Precise Android API Protection Mapping Derivation and Reasoning

Yousra Aafer
Purdue University
yaafer@purdue.edu

Guanhong Tao
Purdue University
taog@purdue.edu

Jianjun Huang
Renmin University of China
hjj@ruc.edu.cn

Xiangyu Zhang
Purdue University
xyzhang@purdue.edu

Ninghui Li
Purdue University
ninghui@purdue.edu

ABSTRACT

The Android research community has long focused on building an Android API permission specification, which can be leveraged by app developers to determine the optimum set of permissions necessary for a correct and safe execution of their app. However, while prominent existing efforts provide a good approximation of the permission specification, they suffer from a few shortcomings. Dynamic approaches cannot generate complete results, although accurate for the particular execution. In contrast, static approaches provide better coverage, but produce imprecise mappings due to their lack of path-sensitivity. In fact, in light of Android's access control complexity, the approximations hardly abstract the actual co-relations between enforced protections. To address this, we propose to precisely derive Android protection specification in a path-sensitive fashion, using a novel graph abstraction technique. We further showcase how we can apply the generated maps to tackle security issues through logical satisfiability reasoning. Our constructed maps for 4 Android Open Source Project (AOSP) images highlight the significance of our approach, as ~41% of APIs' protections cannot be correctly modeled without our technique.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

Access Control; Permission Model; Android

ACM Reference Format:
Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API Protection Mapping Derivation and Reasoning. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243842

1 INTRODUCTION

One keystone of the Android's access control model is *permissions*, which an app must request to access sensitive resources in the framework. The code that implements a security or privacy relevant API would enforce corresponding permissions. That is, the code would check whether the calling app has the required permissions, and throw an exception if the app does not. When developing an app, developers need to know what permissions are required by the API calls that are used in the app, and request these permissions. Doing so requires an accurate specification of API to permission mapping. However, in light of the framework's codebase size and access control complexity, it is challenging to demystify the permission specification of the Android APIs. This has naturally led developers to make mistakes causing various security problems such as *component hijacking* [25, 37, 39] and *permission over-privilege* [11, 37]. To address these issues, the research community has long focused on developing methods that can create a permission map for the Android framework APIs. This permission map, mostly incomplete in the official Android documentation, can be used by app developers to determine the optimum set of permissions necessary for a correct and safe execution of their app.

Prominent efforts of providing permission maps include *Stowaway* [11], *PScout* [6] and more recently *Axplorer* [7]. *Stowawy* used feedback-directed API fuzzing and unit testing to observe the required permissions of framework API calls. *PScout* statically performed reachability analysis between API calls and permission checks to produce a specification that lists the permissions required by each Android API. *Axplorer* built on top of new insights to address important challenges of statically analyzing the Android framework and retrieved a more precise permission map compared to the previous approaches. The generated permission maps have been a valuable input to address various Android security problems. First, the map can be used to study whether an app follows the principle of least privilege, that is, the app is not requesting more permissions than it actually needs. *Permission over-privilege* can increase the impact of a bug/vulnerability found in the target app. Second, the permission map can be used to detect *component hijacking* where a malicious app gains access to a security-sensitive resource without holding the corresponding permission and thus escalates its privilege.

Clearly, the quality of the generated permission maps can greatly influence the detection results. In fact, while the existing approaches provide effective approximations for API permission specifications, they suffer from a number of shortcomings. Although accurate for a particular execution, dynamic approaches do not achieve complete code coverage, and hence generate incomplete mappings. Besides, this approach involves extensive manual efforts in generating valid inputs. The static approaches do not have the coverage problem, but they have the following limitations.

First, due to their lack of path-sensitive analysis, existing static analyses assume that all permissions ever identified for a particular API are indeed required. However, this assumption is not necessarily correct and may lead to inaccurate mappings. Particularly, the existence of one or more permission enforcements in an API's implementation does not imply they are *unconditionally* required. Often, the permissions checks are disjoint rather than conjoint, i.e., if a specific condition holds, permission A is enforced, otherwise, permission B is enforced.

Furthermore, existing static approaches overlook an important aspect of Android access control. The permission enforcements are often conducted together with other security features: UID checks and User checks. On one hand, UID privilege checks are leveraged to verify if the caller is *privileged enough* or *entitled* to perform an operation without holding a permission. On the other hand, User checks are pertinent to the multi-user feature and aim to enforce privilege separation between multiple users sharing an Android device.

In this work, we propose to derive a precise protection specification for Android APIs. Our proposed solution, called *Arcade*, constructs the map in a path-sensitive fashion. Specifically, it first builds a control flow graph starting from each public Android API. Since the CFG includes information that is not pertinent to access control, it transforms the CFG into an Access-control Flow Graph (AFG) that abstracts all security checks along with the conditions that determine which checks ought to be performed, and in the meantime respects the original control flow. It then processes the constructed AFG to extract the access control conditions enforced by the API and their correlations (e.g., conjunction and disjunction), which can be concisely represented as a first-order logic formula,

We further leverage our generated mapping to tackle the widely studied *overprivilege* and *component hijacking* issues. Since our map is presented in the form of a first-order logic formula, it is not straightforward to use it for this purpose. To this end, we propose to translate the detection problems into a logical reasoning problem. To detect permission overprivilege, we conduct logical satisfiability reasoning to extract the least privileged permission(s) the app needs to hold, given its invoked APIs and related contextual information (e.g., parameter values). We then compare this set with the requested permissions. Similarly, to detect component hijacking, we test whether the enforced protection configuration of a component satisfies our generated map for the invoked APIs.

We have used Arcade to derive protection maps for 4 Android AOSP codebases. A breakdown of the maps reveals that ~41% of APIs apply conditional protection checks, thus highlighting the significance of our approach. Furthermore, we have employed our maps to detect permission overprivilege and component hijacking. Compared with other approaches that rely on the permission maps produced by Axplorer (considered to be best performing), we are able to detect on average 43% more unneeded permissions and reduce false alarms in detecting component hijacking by 11 components on average (per image).

Contributions. We make the following contributions:

- We derive a precise protection specification for Android APIs, using path-sensitive analysis and a novel graph abstraction technique.
- We propose a logical reasoning based solution that leverages our map to detect permission overprivileges and component hijacking.

To allow other researchers to benefit from our work, we publish Arcade's generated protection maps at https://arcade-android.github.io/arcade

2 MOTIVATION

In this section, we explain the limitations of existing static analyses based permission map generation techniques and motivate our method.

2.1 Permissions as a Set Are Imprecise

Existing approaches construct the permission specification for a given framework API through identifying permission checks in the implementation of the API. The extracted set of permissions, which might occur on different program paths, are all treated as required by the API. In other words, all the identified permissions are treated as having an AND relation, which might not reflect the actual requirement. In fact, these permissions are often not a simple conjunction. They may have other relations such as OR, dictated by the various program paths. As a result, an app invoking the analyzed API might not require the full set of extracted permissions.

To illustrate how path-insensitivity could lead to inaccurate permission mapping, consider the following motivating example in Figure 1. The code snippet depicts an extract of the access control implementation of the public API `listen` in the `TelephonyRegistry` service, retrieved from the AOSP codebase (API level 25).

```
 1 public void listen(.., PhoneStateListener listener, int events) {
 2   ...
 3   if ((events & LISTEN_CELL_LOCATION) != 0)
 4     enforceCallingPermission(ACCESS_COARSE_LOCATION);
 5   if ((events & LISTEN_DATA_CONNECTION_STATE) != 0)
 6      if(checkCallingPermission(READ_PRIVILEGED_PHONE_STATE) !=
          PERMISSION_GRANTED)
 7        enforceCallingPermission(READ_PHONE_STATE);
 8   if ((events & LISTEN_PRECISE_CALL_STATE) != 0)
 9      enforceCallingPermission(READ_PRECISE_PHONE_STATE);
10   if ((events & LISTEN_OEM_HOOK_RAW_EVENT) != 0)
11      enforceCallingPermission(READ_PRIVILEGED_PHONE_STATE);
12   if ...
13   // do the actual work ...
14 }
```

Figure 1: Motivating Example for Path-Sensitivity

As illustrated, `listen` allows registering a listener object to receive notifications of changes in desired telephony states, specified by the input *events*. Depending on the supplied *events* value(s), the API enforces different permission checks. An app that wishes to invoke the API should request a subset of the enforced permissions depending on the supplied *events* value. For example, if *events* = `00010000` (where the bit represents `LISTEN_CELL_ LOCATION` state), then the calling app needs the permission `ACCESS_COARSE_ LOCATION` (lines 3-4 in Figure 1). Similarly, if *events* = `10010000`, i.e., it includes the two states (`LISTEN_CELL_LOCATION | LISTEN_DATA_ CONNECTION_STATE`), then two permissions are required.

Further complicating the situation is the conditional permission enforcement applied in lines 5 to 7. If one of the bits in *events* matches `LISTEN_DATA_CONNECTION_STATE`, either permission `READ _PRIVILEGED_PHONE_STATE` or `READ_PHONE _STATE` is sufficient.

