies. First, Kratos is not able to deal with implicit control flow transitions, e.g., callbacks. To address this problem, we could implement some principles found in EdgeMiner [15] and FlowDroid [11]. Namely, their implicit control flow and context/flow/lifecycle analysis principles, respectively. While this would help identify another security check present within the Android framework, it doesn't completely solve our false negative problem.

Because we use heuristics to reduce computation time for identifying security enforcement circumventions, there is the potential that a heuristic may rule out a true positive. As with all heuristics, they come at a cost of introducing false negatives or false positives. This very tradeoff is one we work hard to balance; keep the runtime fast and minimize the false positive and false negatives. Moreover, vendors/carriers may introduce new means for enforcing their security policies besides the four we have considered.

Kratos currently does not handle the native code (C/C++) that comprises the lower levels of Android and some small portion of the service code base, leading to false negatives. To address this, additional work is needed to build and analyze a control flow graph at the native layer, which is part of our future work.

Difficulty in verifying violations. Currently, Kratos is unable to automatically verify the presence of a circumvention. To decide if a violation is exploitable, one needs to understand the semantics of the code. In most cases, a review of the actual source code is the only way to ascertain the semantics. One must know what operations are able to interact with untrusted space and also design a feasible way to mount the attack. This is the most time consuming portion of Kratos.

Context-insensitive and path-insensitive analysis. Kratos depends on Spark to build a context-insensitive call graph, which could introduce false negatives or positives. It is understood, through work conducted by Lhoták and Hendren [31] , that a context-insensitive call graph may impact the call graphs accuracy. However, Lhoták and Hendren go on to prove that the improvements context-sensitivity provide to the accuracy are minimal. Through these findings, we justify our optimization for a context-insensitive call graph; the memory and computational overhead of a context-aware call graph analysis does not improve the accuracy enough relative to its costs. Path-insensitivity is not applicable here because we are not interested in how branching affects a call chain. We are only interested in security enforcement circumvention, which does not depend on branching. Thus, we are able to safely ignore utilizing this analysis for our call graph.

Scalability. While we have made every effort to allow Kratos to be scalable at every facet, there is one specific place in which we cannot run a parallelized computation — our use of Spark. Spark has significant limitations in scalability. Because of the reliance on Spark, the "Call Graph Construction" phase is unable to be threaded. Even in spite of this limitation, the IV-B phase completes in minimal time (see results in Table II). Every other aspect of Kratos leverages threading in an effort to reduce run time.

Native System Interfaces. We also observe an interesting case where an app can get the MAC address of a network interface card (NIC) using three different ways, guarded by different security enforcement. The first approach is to call the Connectivity Service's `getLinkProperties()`, which checks `ACCESS_NETWORK_STATE` permission. Second, an app can run the command line tool `/system/bin/netcfg` to obtain a list of available NICs and information, including MAC addresses. This requires the app owns `INTERNET` permission. However, the third approach, reading MAC address directly from the file `/sys/class/net/[nic]/address`, does not require any permissions. This motivates our future improvement of Kratos — to handle inconsistencies across different layers of Android.

VII. Related Work

Security policy violation detection and verification. Quite a few of program analysis and verification tools have been proposed to verify security properties or detect security vulnerabilities caused by policy violation [13], [17], [20], [23], [38]. All of them, except AutoISES [38], require developers or users to provide code-level security policies. AutoISES [38] can automatically infer security specifications with the input

of a security check function list, and leverage inferred specifications to automatically detect security violations. Our tool also supports security policy inference like AutoISES, freeing users from providing specification manually. AutoISES works effectively on Linux kernel and Xen, but compared to Kratos, its design may suffer from several limitations when applied to code bases like Android. First, it assumes that security checks are placed just before accessing sensitive objects, while in Android they are placed earlier. In our design, we solve this by using function calls to represent sensitive operations instead of sensitive objects. Second, AutoISES needs a security check function list as input, while in Android it is difficult to enumerate all possible ones due to the existence of different security check types. In contrast, in Kratos only security policy types are needed as input. Last but not least, AutoISES cannot handle Android-specific features like conflated layers (Java and C/C++) and IPCs.

Android permission check circumvention. Felt et al. [21] propose the notion of permission re-delegation in the generalized contexts applicable for both web and smartphone apps. They identified vulnerabilities in Android applications that can be exploited by malware to access privileged functionality without having corresponding permissions. Wu et al. [43] also study this problem but focus on vendor customized Android. Woodpecker [26] is a tool for systematically detecting capability leaks in stock Android smartphones. Our work differs from them in two major aspects. First, we target Android services instead of applications. Second, we are detecting security enforcement inconsistencies that can lead to direct security policy bypass without the need of another app (e.g., a delegate).

Automated security policy enforcement. To date, a few automated solutions have been proposed to address inconsistencies in security policy enforcement. Ganapathy et al. [22] present a technique for automatic placement of authorization hooks, and apply it to the Linux security modules (LSM) framework. Muthukumaran et al. [34] propose an automated hook placement approach that is motivated by a novel observation that the deliberate choices made by clients for objects from server collections and for processing those objects must all be authorized. These solutions cannot be directly adopted by Android for distributing security enforcement as they require perfect knowledge about the security policies that need to be enforced. In Android, this is unlikely the case as (1) Android has multiple layers and its policies are implemented in different layers using different code, e.g., Java and C/C++, and (2) multiple parties (e.g., vendors, carriers) are involved in the customization and each of them may have their own policies. Although automatic hook placement is promising and we should encourage automatic hook placement, it is still highly desirable to seek solutions like Kratos that can automatically detect security vulnerabilities from existing or legacy source code.

