Analysis and Transformations in Support of Android Privacy

Denis Bogdanas, Nicholas Nelson, Danny Dig
Oregon State University, USA
Email: {bogdanad, nelsonni, digd}@oregonstate.edu

***Abstract*—To protect user's privacy and system's integrity, mobile platforms use permission models to control accesses to protected resources such as GPS location, Contacts, etc. The previous major version of Android used a static permission model, which compromised the security and privacy of apps. Android 6 overhauled its permission model to ask permissions at runtime which reduces the risk of permission abuse. However, migrating to the runtime permission model requires significant effort from the app developers.**

In this paper we first present a large-scale formative study to understand how app developers use and migrate to the new permission model. Inspired by these findings, we designed, implemented, and evaluated a tool suite that (i) recommends locations where to insert permission requests and (ii) automatically inserts all the permission-related code. Our empirical evaluations on a diverse corpus of real-world apps show that our tools are highly applicable and accurate.

I. Introduction

To protect user's privacy and system's integrity, mobile platforms use permission models to control accesses to protected resources such as GPS location, Contacts, etc. Prior to Android 6, a user had to choose whether to grant permissions at installation time. He had only two choices: to grant all requested permissions and to install the app, or to reject them and to cancel installation.

This static permission model compromised the privacy and security of apps. Would they have a choice, users would deny one third of app requests to protected resources, citing privacy concerns [1]. Felt et al [2] showed that about one third of apps are over-privileged. This increases the chance that malware can abuse granted permissions [3], [4] to perform several malicious activities: to steal data, access fraudulent sites, allow remote access, listen to calls and read personal text messages and contact information.

To address these issues, Android 6, the most recent version available, overhauled its permission model. Apps now ask permissions as needed at runtime. Users can now make informed decisions whether to grant permissions based on the context in which permissions are used. This reduces the risk of permission abuse.

However, the runtime permission model requires extra effort from the app developers. First, app developers have to identify code locations where they need to insert permission requests. Then they have to insert the permission guard, i.e., code that checks whether a method has the proper permission to execute. In addition, developers have to requests permissions before executing the guarded code. Moreover, developers have to write a callback to handle the permission request result from the user.

In this paper we first present a formative study to understand how app developers use the new permission model. We analyzed a corpus of 1911 apps from f-droid [5], a repository of open-source apps, alternative to Google Play. We also studied in depth how developers migrate apps to the new permission model. We analyzed a corpus of 71 randomly selected open-source apps from GitHub that target Android 6. The formative study answers the following questions:

RQ1: What is the rate of migration to Android 6.0? We found that only 22% of all apps in f-droid were migrated to Android 6 at the moment of writing. Among the most actively developed apps in the last 3 months, only 75% of apps have migrated.

RQ2: Which permissions are most used? We found that developers used 24 kinds of permissions. We found that 33% of apps use Android methods that require permissions.

RQ3: Is there variety among the changes developers make when introducing permissions? We found that the changes vary based on (i) the Android component that encapsulates the permission guard, (ii) whether the developer first applied other enabling changes, and (iii) whether the developer had to drastically change the work-flow of the app.

Inspired by these findings, we designed, implemented, and evaluated a tool suite, DroidPerm, that enables developers migrate apps to the Android 6 permission model. DroidPerm consists of two tools. DP-Detect recommends locations where to insert permission requests. DP-Transform automatically inserts all the permission-related code. To migrate their app, the developers only need select the location of the code that requires permissions, and DP-Transform will automatically change the code. Or they can use DP-Detect to recommend even the location.

Manually finding the location where to insert permission requests is non-trivial for two reasons. The protected resources might be nested deep inside call chains that may extend into third party code. Moreover, Android allows permission requests only within Android UI components. Thus, developers have to decide how to "bubble up" permission requests from the place where the protected access is necessary to the place where a request can be issued to the user.

For these reasons we designed DP-Detect, a static analysis tool that detects locations where permission guards have

to be inserted. DP-DETECT models the Android framework (which is primarily event-driven) for a complete coverage of reachable code. It also constructs precise context-sensitive paths from Android UI components to protected resources.

Manually inserting permission-related code also requires a significant effort. Often, before inserting the permission requesting code, the developers need to carry enabling refactorings such as method and field extraction.