As such, the precise permission mapping for the API `listen(.., int events)` is represented by a first-order logic formula shown in Figure 2, where `ACCESS_COARSE` is a shorthand for Android permission `ACCESS_COARSE_LOCATION`, `READ_PRIV` for permission `READ_ PRIVILEGED_PHONE_STATE` and `READ_PHON` for permission `READ_ PHONE_STATE` , `CELL` for the telephony event `LISTEN_CELL_ LOCATION` and `DATA` for the event `LISTEN_DATA_CONNECTION_STATE`. Please note that we only present the permission mapping for lines 3-7 in Figure 1 for simplicity.

$$
\begin{array}{l}
\boxed{\textbf{event = 00010000}}\\
\boxed{\begin{array}{l}\{(events\&CELL \neq 0) \land \neg(events\&DATA \neq 0)\\ \land Perm = ACCESS\_COARSE\}\end{array}}\\
\lor\\
\boxed{\textbf{event = 10000000}}\\
\boxed{\begin{array}{l}\{\neg(events\&CELL \neq 0) \land (events\&DATA \neq 0)\land\\ (Perm = READ\_PRIV \lor Perm = READ\_PHON)\}\end{array}}\\
\lor\\
\boxed{\textbf{event = 10010000}}\\
\boxed{\begin{array}{l}\{(events\&CELL \neq 0) \land (events\&DATA \neq 0)\\ \land Perm = ACCESS\_COARSE\_\land\\ (Perm = READ\_PRIV \lor Perm = READ\_PHON)\}\\ \lor\ldots\end{array}}
\end{array}
$$

Figure 2: First-order Logic Representation of Permission Map for `Listen(..)`; boxes and tags are to facilitate understanding

Now, let us consider the permission map for this API provided by AXPLORER [7] [1]:

`TelephonyRegistry.listen(...)::`
`[ ACCESS_COARSE_LOCATION, READ_PHONE_STATE , READ_PRECISE_PHONE_STATE , READ_PRIVILEGED_PHONE_STATE].`

Basically, AXPLORER's map states that the API requires *all* these permissions, which is imprecise, leading to the following problems:

Wrong Specifications to Developers. The official site for Android developers [35] provides the Android SDK and documentation for app developers. However, its permission related information is incomplete [11]. Under these limitations, developers resort to other efforts devoted to building a more precise permission mapping. However, the lack of accurate mapping (e.g., providing a set of permissions while only a subset is needed) may cause developer confusion. Consequently, in an attempt to make their application work, they will add many unneeded permissions, leading to permission overprivilege.

False Positives in Detecting Component Hijacking. To vet apps for component hijacking, state-of-art approaches first identify sensitive resources reachable by an Android component. Once such resources are found, the solutions compare the corresponding permissions (based on the map) to the security protection enforced at the level of the component's declaration. The detection results can be influenced by the existing protection map. Under this scenario, treating an OR as an AND relationship can trigger false alarms. Consider the following app code snippet:

[1]retrieved from https://github.com/reddr/axplorer/blob/master/ permissions/api-25/framework-map-25.txt

```
public class EmailService extends Service {
  mBinder = new IRemoteService.Stub() {
  // exposed servie method
  public string getUniqueId{
          return mTelephonyManager.getImei();
```

```
<service android:name=".EmailService"
   android:permission="android.permission.READ_PHONE_STATE" >
```

Figure 3: False Alarms in Component Hijacking

The above app component `EmailService` exposes a method `getUniqueId`, which retrieves the device's Imei through invoking the framework API `TelephonyManager.getImei`. Based on a mapping generated without path-sensitive analysis, the above component is not correctly protected, as the mapping wrongly concludes that for `getImei`, two permissions are needed: `READ_PRIVI LEGED_PHONE_STATE` and `READ_PHONE_STATE`. However, a path- sensitive analysis (on the implementation of `getImei`, which is omitted here for brevity) reveals that either one is sufficient, thus, the above component is actually safe. Our experiment reveals that, using the existing permission maps, on average 11 instances of component hijacking are false alarms (per image).

2.2 Additional Access Control Dimension

In addition to (explicit) Android permissions, Android frameworks also have other access control mechanisms, including *UID checks* and *User checks*. Existing permission map generation approaches do not account for these additional checks, which may lead to both false positives and false negatives in security analysis.

```
public void setUidCleartextNetworkPolicy(int uid, int policy) {
   if (Binder.getCallingUid() != uid)
     enforceCallingOrSelfPermission(CONNECTIVITY_INTERNAL);
  // do the actual work ...
}
```

Figure 4: Motivating Example For UID checks

UID Checks. UID checks can affect API permission requirements in two ways. First, some specific UIDs are treated as privileged such that apps with those UIDs can access certain APIs without the need to hold corresponding permissions. Second, when the calling UID is the same as the UID of the process affected by the API call, then certain permission requirements are waived. The above code snippet extracted from the `NetworkManagementService` is an example of the latter case. It denotes a disjunction of two checks: if the caller attempts to invoke `setUidCleartextNetworkPolicy` on a UID that does not match his own, it needs to hold the permission `CONNECTIVITY_INTERNAL`. The permission map by AXPLORER would just indicate that the API needs permission `CONNECTIVITY_ INTERNAL`. An app may invoke the API without the permission when it updates its own network setting, which is completely legitimate. But using AXPLORER's map, a developer will unnecessarily request `CONNECTIVITY_INTERNAL`, thus violating the principle of least privilege. Furthermore, our analysis have identified many APIs that do not require any explicit permissions, but rather just

UID checks. We believe that it is important to provide a protection mapping for these APIs as well, as they may be invoked by app developers and lead to component hijacking if not correctly protected.

User Checks. With the introduction of the Android multi-user feature, Android APIs have been incrementally updated to include *User* checks to separate the functionalities and privileges of multiple users. Under the multi-user scenario, two access control aspects are usually implemented. First, a user should be able to perform operations in her own context. For example, a user can only uninstall her own apps. Second, a user in the background should not be able to affect active users. For instance, a background user cannot turn on/off wifi while another user is logged in.

We believe that physical user check should be considered in the generation of APIs protection mapping. In fact, one of the top conditional access control enforcement pattern is adopted to enforce users separation. Prominently, the permission `INTERACT_ACROSS_USERS` is usually checked in disjunction with a user id check. That is, when the user is attempting to perform an operation for another user, the above permission is enforced.

Existing statically-derived efforts present a permission map as a set of permissions. Using the map in security analysis is straightforward as the analysis only needs to compare the map of the APIs invoked with the permissions requested (for overprivilege detection) or enforced (for component hijacking detection). Considering path sensitivity and the additional dimension of checks requires not only novel techniques to model access control behaviors on the framework side, but also sophisticated analysis on the app side as we can no longer use simple set comparison.

We aim to build a protection specification for Android APIs. We call it *protection map*, to differentiate from the permission map by existing works, indicating it is much more than just permissions. We also aim to develop techniques to use protection maps in security analysis.

3 SYSTEM DESIGN

Figure 5 presents the high level work-flow of our proposed system, Arcade. As depicted, it consists of two modules: a *framework analysis* component and an *application analysis* component.

The *framework analysis* module statically analyzes the Android framework in order to construct a precise and path-sensitive API protection mapping. Specifically, It first identifies the public entry points (or APIs) in the exposed interfaces of the framework system services. Then, for each identified API, it builds the Control Flow Graph (CFG). Depending on the API's code complexity, the CFG could be quite complex. Since the CFG contains a lot of nodes irrelevant to enforcing access control (e.g., nodes performing the actual functionality of the API), Our analysis transforms the CFG to an *Access-Control Flow Graph* (AFG) which preserves the access control logic while respecting the original control flow and abstracting away implementation details irrelevant to access control. Finally, the AFG is processed to produce a succinct representation of the access control conditions enforced by the API, taking the form of a

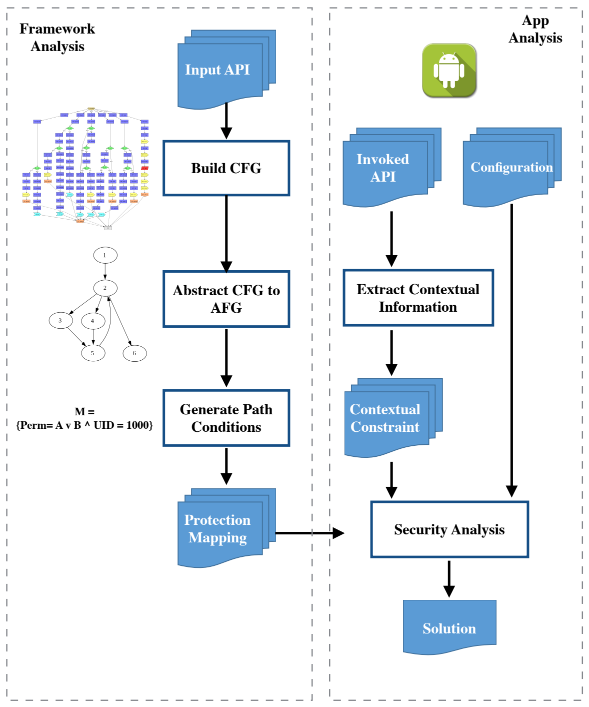


Figure 5: Arcade Design Overview

first-order logic formula on API call arguments and other contextual information. We call the mapping from each API call to such a first-order logic formula the *protection map*.