Static analysis tools on Android. There has been significant work in using static analysis techniques combined with call graphs to map the Android framework, understand the permission specification, understand how data is disseminated within the Android framework, and enable the functionality of gleaning information from avenues that are unassuming. PScout [12] uses static analysis tools to enumerate all permission checks within the Android framework. They are able to map all permission usages to their appropriate methods and understand the utility of permission usage within the framework. While PScout identifies permissions, they don't look at paths through the framework which would allow the invocation of the permissions they discover. Static taint analysis tools such as FlowDroid [11], AndroidLeaks [24], DroidSafe [25], and Amandroid [41] work to understand how, why, where, and what data travels through the Android framework as a user uses an app in order to perform privacy leakage detection. They track this data from source to sink and watch how and what uses the data they wish to observe. Their use of static analysis techniques and source/sink flow control is an important concept in our work, but in comparison, we identify how and where security enforcement occurs instead of watching data flow. In our case, our sources are specific entry points in the AIDL services provided by Android and our sinks are the convergence point of two APIs.

VIII. Conclusion

In this work, we propose Kratos, a static analysis tool for systematic discovering inconsistencies in security enforcement which can lead to security enforcement circumvention vulnerabilities within the Android framework. We have demonstrated the effectiveness of our approach by applying our tool to four versions of Android AOSP frameworks as well as two customized Android versions, conservatively uncovering at least 14 highly-exploitable vulnerabilities that can lead to security and privacy breaches such as crashing the entire Android runtime, arbitrarily ending phone calls, and setting up a HTTP proxy with no permissions or only low-privileged permissions. Interestingly, many of these identified inconsistencies are caused by the use of hidden interfaces of system services. Our findings suggest that some potentially promising directions to proactively prevent such security enforcement inconsistencies include reducing service interfaces and restricting the use of Java reflection (for accessing hidden interfaces).

We have shown that security enforcement circumvention is a systemic problem in Android. Our work demonstrates the benefit of an automatic tool to systematically discover anomalies for security enforcement in large codebases such as Android. We expect Kratos to be useful for both Android developers as well as vendors who offer customized Android codebases.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful feedback, as well as Shichang Xu and Ashkan Nikravesh for their proofreading efforts. This material is based upon work supported in part by the National Science Foundation under grants CNS-1345226, CNS-1318306 and CNS-1526455, and by the Office of Naval Research under grant N00014-14-1-0440. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Office of Naval Research.

References

[1] Android Binder. https://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf.

[2] ART and Dalvik. https://source.android.com/devices/tech/dalvik/.

[3] Dextra - A tool for DEX and OAT dumping, decompilation. http://newandroidbook.com/tools/dextra.html.

[4] Kratos Results Website. http://tinyurl.com/kratos15.

[5] Multicast DNS. https://tools.ietf.org/html/rfc6762.

[6] Problem in Making Call Flow Graph from Class or Java files. https://mailman.cs.mcgill.ca/pipermail/soot-list/2014-May/006815.html.

[7] Smali An assembler/disassembler for Android's dex format. https://code.google.com/p/smali/.

[8] Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar.

[9] Using Network Service Discovery. http://developer.android.com/training/connect-devices-wirelessly/nsd.html.

[10] Vulnerability summary CVE-2006-1856. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-1856.

[11] 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 *Proc. of ACM PLDI*, 2014.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. of ACM CCS*, 2012.

[13] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of ACM POPL*, 2002.

[14] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proc. of ACM CCS*, 2010.

[15] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of ISOC NDSS*, 2015.

[16] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. USENIX ATC*, 2010.

[17] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of ACM CCS*, 2002.

[18] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proc. of USENIX Security*, 2014.

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. of ACM MobiSys*, 2011.

[20] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proc. of ACM CCS*, 2002.

[21] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proc. of USENIX Security*, 2011.

[22] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *Proc. of ACM CCS*, 2005.

[23] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proc. of ACM CCS*, 2003.

[24] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proc. of TRUST*, 2012.

[25] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *Proc. of ISOC NDSS*, 2015.

[26] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proc. of ISOC NDSS*, 2012.

[27] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of ACM OOPSLA*, 1997.

[28] L. Hendren. Scaling Java points-to analysis using Spark. In *Proc. of Compiler Construction, 12th International Conference, volume 2622 of LNCS*, 2003.

[29] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy*, 2012.

[30] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[31] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction*, pages 47–64. Springer, 2006.

[32] P. Loscocco. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the USENIX ATC*, 2001.

[33] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, 2014.

[34] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *Proc. of ACM CCS*, 2012.

[35] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. IEEE/ACM MICRO*, 2009.

[36] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Proc. of IEEE Symposium on Security and Privacy*, 2010.

[37] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139, 2001.

[38] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *Proc. of USENIX Security*, 2008.

[39] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, And Tools (2nd Edition)*. Addison Wesley, 2006.

[40] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[41] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proc. of ACM CCS*, 2014.

[42] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proc. of USENIX Security*, 2015.

[43] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proc. of ACM CCS*, 2013.

[44] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proc. of ACM WiSec*, 2012.

[45] L. Zhang, B. Tiwana, R. Dick, and Z. M. Mao. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of ACM CODES+ISSS*, 2010.

[46] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proc. of IEEE Symposium on Security and Privacy*, 2014.