For these reasons we designed DP-TRANSFORM, a program transformation tool for automatic insertion of permission requests. To handle the variations we discovered in the RQ3-part of the formative study, DP-TRANSFORM provides user-customizable templates. The tool determines which template to use based on the context of the transformation.

We empirically evaluated our tools to measure their accuracy and applicability. For DP-DETECT we used a corpus of 26 apps that are already migrated by developers to Android 6. We ran DP-DETECT to recommend locations and evaluated their accuracy through manual code inspection. DP-DETECT achieves a precision of 89% and recall of 93%. Moreover, we also evaluated the effect of context-sensitivity on the tool's precision and we found that context-sensitivity is required for usable results.

For DP-TRANSFORM we used a diverse corpus of 71 apps that are already migrated to Android 6. We rolled back the permission-related code and then used DP-TRANSFORM to reintroduce permissions in the same locations. Then we compare the developers' changes with those carried out by DP-TRANSFORM. We found that DP-TRANSFORM is applicable in 84% of the cases. We found that on average, DP-TRANSFORM saves developers from changing 62 lines of code per permission request. DP-TRANSFORM has an accuracy of 94%.

This paper makes the following contributions:

1) **Problem description:** To the best of our knowledge, we are the first to study the problem of migration from static to dynamic permissions in Android apps.
2) **Formative study:** This paper presents the rate of migration, the diversity of permissions, and, the variety of changes that developers make when adding runtime permissions.
3) **Analysis:** We present DP-DETECT, a static analysis tool for recommending locations that require permission guards.
4) **Transformation:** We present DP-TRANSFORM, an Android Studio plugin for automatic insertion of permission guards.
5) **Empirical evaluation** We measure the accuracy and applicability of DP-DETECT and DP-TRANSFORM using a diverse corpus of real-world apps.

An earlier version of DP-TRANSFORM was demo-ed at the Google I/O'16 developer conference. Our tools, the evaluation corpora, and the detailed empirical results are available at:
HTTP://COPE.EECS.OREGONSTATE.EDU/DROIDPERM

II. BACKGROUND

We first present the Android 6 permission model, then an example of inserting an Android 6 permission request.

A. Android 6 permission model

Starting from Android 6, Google adopted a fine-grained permission model, requiring the apps to ask permissions at runtime. Figure 1 shows a runtime permission request insertion. The code is inspired from a real app, `Speed of Sound` [6]. In this subsection we present the typical permission guard and request code pattern, Fig. 1b. Method `guarded()` invoked on line 18 indirectly calls `requestLocationUpdates` on line 37 on the left side. This second method requires the permission `ACCESS_FINE_LOCATION`. We call such methods requiring permissions *sensitives*. Before calling a sensitive, the application must ensure that the respective permission is granted, by calling the permission guard `checkSelfPermission()` on line 11.

If permission is not yet granted, the app will invoke `requestPermissions()` on line 14 to request the respective permissions from the user. Because this method requires user interaction, it may only be invoked from the UI thread. It will pop up a special screen for asking permissions. Once the user accepts or rejects the request, Android invokes the callback method `onRequestPermissionsResult()`. Developer is expected to override this method to handle permission request results. If permissions were granted, application will usually invoke the same sensitive code, e.g. `guarded()` in our case. Otherwise, it might either show a message to the user explaining why permissions were required, or disable functionality requiring permissions.

Not all permissions require runtime handling. Some of them, most notably access to internet, are deemed safe. They are granted at installation time, following pre-Android-6 permission model. Permissions requiring access to private data, such as CAMERA, CALENDAR or LOCATION, are deemed dangerous and require runtime permission requests.

Permissions can be divided into three categories by their target. We call permissions associated with sensitive methods, as in the example above, *method-based*. There are also permissions linked with URIs that encode access to specific UI components or resources; we call them *URI-based*. One example is CALENDAR. The 3rd category are permissions required to access files on the flash storage that are not part of app cache; we call them *storage-based*. Some permissions, CAMERA among them, can be both method- and URI-based.

B. Permission Request Insertion Example

We now illustrate a typical scenario for inserting the runtime permission requests, the way it would be performed by DP-TRANSFORM.