Our *application analysis* module leverages the constructed protection map to address two well-known security problems: *permission over-privilege* and *component hijacking*. The protection map are represented using first-order logic formulas, which is not straightforward to use. To this end, the application analysis module proceeds as follows:

To tackle permission overprivilege, it first extracts the app's invoked APIs and the corresponding contextual information (e.g. UIDs and arguments values) for each API invocation. Then, based on the API invocation information and our protection map, it conducts boolean satisfiability reasoning to extract the least privileged permission(s) the app needs to hold. Detecting overprivileges is then possible through comparing the app's requested permissions to the generated permissions.

Detecting component hijacking is similarly performed through logical reasoning. The application analysis first extracts the protection configuration a component applies at its declaration site. It then conducts reachability analysis to extract the APIs invoked by it (as well as other contextual information). Last, detecting hijacking vulnerability translates into testing whether the enforced protection configuration satisfies our generated map for the invoked APIs. If not, a malicious app could invoke the component with weak protection to invoke APIs requiring stronger permissions.

4 EXTRACTING FRAMEWORK ACCESS CONTROL

In the following, we explain the details of the individual steps, using the public API `setComponentEnabledSetting` from the `PackageManagerService` as an illustrative example.

4.1 Abstracting CFG to Access-Control Flow Graph (AFG):

Given the CFG for an API function, not all of the nodes in the CFG are of interest in the construction of protection map. For instance, the CFG of `setComponentEnabledSetting` contains a huge number of nodes and edges as its implementation is fairly complicated. To extract the precise protection, we are interested in the instructions that perform security checks, and the instructions that change the values of variables that are used in the security checks. Dependencies of security checks are important because different access control may be enforced depending on different input conditions. Figure 1 in Section 2 depicts such a case: different permissions are enforced depending on the supplied input *events*. We want to point out that simply performing backward program slicing to extract the data/control dependence transitive closure starting from the permission checks results in the inclusion of code unnecessary for building the protection map. For instance, the backward slice would include all code that could affect the control flow before reaching the access control related logic, such as input validation checks, which can be quite complex. Since such code is not part of the access control logic, we need to exclude it through our analysis. Moreover, tracking data dependencies alone is not sufficient because conditionals that determine the different permissions along different paths need to be included.

To precisely capture the access control logic, we introduce the concept of *Access-Control Flow Graph* (AFG), which contains only the instructions and control structures needed for building the protection map. AFG is defined as follows.

Definition 4.1. An Access-Control Flow Graph (AFG) is an abstracted Control Flow Graph. A node in an AFG is a security check instruction or an instruction that is along a program dependence path leading from some API parameter to a security check, precluding those performing input validation. There are also two special nodes *Exception* and *Granted* used to denote the possible exits of an AFG: security exception and access granted. An edge from node n to m is introduced to abstract a control flow path between the two nodes in the program. If n is a conditional statement, the edge also has an annotation (T / F) to distinguish the branch outcome.

Example. Figure 6(A) depicts a code snippet extracted from `setComponentEnabledSetting`'s implementation. This API is for setting the enabled-setting for a package or its component (such as an activity, receiver, etc). The implementation is as follows. Line 2 ensures that the supplied user id exists. Then, lines 5 to 6 verify whether it is dealing with a package level or component level state update. The subsequent lines enforce various security checks. If the caller is trying to update the given component for another user and the caller is not `SYSTEM`, the code enforces the permission `INTERACT_ACROSS_USERS` (lines 7 to 9). Then, the code enforces another layer of access control: if the caller does not own the given component, it enforces the permission `CHANGE_COMPONENT_ENABLED_STATE` (lines 10 to 12). Once the security checks succeed, the actual functionalities of the API are carried out (lines 13 and onward).

Algorithm 1 Constructing AFG.

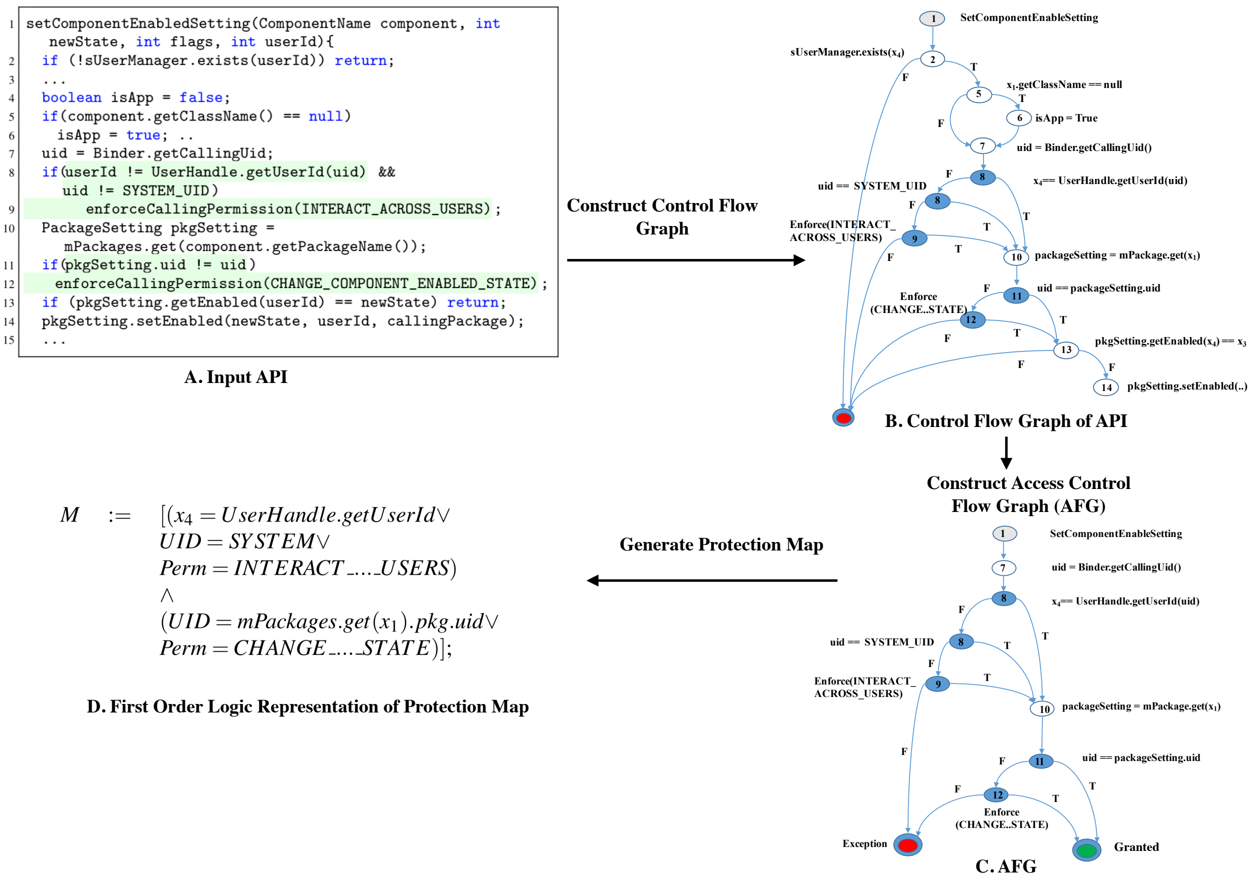
```
Require:
 1: $CFG = (N, E)$ where $N$ is a set of nodes and $E$ is a set of edges.
 2: isSec($n$) = function that checks if a given node $n$ is a security check
 3: isSecRelated($n$) = check if $n$ is a node on which some security check is dependent.
 4: isPred($n$) = check if $n$ is a predicate statement.
 5: pathExists($n$, $m$, $G$, $N$) = check if there exists a path from $n$ to $m$ in $G$ that does
    not include any node in $N$ other than $n$ and $m$.
Ensure:
 6: AFG = ($N'$, $E'$) where $N'$ is node set and $E'$ edge set.

 7: function constructAFG
 8:     $N'$ = {Granted, Exception}
                                                         ▷ Step I: Adding nodes
 9:     for all each node $n \in N$ do
10:         if isSec($n$) then
11:             $N' = N' \cup \{n\}$
12:         if isSecRelated($n$) then
13:             if isPred($n$) and there is a path from $n$ to an exception $m$ along which
    there is no security check then
14:                 continue
15:             $N' = N' \cup \{n\}$
                                                         ▷ Step II: Adding edges
16:     for all each pair $(n, m) \in N'$ do
17:         if pathExists($n$, $m$, $CFG$, $N'$) then
18:             $E' = E' \cup \{(n, m)\}$
                              ▷ Step III: Adding edges to the two special nodes
19:     for all each node $n \in N'$ do
20:         if isPred($n$) and $n$ has less than two edges in AFG then
21:             $E' = E' \cup \{(n, Granted)\}$
22:         if isSec($n$) then
23:             $E' = E' \cup \{(n, Exception)\}$
24:         if isSec($n$) and $n$ has less than two edges then
25:             $E' = E' \cup \{(n, Granted)\}$
```

The actual CFG of the code for `setComponentEnabledSetting` is quite complicated due to its implementation complexity. For simplicity, we construct the CFG for the code snippet (Figure 6(A)). Figure 6(B) depicts the constructed CFG. We label each node with a number indicating the corresponding line number in the code snippet. For instance, node 2 corresponds to the user id input validation. Nodes 7 to 12 represent the access control enforcement logic and may lead to the raise of an exception (the red node on the left) if the enforcement does not hold.