Fig. 1a shows the code before using runtime permissions. Statement requiring permissions is `service.startTracking` on line 22. However, the execution of lines 22-25 only makes sense together, for this reason they all have to be guarded.

```
1  public class SpeedActivity extends
2      ActionBarActivity {
3    private SoundService service;
4
5
6
7
8    private void onClick(View view){
9      boolean isChecked =
10         ((CheckBox) view).isChecked();
11     ...
12
13
14
15
16
17
18
19
20
21
22     if (isChecked) {
23       service.startTracking();
24       updateStatusState(ACTIVE);
25   }}
26
27
28
29
30
31 }
32
33 public class SoundService extends Service {
34   public void startTracking() {
35     ...
36     locationManager
37       .requestLocationUpdates(...);
38     ...
39 }}
```

(a) before

```
1  public class SpeedActivity extends
2      ActionBarActivity {
3    private SoundService service;
4    private boolean isChecked;
5    private static final int
6        ACCESS_FINE_LOCATION_RCODE = 1;
7
8    private void onClick(View view){
9      isChecked = ((CheckBox) view).isChecked();
10     ...
11     if (checkSelfPermission(this,
12         Manifest.permission.ACCESS_FINE_LOCATION)
13         != PackageManager.PERMISSION_GRANTED) {
14       requestPermissions(this, new String[]{
15           Manifest.permission.ACCESS_FINE_LOCATION},
16           ACCESS_FINE_LOCATION_RCODE);
17     } else {
18       guarded();
19   }}
20
21   private void guarded() {
22     if (isChecked) {
23       service.startTracking();
24       updateStatusState(ACTIVE);
25   }}
26
27   @Override
28   public void onRequestPermissionsResult(
29       int requestCode, String[] permissions,
30       int[] grantResults) {
31     if (requestCode = ACCESS_FINE_LOCATION_RCODE) {
32       if (grantResults.length == 1 &&
33           grantResults[0] ==
34           PackageManager.PERMISSION_GRANTED) {
35         guarded();
36       } else {
37         Toast.makeText(this, "Permission Denied",
38             Toast.LENGTH_LONG).show();
39 }}}}
```

(b) after

Fig. 1: Relevant code inspired by the `Speed of Sound` app. The left-hand side (a) shows the original code, whereas the right-hand side (b) shows the transformed code by DP-TRANSFORM.

Guarded code is expected to be called from 2 locations after transformation - the old location in `onClick` and from `onRequestPermissionsResult`. To avoid duplication, it has to be extracted to a method first. Yet the method needs access to local variable `isChecked` from both invocation contexts. Consequently it cannot be a method parameter. For this reason, before extracting the method we will have to extract variable `isChecked` to a field.

Fig. 1b shows the result of applying both refactorings and inserting permission guard and request code. These enabling refactorings ease the transition to runtime permissions by resolving scoping issues, reducing unnecessary code duplication, and preserving the correct state of in-scope variables.

It is possible to have multiple permission requests in one class. Yet the class can have only one `onRequestPermissionsResult` implementation. Thus, to distinguish between potential multiple requests, each is assigned a unique code stored in a special constant field (line 5). Creating this field is also a part of the transformation.

This example illustrates the complexity of changes that a developer would perform when migrating to Android 6.

III. FORMATIVE STUDY OF ANDROID RUN-TIME PERMISSIONS

To get a deep understanding on how runtime permission guards and sensitives are used in Android apps, we conducted a formative study on popular open-source apps. We ask three research questions.

RQ1: What is the rate of migration to Android 6.0?

Corpus 1. For the first two research questions, we used a corpus of 1911 apps from f-droid [5], an app repository of open-source apps alternative to Google Play. It provides the metadata database of all apps in a convenient XML format. We built a tool to parse these metadata.

We found that only 22% of the total number of apps have migrated to Android 6 at the moment of writing. A possible reason for this is that some apps are not actively developed. Then, we only counted the apps that had a new release in the last 3 months; 75% of them were migrated. This shows that migration to Android 6 is highly sought by developers.

Type	# Instances
`Activity`	157
`Fragment`	81
`Service`	21
`Other`	38

TABLE I: Android components containing permission guards in Corpus-2

RQ2: Which permissions are most used?