Figure 6(C) depicts our constructed AFG for the same API. It only contains the security enforcement nodes (e.g., nodes 9 and 12) and those providing values to be used in the enforcement (e.g., nodes 7 and 10). The input validation node (node 2) and other nodes which do not affect the security enforcements (nodes 5 and 6) are pruned and abstracted with a single edge (linking node 1 to 7). Observe that although node 2 is in the program slice of the security checks (nodes 9 and 12) as the checks are transitively control dependent on node 2, it is not part of the AFG as there is no data dependency on node 2. Nodes 13 and 14 are also removed and abstracted with an edge to the special node *Granted*. Intuitively, they belong to the situation in which the access is granted. Clearly, the abstracted AFG is much more concise and depicts the access control mechanism. □

Algorithm 1 outlines our process for constructing the AFG from a given CFG. It takes the $CFG(N, E)$ and produces the corresponding AFG denoted by (N', E'). To facilitate discussion, we define a



```
1  setComponentEnabledSetting(ComponentName component, int
      newState, int flags, int userId){
2    if (!sUserManager.exists(userId)) return;
3    ...
4    boolean isApp = false;
5    if(component.getClassName() == null)
6      isApp = true; ..
7    uid = Binder.getCallingUid;
8    if(userId != UserHandle.getUserId(uid) &&
       uid != SYSTEM_UID)
9        enforceCallingPermission(INTERACT_ACROSS_USERS);
10   PackageSetting pkgSetting =
       mPackages.get(component.getPackageName());
11   if(pkgSetting.uid != uid)
12     enforceCallingPermission(CHANGE_COMPONENT_ENABLED_STATE);
13   if (pkgSetting.getEnabled(userId) == newState) return;
14   pkgSetting.setEnabled(newState, userId, callingPackage);
15   ...
```

A. Input API

$$
\begin{aligned}
M \quad := \quad & [(x_4 = UserHandle.getUserId \vee \\
& UID = SYSTEM \vee \\
& Perm = INTERACT\_....\_USERS) \\
& \wedge \\
& (UID = mPackages.get(x_1).pkg.uid \vee \\
& Perm = CHANGE\_....\_STATE)];
\end{aligned}
$$

D. First Order Logic Representation of Protection Map

Construct Control Flow Graph
1 SetComponentEnableSetting
sUserManager.exists(x4)
2
5 x1.getClassName == null
6 isApp = True
7 uid = Binder.getCallingUid()
8 x4== UserHandle.getUserId(uid)
uid == SYSTEM_UID
Enforce(INTERACT_ACROSS_USERS)
9
10 packageSetting = mPackage.get(x1)
11 uid == packageSetting.uid
Enforce (CHANGE..STATE)
12
13 pkgSetting.getEnabled(x4) == x3
14 pkgSetting.setEnabled(..)
T
F
B. Control Flow Graph of API
Construct Access Control Flow Graph (AFG)
Generate Protection Map
Exception
Granted
C. AFG

Figure 6: Proposed Approach for Extracting a Protection Map for an API; Blue Nodes Denote Security Checks

number of auxiliary functions in lines 1-5. The algorithm consists of three steps.

It first initializes the AFG with the two special nodes (line 8). In lines (9-15), it adds nodes to the AFG. Lines 10-11 adds all nodes that perform security check (i.e., UID check, User id check, and permission enforcement). Lines 12-15 add the nodes on which security checks depend. Line 13 excludes input validation checks. Specifically, given a predicate n that some security check directly/-transitively depends on, if there is a path from n to some exception (including return with error code), and there is not any security check along the path. It must be an input validation check. Intuitively, the exception must be caused by something other than security checks (such as line 2 in Figure 6(A)). In contrast, if the path from n to exception has some security check (e.g., lines 8 and 11 in Figure 6(A)), n is access control related and hence added to AFG. Note that nodes on which security checks are data dependent on (e.g., line 7 in Figure 6(A)) are always included.

In the second step, the algorithm adds edges to AFG to abstract the original control flow. Specifically, lines 16-18 connect each pair of nodes in AFG as long as there exists a path in the original CFG and along the path, there are no other AFG nodes. That is, there is reachability between the two and they are the closest. For example, the paths between lines 1 and 7 are abstracted as an edge.

In the third step, the algorithm connects nodes to the two special nodes *Exception* and *Granted* (lines 19-25). Particularly, lines 20-21 add an edge from a predicate n to *Granted* if n currently has only one edge in the AFG. Specifically, the presence of n in the AFG indicates that it must have a path to some security check in the AFG. Therefore, the other (missing) branch of n must correspond to cases in which access is granted. For example, after steps one and two, line 11 in Figure 6(A) is added to the AFG and it has only one edge to line 12. The other branch corresponds to that the access is granted so that the API can proceed with its functionalities. As such, we add an edge from line 11 to *Granted*. Since each security check itself is essentially a predicate, lines 22-25 add edges from security checks to the special nodes.

In our algorithm, statements that perform normal functionalities such as lines 13-14 in Figure 6(A) are completely abstracted away. Also observe that an AFG is a well-formed control flow graph (i.e., each predicate has two edges). Intuitively, one can think of it as a simplified version of the original API code.

Algorithm 2 Generating Protection Map.

```
Require:
1:  $AFG = (N, E)$ with $N$ the node set and $E$ the edge set.
2:  isPred($n$)= checks if $n$ is a predicate.
3:  isSec($n$)= checks if $n$ is a security check.
4:
Ensure:
5:  $p$ = the current path condition
6:  $M$ = the protection map
7:

8:  function DFS($n$, $p$)
9:      if $n$ == Granted then
10:         $M = M \vee p$
11:     if isPred($n$) or isSec($n$) then
12:         for all $\langle n, m \rangle \in E$ do
13:             if $\langle n, m \rangle$ denotes the true branch then
14:                 $p = p \wedge$ {the condition denoted by $n$}
15:             if $\langle n, m \rangle$ denotes the false branch then
16:                 $p = p \wedge \neg$ {the condition denoted by $n$}
17:             DFS($m$, $p$)
18:     else                                          ▷ Must be an assignment
19:         Let $n$ be an assignment $x = e$ with $e$ an expression
20:         $p = (p \wedge x = e)$
21:         Let $m$ be the successor of $n$
22:         DFS($m$, $p$)

    DFS($AFG.root, false$)
    $M$ = LogicalFormulaReduction(($M$))
```

4.2 Generating Protection Map

Given the AFG of an API, its protection map is constructed by extracting the path conditions of all the paths from the entry to the *Granted* node. Specifically, each path denotes a way to acquire the needed access. Hence, the protection map is a first-order logic formula formed by the disjunction of all these path conditions.

Algorithm 2 outlines our process for generating protection map. It takes the AFG and produces a protection map M that is a first-order logic formula.

The algorithm performs a depth first traversal of the AFG starting from the root (i.e. function entry) to node *Granted*. It collects a condition p along each path. The map M is the disjunction of the p of individual paths (lines 9-10). During the traversal of each distinct path, if the node n is a predicate or a security check which is essentially a predicate as well (line 11), the current path condition is conjoined with the condition denoted by n or its negation (lines 13-16). Otherwise, n must be an assignment. In this case, p is conjoined with an assertion denoting the assignment (line 20). Then, the traversal is conducted recursively (lines 17 and 22).

After recursion, M is in the following logical form.

$$M \quad := \quad [p_1 \vee \ldots \vee p_k];$$

where each p_i is represented as follows:

$$p_i \quad := \quad [(y_1 = e_1) \wedge \ldots \wedge (y_n = e_t)]$$

While the above generated logical formula suffices for precise representation of the access control semantics, it might have an arbitrary complicated structure and could be difficult for developers to understand. Thus, we further simplify the logical formula through reducing its number of terms and clauses (function call *LogicalFormulaReduction* in Algorithm 2). This is done using the Quine–McCluskey algorithm [30].

Example. To generate the protection map for the sample API `setComponentEnabledSetting`, we perform a DFS traversal of the AFG in Figure 6(C) starting from the root node. The traversal reveals 6 disjoint unique paths leading to *Granted*. Each path condition is constructed by conjoining the predicates and assignments along the way. We use variable $Perm$ to denote the needed permission(s), UID to denote the enforced UID and variables x_i to denote the ith parameter of the API.

$$\begin{aligned}
M \quad &:= \quad [p_1 \vee ..p_6];\\
p_1 \quad &:= \quad [\neg x_4 = UserHandle.getUserId \wedge\\
&\qquad \neg UID = SYSTEM \wedge\\
&\qquad Perm \ni INTERACT\_..USERS) \wedge\\
&\qquad \neg UID = mPackages.get(x_1).pkg.uid \wedge\\
&\qquad Perm \ni CHANGE\_\ldots\_STATE];\\
p_2 \quad &:= \quad [\neg x_4 = UserHandle.getUserId \wedge\\
&\qquad UID = SYSTEM \wedge\\
&\qquad \neg UID = mPackages.get(x_1).pkg.uid \wedge\\
&\qquad Perm \ni CHANGE\_\ldots\_STATE];\\
&\qquad \ldots
\end{aligned}$$

Observe that the above formula is complicated and verbose. Nonetheless, our reduction step produces a succinct representation as in Figure 6(D).