Among apps in f-droid corpus, 33% use method permissions, 5% use URI permissions, 45% use storage permissions and 6% use permissions that could be either method-based or URI-based. (App sets are not disjoint, apps using multiple permission categories are counted for each category.)

RQ3: Is there variety among the changes developers make when introducing permissions?

Corpus 2. For this question we used a corpus of 71 randomly selected open-source apps from GitHub, comprising 920K lines of code, that target Android 6.

To find the context of permission guards, we did a manual inspection of the source code of each app in Android Studio. First, we searched for usages of all known permission guard methods. For permission guard we recorded the component type of the top-level class in which the guard is found.

The results are displayed in Table I. The most prevalent place where permissions are guarded is `Activity`, the main UI component of Android. Following is `Fragment`, the second most used UI component. These are the main contexts in which permissions could be requested, thus the numbers are not surprising. Inside a `Service` permissions can be guarded but not requested. This cases are also fairly common. Finally, category `Other` means that class in which permissions were guarded is none of the above. This often happens when developers design custom utility classes responsible for guarding the permissions and reuse them across the app code. Some apps use entire custom libraries specifically for permission guarding.

There are slight differences in the way a permission can be requested in `Activity` and `Fragment`. Also, as we stated, inside `Service` permissions can only be guarded, but not requested. Consequently, a migration tool is most helpful when handling all three contexts, which involves extra complexity.

We also found that 10 out of 71 apps were requesting all the permissions in the main activity, at the application startup. If permissions were not granted, the app would shut down. This essentially means migration to Android 6 was faked, and users got the same "all or nothing" choices as with earlier OS versions. This was a surprising find. We can only suspect that developers avoided the extra effort of properly migrating their apps to the new permission model.

IV. DP-DETECT: THE SENSITIVES ANALYSIS

The purpose of DP-DETECT is to help migrating apps from Android 5 to Android 6 by recommending permission guard insertion points. At high level our analysis consists of three steps. First we generate an entry point for the analyzed app, suitable for static analysis (Subsection IV-A). We then pass it to out of the box tools, to generate a call graph and a context-sensitive points-to analysis. These in turn are used to infer guard insertion points (Subsection IV-B). Additional subsections describe various other aspects of our analysis.

A. Entry Point Generation

Despite the fact that Android apps are developed in Java, static analysis tools developed for Java cannot be applied for Android out of the box. Primarily, because Android apps do not have a single execution entry point. They consist of components instantiated by Android framework, which contain methods invoked by the framework when various user or system-generated events occur. Such framework-invoked methods are known as callbacks. Static analysis tools developed for Android typically use as entry point a generated main method, that instantiates the components and invokes the callbacks.

To generate the entry point we use FlowDroid [7] built on top of Soot framework [8]. More precisely we use a custom algorithm to detect callbacks, then pass the callbacks to FlowDroid to generate the dummy main method. FlowDroid already has an advanced callback detection algorithm. It starts with classes configured in Android configuration files, then iteratively adds to them all the callbacks in the reachable code, based on a predefined set of types hosting callbacks and call graph traversal. This algorithm is more suitable for taint analysis, where it is desirable to link each callback to its parent UI component. Yet our experiments showed that it misses more than half of the callbacks, due to only a subset of callback categories being supported. Most notably, FlowDroid does not support `Fragment`, the second most important UI component in Android after `Activity`.

To be useful for software developers, DP-detect requires analysis of all the callbacks. For this reason, we used a different callback detection method. It traverses all the classes in the app and collects the callback methods, defined as following.

A method `C.f()` is considered callback by DroidPerm if it either:

- Overrides a method from an Android class.
- or:
  - Implements a method from an Android interface `I` and
  - An object of type `C` is passed as argument to a method with a formal parameter of type `I`.

For the purpose of callback detection, Android classes/interfaces are all those located in packages `android.` and `com.google.android`, with the exception of `AsyncTask`. This is the only class for which we want member methods to be analyzed in their invocation context, rather than as independent callbacks.