5 USING PROTECTION MAPPING TO IDENTIFY SECURITY PROBLEMS

The existing Android permission maps have been a valuable input to address classic Android security problems, particularly, *permission over-privilege* and *component hijacking*. Intuitively, since the existing statically-derived efforts [6, 7] present an API's required protection as a single set of conjoint permissions, using such permission sets to identify the aforementioned two problems is quite straightforward. Simply put, to detect overprivilege, they extract the set of permissions an app possesses and compare it with the invoked API's permission map. Similarly, to infer whether an app's component is correctly protected, the solutions compare the enforced component protection with the permission set of its control-flow reachable APIs.

Since Arcade uses first-order logic formulas to precisely model access control mechanisms, using our protection map requires sophisticated analysis of the apps and detecting security issues requires reasoning about satisfiability of logic formulas. In the following subsections, we describe how we analyze apps and use protection maps to detect security problems.

5.1 Over-Privilege Detection

As discussed in the previous section, our constructed protection map for each Android API is a first-order logic formula (as illustrated in Figure 6(D)), which essentially denotes a disjunction of the path conditions leading from the entry point of the API to the *Granted* node in the AFG. In order to invoke a protected Android API, a calling app needs to satisfy at least one of the path conditions. To detect over-privilege problems, we identify the weakest permissions the app needs to possess. We translate the problem into a logic inference problem.

Recall that our protection map formula contains several contextual factors affecting subsequent enforced security protection. That is, different protections might be enforced depending on the contextual condition an app holds. Intuitively, the context of an

API invocation within an app contains (1) *a number of implicit security properties* of the app including the app's UID, User ID, and system settings; these properties cannot be inferred by analyzing the source code of the app, rather, they are *implicitly* encoded in various configuration files; and (2) *the values of API parameters* that can be *explicitly* determined by analyzing the app source code using traditional data flow analysis. Formally, we model context $C = I \wedge P$. I is the *implicit* set of conditions and P the set of *explicit* conditions.

Intuitively, I is a conjunction of individual implicit conditions, such as $i_1 \wedge i_2 \wedge ... \wedge i_n$, where i_t represents an implicit condition. For example, to represent that an app's UID is 1000, we have i_t being (UID = 1000). Similarly, P is a conjunction of individual explicit conditions, such as $e_1 \wedge e_2 \wedge ... \wedge e_n$, where e_i represents the condition for the ith API parameter. For example, if the first parameter may have values 1 and 2, we have e_1 being $x_1 = 1 \vee x_1 = 2$ with x_1 denoting the first API parameter in our protection map. For the cases where we cannot extract the value(s) of an explicit condition (e.g., the parameter value is provided at runtime), we conservatively assume it can be any value.

Example. Let us revisit the example API of `TelephonyRegistry.listen(...)` in Section 2, in which *events* is the 3^{rd} parameter of the API. The system app `GmsCore.apk` (version 6.0-2166767) invokes the API as follows.

```
mTelephony.listen(.., LISTEN_CELL_LOCATION | LISTEN_DATA_CONNECTION_STATE | ...)
```

We extract the explicit condition for parameter *events* (e_3) from the above call site.

$$\begin{aligned} events \;\; &= \;\; LISTEN\_CELL\_LOCATION \\ &\quad \mid LISTEN\_DATA\_CONNECTION\_STATE \mid ... \end{aligned} \quad \square$$

Given the context C for an API at a specific call site and the protection mapping M of the API, we query a constraint solver for the solution of variable $Perm$, which denotes the needed permission(s), in order to satisfy $C \wedge M$. Intuitively, the permission(s) need to satisfy both the contextual condition and the protection map. Since there may be multiple solutions, we enumerate the possible solutions one by one as follows. Assume the first solution is S_1, in order to acquire the second solution, we query $C \wedge M \wedge Perm \neq S_1$. Assume the second solution is S_2, in order to acquire the third solution, we query $C \wedge M \wedge Perm \neq S_1 \wedge Perm \neq S_2$, and so on until it is unsatisfiable (UNSAT). Since the number of solutions is limited, the process quickly terminates.

Example Continued. The permission map M of the previous API is in Figure 2. Inferring a possible permission $Perm$ that `GmsCore` should satisfy to invoke `listen` under the contextual condition C = $\{e_3\}$ entails solving the conjunction of $C \wedge M$. The solver returns two solutions S_1 and S_2.

$$\begin{aligned} S_1 \;\; &:= \;\; Perm = \{ACCESS.., \; READ\_PRIV..\} \\ S_2 \;\; &:= \;\; Perm = \{ACCESS.., \; READ\_PHON..\} \end{aligned} \quad \square$$

Given the multiple solutions, each denoting some permission configuration, we need to identify the weakest one. Note that in Android, different permissions have various privilege levels. According to [2], these permissions can be classified into four categories, whose strength can be ordered as follows. *System = Signature > Dangerous > Normal.* Based on this partial order, we can determine the weakest condition(s) from the multiple solutions returned by the solver.

In our example, the returned solutions S_1 and S_2 correspond to different privilege levels. Since `READ_PRIVILEGED_PHONE_STATE` and `READ_PHONE_ STATE` fall into the *System* and *Dangerous* protection levels, respectively, the first solution S_1 is more privileged than the second one S_2. Hence, requesting S_1 would lead to a permission over-privileged problem.

5.2 Component Hijacking Detection

Classic solutions [25, 37] detect whether a component is correctly protected through comparing its enforced permission at its manifest declaration to the permission sets of its control-flow reachable APIs. If the former is weaker than the latter, a flag is raised. Given our protection map structure, we translate detecting this class of vulnerabilities into a boolean satisfiability problem.

Specifically, we define the same constraints C and M as in the previous section, where C is the contextual condition an app's component holds at its control-flow reachable API and M is our extracted protection map. In addition, we need to define one more constraint D denoting the enforced permission at the component's declaration. Now, detecting whether the target component is not safe is therefore a test of satisfaction of $D \Rightarrow C \wedge M$. Intuitively, we are testing if D is equally strong or stronger than $C \wedge M$.

Permission Normalization. However, the above test may lead to false positives if not carefully designed due to the following app development practice: Any permissions in the *signature* class indicate the same level of protection. As such, if a component is meant to provide a privileged functionality exclusively to other apps signed by the same developer, an arbitrary *signature* level permission may be used, which may not appear in its invoked API protection map. The practice is safe since the target components are actually *overly* protected. However, without encoding the strength of permissions, the solver would return UNSAT, leading to false positives.

To handle such cases, we first classify the permissions in both M and D to the four categories mentioned in the previous section. We further encode the partial order of categories as part of the formula. As such, the solver could correctly compare the different permissions. Details about such normalization are omitted as it is not our contribution.

6 EVALUATION

We implement Arcade on top of Wala [19], which is a comprehensive analysis infrastructure for Java and Dalvik code and can handle large code bases. It has been used in a number of Android analysis projects (e.g., [2, 16–18, 24, 25]). It provides a rich set of analysis primivitves such as alias analysis, dependence analysis, and entry point recogniztion for Android apps. We have also implemented a simplified version of Android IPC resolver similar to [2, 32]. We use Z3 [26] as our solver. For each Android image under study, Arcade extracts and processes its framework class files. As different images might pack the code differently, we employ several existing tools to handle each format gracefully [1, 4, 8, 31].

6.1 Analysis of API Protection Map

Our proposed path-sensitive analysis of the Android framework produced a protection map that correlates APIs with a set of disjoint protection paths dictated by input conditions. Each path is further denoted with a set of conjoint conditions. While the achieved results of our tool ARCADE partially align with a subset of the prior permission maps (e.g., when the protection map has a single permission), they also demonstrate differences for a significant percentage of Android framework APIs (~ 41%), indicating the potential inaccuracy of existing mappings.

In this section, we discuss the characteristics of our constructed API protection mapping and compare the results to the permission specification produced by the most recent effort AXPLORER [7].

6.1.1 *Codebases.*

AOSP codebases. We use ARCADE to extract the protection mapping for 4 Android versions: 6.0 and 6.0.1, 7.0 and the recent release 7.1. Table 1 (rows 2 to 5) summarizes the statistics generated for the analyzed images. As shown, the framework complexity increases between major versions: the number of exposed framework APIs (column 3) is larger in the latest releases.

Custom codebases. Our analysis mainly aims to generate protection specifications for AOSP images, since 3^{rd} party and AOSP system-app developers mostly invoke AOSP's documented APIs. However, we propose to further analyze custom Android images for two reasons. First, we aim to provide new insights about the effect of customization on our generated maps. Second, since vendor apps, accounting for the majority of preloaded apps [37], do invoke custom APIs, we believe that providing protection references for vendor APIs can be quite valuable. Table 1 lists our collected custom images. We select representative images from major vendors: Samsung Galaxy S6 Edge (6.0.1), S8 (7.0), Sony Xperia XZ (7.0) and LG Q6 (7.1). As shown, it is obvious that the vendors conduct heavy customization. The number of exposed APIs drastically increases in the custom images.

Performance. Column 2 in Table 1 reports the time consumed by ARCADE to process and analyze our collected images. As shown, it takes on average 36.5 min to conduct our analysis, with Samsung S7 incurring the longest time (47 min). Since this is a one time effort, the time is acceptable.