This seemingly over-inclusive algorithm can have two sources of imprecision. One is labeling as callbacks methods

```
1  class CameraHandlerThread {
2    void startCamera(final int cid) {
3      Handler localHandler
4        = new Handler(this.getLooper());
5      localHandler.post(new Runnable() {
6        public void run() {
7          Camera camera = Camera.open(cid);
8          ...
9        }
10     });
11   }
12 }
13
14 class SomeActivity extends Activity {
15   void someCallback() {
16     handler.post(new Runnable() {...})
17   }
18 }
```

Fig. 2: App code example requiring 1-CFA context sensitivity

that are actually called by the app code and are not managed by the system. We overcome this issue by excluding `AsyncTask` from the list of classes defining callbacks and by excluding some classes from the analysis (Subsection IV-D). No other classes to our knowledge need special treatment. The other source of imprecision might be including in the analysis unused code. We leave this case for future work.

After collecting the list of callbacks, DP-detect passes it to FlowDroid infrastructure to generate the main method.

B. Guard Insertion Point Inference

Achieving high precision in path detection requires context sensitivity, as we will show in Section VI. To achieve this, DP-detect uses the entry point generated in the previous step to produce two static analysis structures. We use SPARK [9] to construct a context-insensitive call graph, and GEOM [10] to construct a 1-CFA context-sensitive points-to analysis. This duality was required to combine the strenghts and to overcome the limitations of the two tools: SPARK is context-insensitive, while GEOM does not generate a call graph[1].

The analysis then traverses the call graph from top to bottom, separately for each callback, and collects all the paths reaching sensitives. These paths are then compiled into a report containing the lines of code directly inside each callback that need to be guarded by permission checks.

DroidPerm uses a predefined, but extensible set of sensitive definitions. It is populated from Android SDK documentation and additional sensitives encountered during DroidPerm evaluation.

To have a context-sensitive traversal, we refine the context-insensitive set of edges produced by SPARK with the context-sensitive points-to data given by GEOM. More precisely, if a method call `a.f()` in the call graph contains edges to two implementations `A.f()` and `B.f()`, but points-to data for `a` equals `{B}`, then we will only traverse the edge to `B.f()`.

[1]GEOM actually refnes the call graph produced by SPARK, but it still remains context-insensitive.

This is especially important to handle Java and Android asynchronous constructs, such as `Thread`, `ExecutorService`, `Handler` and `AsyncTask`. Consider the example in Figure 2. Here the class `CameraHandlerThread` is a simplified version of code from [11]. The method `startCamera()` instantiates an anonymous `Runnable` that opens the camera. The call to `Camera.open()` on line 7 is a sensitive that requires `CAMERA` permission. This runnable is then scheduled to be executed on a separate thread by calling `localHandler.post()` on line 5. If the application has other callback events calling `Handler.post()` with different types of `Runnable`, such as line 16 in the example, then a context-insensitive call graph would reach all these instances of `Runnable` from all sites calling `Handler.post()`. As a result all callback events reaching `handler.post()` will be flagged as requiring `CAMERA` permission. Context-sensitive points-to refinement elliminates this sort of false positives.

C. Analysis classpath crafting

The source code supplied with Android SDK for `android.` and `java.` packages does not contain any implementation. Instead, it contains just the subset of classes visible to the user, with stub, e.g. empty implementations of all methods. This lessens the burden on call graph generation tools, but the resulting call graph is vastly incomplete. In particular, SPARK will not generate any edges for a method call `a.f()` if no allocation site can be inferred for values of `a`. Consequently, if guard insertion inference algorithm above would be used on Android SDK + app code alone, almost no sensitives would be found. All classes that host non-static sensitive methods have their allocation sites in Android Framework.

To overcome this issue we used a crafted Android SDK for the analysis. It retains stub implementations for most classes, but uses full or simplified custom implementation for a small subset. We used full implementation for code inside `java.util`, to fully analyze concurrency and collection classes. In addition we supplied our custom implementation for key Android classes.

For example, to get the last known location, the app should invoke a variation of this code:

```
LocationManager lm
  = activity.getSystemService(
      LocationManager.class);
Location loc
  = lm.getLastKnownLocation(provider);
```

We modeled `getSystemService()` to instantiate and return all known system services.

Another use of custom implementations was to simplify the call flow through asynchronous construct `ExecutorService`. We implemented methods like `execute(runnable)` and `submit(runnable)` to directly call `runnable.run()` rather than go through

a longer list of internal calls. This made 1-CFA context sensitivity enough to properly handle the context. Other asynchronous constructs were modeled similarly.

D. The exclusion list

There are cases when we want to exclude some classes from the analysis. In one case it is needed to avoid duplicate analysis and unfeasible paths. It includes a few classes in `android.support.` which are usually bundled with the app bytecode; thus analysis has access to their full implementation, not just stubs. They internally invoke callback methods on other UI components, for callbacks which are already modeled by our main method generation, thus leading to duplicate analysis of the same callbacks. We also exclude methods overriding `onPermissionRequestResult()`, as it is expected that inside those methods all the required permissions were already granted.

DP-DETECT only infers guard locations for method-based permissions. Detecting guard locations for other permission types (e.g., URI-based or Storage-based) requirea dataflow analysis, we leave this for future work. However, once the guard location is known, the transformation for inserting the permission check/request is similar for all permission types. Thus, DP-TRANSFORM can handle all three categories of permissions.

V. DP-TRANSFORM: PERMISSION INSERTION TRANSFORMATIONS

DP-TRANSFORM is a tool that takes as input a code selection and a permission to be guarded for that code selection. It then inserts permission checks, permission requests, and permission request callback handlers. We first explain the overall workflow of the tool, and then illustrate the code transformations.

A. Transformation Workflow and Preconditions

We have implemented DP-TRANSFORM as a plugin in the Android Studio IDE, which is based upon IntelliJ IDEA IDE [12].

When DP-TRANSFORM is used standalone, the developer selects the line(s) of code in the editor that should be guarded and chooses CONVERT TO ANDROID RUNTIME PERMISSIONS from the refactoring menu. A listbox appears with the list of all known permissions. If the top-level class containing the selection is an UI component, a checkbox will also appear, allowing the developer to insert a permission request. The user has to select the permission, and to check whether he wants a permission request to be inserted. If the checkbox is not checked, only a permission guard is inserted. DP-TRANSFORM first extracts referenced local variables and method parameters into fields if they are declared or referenced outside of the selected code. Then it extracts the selected code to a method if there are more than 2 lines of code in the selection. Finally DP-TRANSFORM inserts the permission guard block around the selected code. If permission request checkbox was checked, the call to `requestPermissions` and implementation of `onRequestPermissionsResult` are also inserted.

Fig. 1a shows a code snippet inspired from an Android app, `Speed of Sound`. If the developer applies our transformation on lines 22 to 25, DP-TRANSFORM will transform the code to Fig. 1b. In a subsequent version of `Speed of Sound`, the developers have done this transformation manually. Their new code is semantically equivalent to DP-TRANSFORM's output.

Android Component Support: The permission guard `checkSelfPermission` has several versions, all of them requiring a `Context` object. Components of type `Activity` and `Service` are derived from `Context`. Inside `Fragment`, context can be produced by calling `getActivity()`.

If the developer wants to insert a permission guard in a class that is neither of the above, he has to provide a `Context` instance using his domain knowledge. DP-TRANSFORM does not support this case.

B. Transformation Process

Extract Local Variables to Fields: If there are any variables or method parameters referred from both the code selected from guarding and from outside, they are extracted into fields. For this, DP-TRANSFORM invokes EXTRACT LOCAL VARIABLE refactoring provided by Intellij. For parameters, they are first assigned to a new local variable, then the variable is extracted into a field.

Extract Method: The second step of the transformation is to extract the selected statements into a new method called `guarded()`. This step is only necessary if the selected statements span more than two lines. The goal here is to elliminate duplication of the guarded code, as it now has to be called from two places: the original context, now guarded, and `onRequestPermissionsResult`. To perform this step DP-TRANSFORM invokes EXTRACT METHOD refactoring from Intellij.

Insert Permission Guard: In the third step DP-TRANSFORM wraps the invocation to `guarded` into an `if` statment that invokes `checkSelfPermission` in the `if` clause. To do this, it uses several transformation templates, one for each Android component supporting permission guards: `Activity`, `Application`, `Fragment`, `Service`, and `View`.

Insert Permission Request & Callback Handler Method: The last step of the transformation is to insert the invocation of `requestPermissions` and the implementation of `onRequestPermissionsResult`. There are two different templates for this step, one for each supported component: `Activity` and `Fragment`.