6.1.2 *API Protection Mapping Breakdown.*

Table 1 further presents a breakdown of our generated API protection mapping. The 4^{th} column reports the number of APIs where at least one Android protection path has been identified. That is, there is access control in these APIs. Please note that this reported number is slightly larger than what has been reported by AXPLORER; e.g., for AOSP 7.0, AXPLORER reports 1640 protected APIs, while ARCADE reports 1776. This is because ARCADE considers more protection features (UID checks, etc).

The 5^{th} column presents the number of APIs where an *absolute* permission(s) enforcement is detected. That is, it denotes the APIs in which there is only one path leading from the entry to the *Granted* node in the AFGs. There may be multiple security checks along the path, meaning that multiple permissions are required. The following protection maps are examples of *absolute* permission checks: Perm = {MANAGE_FINGERPRINT}, Perm ={ RECEIVE_SMS, SEND_SMS}. Please note that the APIs having *absolute* permission mapping must have the same requested permissions reported in AXPLORER as well, since this latter constructs the permission mapping required for an API as a set of permissions.

The 6^{th} column reports the number of APIs where a permission is *conditionally* enforced (i.e., different security checks are required in different paths leading to *Granted* in AFGs). As such, these APIs should have at least 2 disjoint protection paths. The following protection maps illustrate such cases: Perm = {INTERACT_ACROSS_USERS} $\vee$ x_1 = UserId, Perm = {SCORE_NETWORKS} $\vee$ Perm = {BROADCAST_NETWORK_PRIVILEGED}.

We report similar analysis results for two other protection features: UID and User Id checks. Columns 7 and 8 present the number of APIs where *absolute* and *conditional* UID checks are present, respectively. An example of absolute UID check is UID = SYSTEM_UID, while an example of conditional UID check is UID = SYSTEM_UID $\vee$ Perm = {MANAGE_APP_TOKENS}, meaning that the caller needs to be either SYSTEM or holds the permission MANAGE_... to invoke the API. Columns 9 and 10 depict the results for the User checks. Please note that the conditional UID or UserId checks paths might (partially) overlap with the conditional permission enforcement paths.

Significance of Path-Sensitive Analysis. The last column of Table 1 presents the ratio of detected conditional protection enforcement. As depicted, it constitutes on average ~41% of the enforced checks (for AOSP). Since they do not perform path-sensitive analysis, existing static analysis approaches (e.g., PScout and AXPLORER) would not be able to accurately generate permission maps for this significant portion, rather, their solutions are an approximation. This result clearly highlights the need for our conducted path-sensitive analysis.

Importance of Other Security Features. Although not as substantial as absolute permissions checks, Absolute UID or UserId checks constitute~6.9% of total checks for AOSP. Their conditional enforcements are more substantial (~28%). Hence, to generate an accurate protection map, these features should be considered.

Effect of Customization. Although the sample LG and Sony images have a similar conditional security checks ratio to AOSP, Samsung's protection map demonstrates that vendor customization may affect this number. Samsung exhibits the highest conditional checks ratio: Up to 51%, which further proves the need for path-sensitive analysis in the construction of protection maps.

6.1.3 *Protection Mapping Complexity.*

As discussed, our protection map consists of disjoint protection paths. Intuitively, the count of disjoint paths identified per API (i.e., paths to *Granted* node in AFGs) reflects the complexity level for the adopted access control. The more paths are detected, the more complex the access control is. Figure 7 presents the distribution of protection paths count generated for AOSP 6.0.1 and 7.0.

As illustrated, the protection path count ranges from one single path to as many as more than 8 paths for a few APIs, with one being the most dominant. The next dominant path count is 2 protections paths followed by 4 paths. Observe that there is also an increase in complexity between versions 6.0.1 and 7.0.

6.1.4 *Conditional Protection Characteristics.*

In an effort to provide some insights about the nature of conditional protection enforcements, we report the most common *disjoint* protection paths generated by ARCADE. Recall that *disjoint* protection paths A and B denote the cases where either A or B is satisfied,

Table 1: Breakdown of API Protection Mapping Results

Image	Analysis Time (min)	# Exposed APIs	# Protected APIs	# Permission Checks		# UID Checks		# User Checks		Conditional Checks Ratio (%)
				Absolute	Conditional	Absolute	Conditional	Absolute	Conditional	
AOSP 6.0	27	4117	1510	842	564	68	227	36	180	40.11
AOSP 6.0.1	28	4189	1519	844	570	69	235	36	186	40.31
AOSP 7.0	35	5073	1776	875	648	71	281	49	309	42.55
AOSP 7.1	35	5274	1832	882	680	88	301	49	313	43.53
S6 Edge (6.0.1)	44	9184	2566	1030	1107	292	541	104	629	51.8
S8 (7.0)	47	8616	2743	1091	1172	332	581	115	669	51.79
LG Q6 (7.1)	37	6676	1988	988	779	166	292	55	369	44.09
Sony Xperia XZ (7.0)	38	8383	2022	1059	776	118	377	69	421	42.29

Table 2: Most Common Disjoint Protections Paths

Category	Protection Path 1	Protection Path 2
User Checks	Perm = android.permission.INTERACT_ACROSS_USERS_FULL	x_i = UserId
	Perm = android.permission.INTERACT_ACROSS_USERS_FULL	UID = SYSTEM_UID
	Perm = android.permission.INTERACT_ACROSS_USERS_FULL	UID = ROOT_UID
	Perm = android.permission.INTERACT_ACROSS_USERS_FULL	Perm = android.permission.INTERACT_ACROSS_USERS
	Perm = android.permission.INTERACT_ACROSS_USERS	x_i = UserId
	Perm = android.permission.INTERACT_ACROSS_USERS	UID = SYSTEM_UID
	Perm = android.permission.INTERACT_ACROSS_USERS	UID = ROOT_UID
	UID = SYSTEM_UID	x_i = UserId
	UID = ROOT_UID	x_i = UserId
UID Checks	UID = ROOT_UID	UID = SYSTEM_UID
	Perm = android.permission.UPDATE_DEVICE_STATS	PID = Process.myPid
	Perm = android.permission.MANAGE_APP_TOKENS	PID = Process.myPid
	Perm = android.permission.UPDATE_APP_OPS_STATS	PID = Process.myPid
	Perm = android.permission.SET_KEYBOARD_LAYOUT	PID = Process.myPid
	Perm = android.permission.UPDATE_APP_OPS_STATS	x_i = UID
Permission Checks	Perm = android.permission.READ_PHONE_STATE	Perm = android.permission.READ_PRIVILEGED_PHONE_STATE
	Perm = android.permission.ACCESS_FINE_LOCATION	Perm = android.permission.ACCESS_COARSE_LOCATION
	Perm = android.permission.GET_TASKS	Perm = android.permission.REAL_GET_TASKS
	Perm = android.permission.DEVICE_POWER	Perm = android.permission.UPDATE_DEVICE_STATS
	Perm = android.permission.SCORE_NETWORKS	Perm = android.permission.BROADCAST_NETWORK_PRIVILEGED
	Perm = android.permission.WRITE_SETTINGS	Perm = android.permission.CHANGE_NETWORK_STATE

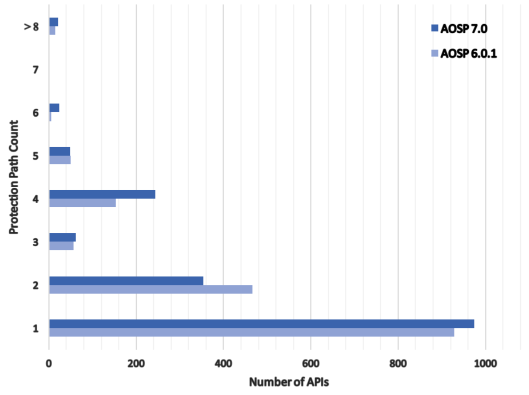


Figure 7: Distribution of Protection Paths Count for AOSP 6.0.1 & 7.0

the access is granted. To this end, we examine the generated disjoint protection paths and group them into possible pairs. if a pair exists in our protection map, we increase the corresponding count. Table 2 lists the most common disjoint paths as generated by our analysis (for AOSP 7.0). Each row denotes a pair.

As shown, we can group the disjoint protection paths into 3 categories. The 1^{st} category denotes a cross user interaction check. This category has been added into the framework to implement the multi-user access control, i.e., enforce that the user has the right to perform certain operations. As listed, user checks consist of either one of the following: verifying if the supplied user id is equivalent to the calling user id (meaning the user is performing a functionality for herself and thus is allowed to do so), holding either one of the permissions `INTERACT_..._FULL`, or `INTERACT_..._USERS` or belonging to a privileged UID (meaning the user is privileged enough to perform operations for another user).

The 2^{nd} category denotes access control where a privileged caller is exceptionally allowed to perform an operation without holding a permission. Such disjoint checks always include a privileged UID / PID check.

The last category includes disjoint permission enforcements, where a permission is required for one path while another is needed for the other path. Our analysis revealed that the condition deciding which permission to enforce is often related to a supplied parameter.

These associated security checks disclose the access control design patterns in Android.

6.2 Applications of Protection Mappings

We leverage the generated protection mapping of Arcade to address permission overprivilege and component hijacking problems. We follow our proposed reasoning of the satisfiability of logic formulas to vet Android apps for these security issues. Following, we describe our collected apps and then present our achieved detection results.

Collected Apps. Our app dataset (Table 3) consists of a large corpus of Android system apps (total 12043 apps) extracted from 62 custom ROMs, which we have collected from various resources

[33, 34] and a few physical devices. The images belong to 12 distinct vendors and operate Android versions 6.0.1 to 7.1.

Table 3: Collected System Apps

Vendor	# Images	# System Apps	Avg # of Apps per Image
Samsung	20	5919	295
LG	1	201	201
HTC	5	781	156
Advan	1	137	137
Kata	3	350	116
Lenovo	11	1505	136
Xiaomi	2	240	120
Mobicel	7	861	123
Oppo	5	1040	208
Sony	1	235	235
Wileyfox	5	633	126
ZTE	1	139	139
Total	62	12043	167

Please note that we focus on system apps because of the following two reasons. First, the usage of System and Signature level permissions, contributing to a significant percentage of the API map, is not relevant in the context of 3^{rd} party apps. Second, any vulnerability detected in system app has a bigger impact as it is automatically present in the corresponding victim images.

6.2.1 *Permission Overprivilege Problem.*

Across all vendors, we identified that 61.5% of apps are overprivileged, with Oppo exhibiting the highest % of overprivileges (~ 74%) and HTC having the lowest (~ 43%). Fortunately, the situation seems to have improved since the overprivilege results (85.78%) reported by [37] for system apps (Android 2.3 to 4.1).

Our conducted app analysis abstracts the context of invoked APIs in an app and selects accordingly the least privileged access control path from its protection mapping, generated by Arcade. This intuitively implies that our set of required permissions per app would be smaller than the ones produced through leveraging PScout's or Axplorer's mappings. To demonstrate whether this indeed leads to capturing more overprivileges instances than other tools, we further perform overprivilege detection using Axplorer's permission mapping. Table 4 reports the average number of *unneeded* permissions per app as produced by our mappings and by the approach relying on Axplorer's.

On average, using Arcade's mapping, we can exclusively detect 2.5 more unneeded permissions per overprivileged app, an average increase of 43.8% over the other approach, which clearly demonstrates the strengths of our conducted path-sensitive analysis. Due to the lack of a ground truth, we manually verified the flagged permissions that were reported as unneeded by our approach. Specifically, we inspected the apps to locate invocation sites to APIs requiring these flagged permissions. Then, for each API, we checked whether the app indeed does not need to hold the permission because of its implicit properties or the specific parameters passed to the APIs. This verification process involves extensive manual work and thus cannot scale to cover a large number of apps. Thus, we performed the manual verification on 120 randomly sampled apps. Our manual analysis revealed that in 112 apps, the flagged permissions are indeed unneeded (92%). For the remaining 8 apps, we could not confirm the result as it was not possible to infer the implicit parameters passed to the APIs.

Furthermore, through our manual inspection, we noticed that the permissions responsible for the observed detection differences

Table 4: Permission Overprivilege across Vendors

Vendor	Avg # of Unneeded Permissions	
	Arcade	Axplorer
Samsung	7.8	5.3
LG	5.5	3.5
HTC	6.5	4.3
Advan	11.8	8.8
Kata	6.6	4.5
Lenovo	9.8	6.3
Oppo	10.4	8.1
Xiaomi	9.8	6.8
Mobicel	7.9	5.1
Sony	8.4	5.9
Wileyfox	6.2	4.2
ZTE	8.5	5.8
Total	8.3	5.7

are unsurprisingly the permissions appearing in Table 2. They were not captured as unneeded by the other approaches, due to their lack of path-sensitive analysis. For example, the permissions `INTERACT_ACROSS_USERS_FULL`, `INTERACT_ACROSS_USERS`, `READ_PRIVILEGED_PHONE_STATE` and `ACCESS_FINE_LOCATION` were all deemed as needed by other approaches, while they were the main reason for the overprivileges detected by our tool.

6.2.2 *Component hijacking Vulnerability.*

Table 5: Component Hijacking

Vendor	Avg % of Vulnerable Apps		Avg #(%) of Additional FP Components by Axplorer[1]
	Arcade	Axplorer	
Samsung	2.7	3.4	7 (9.3 %)
LG	3.8	4.4	9 (14.1 %)
HTC	1.4	1.5	4 (31.2 %)
Advan	2.7	3.4	11 (23.3 %)
Kata	3.9	4.5	12 (15.1 %)
Lenovo	3.8	4.7	11 (14.4 %)
Oppo	4.6	5.8	14 (20.2 %)
Xiaomi	1.8	2.3	8 (22.7 %)
Mobicel	3.1	4	23 (32.3 %)
Sony	1.3	1.7	5 (16.6 %)
Wileyfox	3.1	3.9	18 (18.7 %)
ZTE	2.7	3.5	19 (21.2 %)
Total	2.9	3.4	11.8 (19.8 %)

1. Due to the lack of ground truth, we cannot automatically identify the FP for our tool. Manual inspection on 70 components reported show that 13% are FPs. Note that these FPs are common for both our tool and Axplorer, while the FPs reported in the last column are unique to Axplorer.

Our analysis for component hijacking detection led to the results depicted in Table 5. As reported by our tool (Column 2), the average % of vulnerable apps per vendor ranges from 1.3 % (Sony) to 4.6 % (Oppo). Fortunately, compared to previous reports on component hijacking vulnerabilities [37], the situation seems to be getting better for most vendors (reported 6.77%).

To compare with approaches relying on Axplorer, we further conduct the detection analysis using Axplorer's permission mapping. As depicted in Column 3, the other approach reports a higher % of vulnerable apps. Specifically, while we report 2.9% vulnerabilities, the other approach reports 3.4 %.

To verify whether the reported vulnerabilities by the other approach are false positives, we conducted an additional analysis. We automatically filtered out the app components uniquely flagged as vulnerable by the other approach and investigated the APIs that triggered the hijack-enabling flow from the component's entry point. Unsurprisingly, these APIs had conditional protections in Arcade's generated mappings (such that the component satisfied the strongest protection), which justifies why our analysis did not flag them as vulnerable. We present the average number of components that are additionally flagged as vulnerable by the other approach, per analyzed image. Note that they are all FPs. As shown in the last column, on average 11.8 components are additionally flagged and hence false alarms per image (out of 64 average reported components), reaching up to 23 components (out of 71) in Mobicel and 19 in ZTE (out of 89).

It should be noted that, due to specific static analysis limitations, our implementation of the component hijacking detection (using Arcade's generated map as well as using Axplorer's map) might intrinsically lead to false alarms (common to both of the two approaches). Due to the lack of ground truth, we manually verified 70 randomly sampled components. We identified that 9 cases (i.e., 13%) are false positives. An important thing to note is that they are FPs for both approaches (i.e., using Arcade and Axplorer) while the last column of Table 5 shows the additional FPs produced if relying on Axplorer's mapping. The false positives were due to infeasible code paths that the analysis did not understand. For example, while the analysis discovered a feasible hijacking flow from an entry point to a target sink API, the path contains invocations to native functions that render the path infeasible at runtime. We believe though that our false positive rate is acceptable in a vulnerability filtering scenario.

Although not significant, our approach has also detected some hijacking cases that cannot be detected at all by approaches relying on other mappings. The cause behind this is they did not consider other security features. The following two case studies demonstrate this:

Case Study 1: Denying Bluetooth Discovery for other users. Our analysis reveals a component hijacking vulnerability in several tablet models running versions prior to 7.0, allowing a background user to deny bluetooth discovery and usage for the logged-in users. Specifically, the custom setting app in these devices includes a broadcast receiver `BluetoothDiscoverableTimeoutReceiver` that invokes the API `btservice.AdapterService.setScanMode(..)`, allowing to turn off/on bluetooth discovery. Given this bluetooth related functionality, the receiver is protected with the permission `BLUETOOT_ADMIN`. However, Arcade generates the following protection map for the API: (`UID = SYSTEM` ∨ `UserId = current`) ∧ `Perm = {BLUETOOTH_ADMIN}`. Our tool correctly detects the hijacking vulnerability. Specifically, the above map implies that in addition to the permission, the caller needs to be either `SYSTEM` or belong to an active user, implying that the component is actually weakly protected. In other words, the app should also enforce a UID check, a User check or something equivalent. Otherwise, it could be exploited. We have confirmed the vulnerability by successfully disabling active user's bluetooth using a background user app.

This vulnerability cannot be discovered by approaches relying on other permission maps: Since UID / User Id checks are not considered, the maps consider `BLUETOOTH_ADMIN` as the only protection needed, which is equal to the enforced protection by the component.

Case Study 2: Manipulating Display Colors. In the second case, our analysis discovered a component hijacking problem that cannot be identified by other tools relying on the existing mappings. In a few Samsung devices, we found out that manipulating screen colors (setting it to negative, change color theme, etc) is possible through exploiting a privileged exposed component. Given the privileges of this functionality, the corresponding API `setmDNIeAccessibilityMode` in the `AccessibilityService` is protected with a SYSTEM UID check. However, we found a broadcast receiver component `AccessibilityReceiver` in `SettingsReceiver.apk` that exposes this privileged API without any protection. Our solver consequently returned UNSAT. We have identified other vulnerable components that are caused by this pattern. Prominently, cases allowing to set firewall rules without any privileges in a few Samsung devices. Due to the lack of devices, we could not confirm those.