In addition to provided templates, developer can write his own, in case he wants the transformation to be performed differently. This is useful for example, when using custom utility methods that wrap permission guard/request code.

If there are multiple request permisions in the class, `onRequestPermissionsResult` will handle all of them.

If there is already a `onRequestPermissionsResult` present when this step begins, it will update it to include a new permission request for the present transformation. It supports differentiating between requests through either `if-else` blocks or `switch`.

VI. Evaluation - DP-Detect

In our evaluation we address the following research questions:

RQ1: What is the accuracy of sensitives analysis?

RQ2: What is the effect of 1-CFA points-to analysis on precision?

A. *RQ1: What is the accuracy of sensitives analysis?*

Methodology: To answer this question we selected from our formative study corpus-2 all apps that have either Location or Camera permissions referred in the code. We had to restrict the analysis to these two permissions groups[2] for two reasons. First, these are the most commonly used method-based permissions. Second, there is no complete database of sensitive definitions available. We took as initial reference the sensitives database supplied with Android SDK in XML format. However, it is vastly incomplete. Our early runs of DroidPerm were reporting zero sensitives detected for many apps, despite permission guards being present in the code. For such apps, we were manually searching for sensitive definitions by reading the code documentation, and adding any newly found sensitives to our sensitive definition list. Since this process was time consuming, we restricted the evaluation to two most commonly used permission groups.

The result of filtering for Camera and Location was a list of 32 apps. We then ran DroidPerm on this corpus, allocating 12 GB of heap memory to JVM. Two apps could not be analyzed due to `OutOfMemoryError`. Another 4 had to be left out because they were using URI-based Camera sensitives which DroidPerm does not support. What remained was a final corpus of 26 apps.

To evaluate precision and recall, we counted for each app (a) true positives - the number of correctly detected callbacks requiring permissions, (b) false positives - the number of reported callbacks from which paths to sensitives are unfeasible, and (c) false negatives - the number of missed callbacks that can reach sensitives. Evaluation of each app included two steps: review of logs produced by DroidPerm, and, by need, inspection of the source code in Android Studio.

DroidPerm logs complete paths from every callback to every sensitive it can reach. In addition, for each virtual method invocation in the path, we log (a) the number of edges coming out of that method invocation and (b) points-to values for invocation target variable. If points-to set has more than one value, we manually inspect the code in Android Studio. This pre-screening for points-to imprecision allowed a precise evaluation while avoiding time consuming manual inspection for every single path. If we encounter permission guards linked to a particular callback, we consider all sensitives with matching permissions in that callback to be reachable. However, if the guard seems to be reachable from multiple callbacks, we do not make any further assumptions about the sensitives. Instead, we inspect the calls having multiple points-to values, to make sure the edge selected by DroidPerm is indeed feasible. If the edge is not feasible, we count the callback as false positive.

[2]Location is actually a permission group containing 2 permissions: ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION.

App name	SLOC	Run time (s)	Valid	False Pos	False Neg
Forecastie	2062	25	1	-	-
FreifunkACA	2340	26	1	-	-
GPS2SMS	3004	36	5	-	-
GPSLogger	10860	320	3	-	-
Grelp	3631	97	4	-	-
HetSys	6097	21	2	-	-
MDApp	12728	250	2	-	-
MoneyPit	6044	46	1	-	-
Nethunter	9128	51	1	-	-
Omni-Notes	13563	380	4	2	-
OSMDroid	20422	22	7	-	-
PictureTrack	7067	23	1	-	-
Satstat	8858	47	5	-	-
SpeedOfSound	1774	31	1	-	-
Traccar	1228	5	3	-	-
Tracker	519	23	1	-	-
Vlillechecker	4151	59	1	-	1
Web-Opac-App	32233	450	1	-	-
GpuImage	7808	16	2	-	-
Bitcoin-wallet	18549	2200	2	-	-
Dlib	1582	30	2	5	-
FPlayAndroid	32888	23	1	-	-
Open-Keychain	63476	2550	-	-	2
OpenFoodFacts	6285	520	2	-	-
OTP Auth	1316	110	-	-	1
Vector	25931	280	2	-	-
Total	**303544**	**7600**	**55**	**7**	**4**

Fig. 3: Accuracy evaluation for sensitives analysis.