7 LIMITATIONS

Leveraging Arcade's generated protection map to detect permission overprivilege and component hijacking has an inherent limitation. As our protection map requires understanding of certain contextual factors an app holds at a specific API invocation, failing to infer these conditions will lead to inaccurate detection results. Specifically, due to the nature of static analysis, our conducted detection might not be able to resolve runtime parameters necessary to deduct the exact protection path an app needs to match. For instance, inferring an API's argument corresponding to the current user identifier or to a system-wide setting is not statically possible. Our detection process conservatively assumes such parameter could be anything and thus can lead to false positives. Arcade makes use of existing static analysis primitives such as alias analysis and inter-component communication analysis and hence inherits their limitations.

8 RELATED WORK

Permission Specifications. Stowaway [11] has paved the path for Android permission specification analysis. It extracts the mappings using feedback directed API fuzzing and dynamically logs all permission checks for an API execution. Their mappings are thus accurate but incomplete due to limited code coverage. Compared to our work, each reported permission set by Stowaway for a particular API execution should correspond to one distinct path in Arcade's produced protection mapping.

PScout [6] addresses the code coverage problem of dynamic analysis by statically analyzing the framework and reporting the reachable permission checks from an API. However, their results are conservative: an API may not require the reported permissions in every context. Axplorer [7] produces improved mappings based on a more accurate static analysis of the framework that addresses prominent challenges uniquely characterizing Android. The generated protection mapping of our tool Arcade is similar to that of Axplorer's permission mapping for 60% of the APIs (basically, where an API has one single protection path). However, the rest APIs exhibit different mappings as ours are broken down into disjoint protection enforcement paths.

Analysis of Additional Android Protection Mechanisms. A prominent Android research direction questions the consistency of Android's protection model. Kratos [32] compares the set of security checks in multiple APIs leading to the same resource and reports inconsistent security enforcements. Similar to our work, Kratos also considers other non-traditional security checks; particularly, the UID checks and thread status checks. Some of their reported inconsistencies are due to enforcements containing these checks, which highlights the importance of these features. AceDroid [2] normalizes permissions and security checks (along different paths) to a canonical form that is a tuple of security perspectives, such as *app* and *user*, each perspective having a small set of canonical values with partial order. This enables comparison of multiple protection schemes that have implementation differences. Arcade leverages AceDroid's normalization idea in app analysis. However, our work is different as we focus on generating protection maps that denote the various security enforcements under different contexts. The maps are used to address security problems in the app space whereas AceDroid focuses on detecting inconsistencies within the framework. Our maps can also be used in guiding developers. Furthermore, our technique is based on graph abstraction and logic reasoning.

Vulnerability Detection. Android permission mappings have inspired researchers to identify vulnerabilities at both the framework and application layer. Prominent examples include the re-delegation problem [12, 25], content provider leaks [15], issues in push-cloud messaging [23], in the app uninstallation process [38], crypto misuse in apps [9, 20] and others [3, 10]. In addition, Whyper [28] and AutoCog [29] check the inconsistency between the required permissions and the description of apps. AAPL [24] examines inconsistent behaviors within similar functionalities of similar apps to detect privacy leaks. Our app analysis module aligns well with the works aiming to detect permission re-delegation and permission overprivilege. However, our contribution with this regards lies in our logical reasoning solution aiming to apply our protection map for the detection purpose.

Static analysis on Android. Static analysis techniques have been proposed to address the special characteristics of Android platform. Particularly, FlowDroid [5], DroidSafe [14], AndroidLeaks [13], Amandroid [36] and BidText [17] have employed static taint analysis on Android apps for tracing information flow and detecting privacy leaks. Other tools such as Epicc [27], Didfail [21] and IccTA [22] handle other particular challenges of Android's ICC. Our analysis focuses on the access control aspect and abstracts API implementations to AFGs.

9 CONCLUSION

We propose a novel approach to precisely generate Android API protection specification. Our solution statically analyzes the framework to derive a precise protection specification, using path-sensitive analysis and a novel graph abstraction technique. We further propose a logical reasoning based solution that leverages our maps to detect security issues. Our results demonstrate the strengths of our approach as a significant percentage of our generated specifications cannot be correctly modeled without our proposed analysis.

ACKNOWLEDGEMENTS

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 and 1409668, ONR under contracts N000141410468 and N000141712947, Sandia National Lab under award 1701331, and ARO under 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] Baksmali: a disassembler for Android's dex format. 2017. (2017). Retrieved May 2, 2018 from https://github.com/JesusFreke/smali

[2] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society.

[3] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. 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 (CCS '15).* ACM, New York, NY, USA, 1248–1259. DOI: http://dx.doi.org/10.1145/2810103.2813648

[4] Smali: an assembler for Android's dex format. 2017. (2017). https://github.com/JesusFreke/smali

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* New York, NY, USA, 11. DOI: http://dx.doi.org/10.1145/2594291.2594299

[6] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12).* ACM, New York, NY, USA, 217–228. DOI: http://dx.doi.org/10.1145/2382196.2382222

[7] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *25th USENIX Security Symposium (USENIX Security 16).* USENIX Association, Austin, TX, 1101–1118.

[8] ART compiler. 2017. (2017). https://source.android.com/devices/tech/dalvik/

[9] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS '13).* ACM, New York, NY, USA, 73–84. DOI: http://dx.doi.org/10.1145/2508859.2516693

[10] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. 2013. *Hey, You, Get Off of My Clipboard.* Springer Berlin Heidelberg, Berlin, Heidelberg, 144–161. DOI: http://dx.doi.org/10.1007/978-3-642-39884-1_12

[11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11).* ACM, New York, NY, USA, 12. DOI: http://dx.doi.org/10.1145/2046707.2046779

[12] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Conference on Security (SEC'11).* USENIX Association, Berkeley, CA, USA, 22–22. http://dl.acm.org/citation.cfm?id=2028067.2028089

[13] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST'12).* Springer-Verlag, Berlin, Heidelberg, 291–307. DOI: http://dx.doi.org/10.1007/978-3-642-30921-2_17

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

[15] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS).* http://www.csc.ncsu.edu/faculty/jiang/pubs/NDSS12_WOODPECKER.pdf

[16] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15).* USENIX Association, Berkeley, CA, USA, 977–992. http://dl.acm.org/citation.cfm?id=2831143.2831205

[17] Jianjun Huang, Xiangyu Zhang, and Lin Tan. 2016. 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 (FSE 2016).* ACM, New York, NY, USA, 169–180. DOI: http://dx.doi.org/10.1145/2950290.2950348

[18] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1036–1046. DOI: http://dx.doi.org/10.1145/2568225.2568301

[19] IBM. 2017. WALA: T.J. Watson Libraries for Analysis. http://wala.sourceforge.net. (2017).

[20] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. 2013. Predictability of Android OpenSSL's Pseudo Random Number Generator. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. ACM, New York, NY, USA, 659–668. DOI: http://dx.doi.org/10.1145/2508859.2516706

[21] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. 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 (SOAP '14)*. ACM, New York, NY, USA, 1–6. DOI: http://dx.doi.org/10.1145/2614628.2614633

[22] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *arXiv preprint arXiv:1404.7431* (2014).

[23] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. 2014. 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 (CCS '14)*. ACM, New York, NY, USA.

[24] Kangjie Lu, Zhichun Li, Vasileios P Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. 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)*.

[25] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. ACM, New York, NY, USA, 229–240. DOI: http://dx.doi.org/10.1145/2382196.2382223

[26] Microsoft Research. 2017. Z3 Prover. https://github.com/Z3Prover/z3. (2017).

[27] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 543–558. http://dl.acm.org/citation.cfm?id=2534766.2534813

[28] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 527–542. http://dl.acm.org/citation.cfm?id=2534766.2534812

[29] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1354–1365. DOI: http://dx.doi.org/10.1145/2660267.2660287

[30] Quine, J. 2017. Quine?McCluskey algorithm. https://en.wikipedia.org/wiki/Quine?McCluskey_algorithm. (2017).

[31] sdat2img: Convert sparse Android data image (.dat) into filesystem ext4 image (.img). 2016. (2016). https://github.com/xpirt/sdat2img

[32] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2016. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society.

[33] Android Revolution Mobile Device Technologies. 2017. (2017). Retrieved May 2, 2018 from http://android-revolution-hd.blogspot.com/p/android-revolution-hd-mirror-site-var.html

[34] Samsung Updates. 2017. Samsung Updates: Latest News and Firmware for your Samsung Devices! (2017). http://samsung-updates.com/

[35] Official Android Developer Website. 2018. (2018). https://developer.android.com/index.html

[36] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. 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 (CCS '14)*. ACM, New York, NY, USA, 13. DOI: http://dx.doi.org/10.1145/2660267.2660357

[37] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security (CCS '13)*. ACM, New York, NY, USA, 623–634. DOI: http://dx.doi.org/10.1145/2508859.2516728

[38] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. 2016. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society.

[39] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *In Proceedings of the 20th Annual Symposium on Network and Distributed System Security, NDSS '13*. The Internet Society.