To help evaluating recall, DroidPerm includes one more feature, *undetected sensitives analysis*. Here the whole app classpath is traversed, without using the call graph. Whenever a call to a sensitive is encountered that is neither in the exclusion list nor reached by the analysis, it is logged as *potentially undetected sensitive*.

We manually inspected each potentially undetected sensitive in Android Studio, by searching recursively all the code that could reach them. If potentially undetected sensitives are reachable from callbacks, we consider those callbacks false negatives.

Results: Figure 3 shows the analysis results for our corpus. DroidPerm found 62 callbacks requiring permissions, of which 7 were determined to be false positives during follow-up inspection. Another 4 callbacks with sensitives were not

detected. This leads to a precision of 89% and recall of 93%.

The corpus of apps cumulates more that 300K source lines of code (SLOC). Analysis time per app varied greatly from a few seconds to 40 minutes. Except 2 apps that took the longest time, analysis time was under 9 minutes, and generally unrelated to the project size. Our metrics did not count the size of the included libraries, which has a bigger contribution to total execution time than the app code.

Whenever we encountered incorrectly detected or undetected callbacks with sensitives, we investigated the reason. In the 2 apps where DP-detect reported false positives, the reason was a long path passing through code of a 3rd party library. Such cases could be filtered out by more precise, n-CFA context sensitivity. Notably, none of the false positives were caused by our overly-optimistic definition of what constitutes a callback (Section IV-A), confirming that it was good enough for DP-detect.

Our undetected sensitives analysis reported unused sensitives for 6 apps. However, follow-up investigation found that only 4 instances in 3 apps can be reached from callbacks. In the remaining cases sensitives were either part of unused features of 3rd party libraries, or, in one case, unused app code. Instances of valid but undetected sensitives were caused by call graph incompleteness. For app Omni-Notes root cause was empty implementation of Android SDK method `Context.findViewById()`, responsible for instantiating UI components of type `View`. A more precise modeling of Android would likely help detecting this sensitive. For other 2 apps, the sensitive call was deep inside a 3rd party library, we could not determine where exactly the missing edge in the call graph was.

Another finding of our evaluation were four apps where developers were unnecessarily guarding code that did not involve sensitives. The method `LocationManager.removeUpdates()` is called to stop receiving location change updates. It requires Location permissions according to both Android Javadoc and xml permission definitions. Yet its logic suggests it might be called after Location permissions were denied or revoked. We investigated how developers guard this sensitive in the source code. Out of 18 apps using Location, 8 had callbacks where `removeUpdates()` was the only sensitive for Location. Out of them only half were either guarding for permission before or catching the possible `SecurityException`. Suspecting a common bug in many apps, we executed one of them, making sure that `removeUpdates()` was called without Location permissions being granted. To our surprise, no security exception was thrown. This method turned out to not be a sensitive. Consequently, in the four apps that *were* guarding for Location before `removeUpdates()`, the guard was unnecessary. It was an unnecessary effort spent by the developers facing incorrect documentation. We reported the issue to Google.

Detailed evaluation results can be found on: [13].

B. *RQ2: What is the effect of 1-CFA points-to refinement on precision?*

Methodology: To answer this question we ran DroidPerm with GEOM points-to refinement disabled and compared the results with those of fully enabled DroidPerm. We'll refer to the two configurations as 0-CFA and 1-CFA.

Results: In 6 out of 26 apps 0-CFA reported additional paths. Since 1-CFA is expected to be decidedly more precise than 0-CFA, and our RQ1 evaluation did not show any indications that we might miss paths due to issues with 1-CFA, we consider all the extra paths to be false positives. The number of additional paths was highly nonuniform, ranged from 1-3 for 3 apps to 40-300 for the other 3. In total, 413 more paths were detected, leading to a precision of 0-CFA across the whole corpus of just 12%. This proves that 1-CFA points-to analysis was a necessary feature to make DroidPerm usable.

VII. Evaluation - DP-Transform

To empirically evaluate whether DP-Transform is useful, we answer the following evaluation questions.

RQ1: Applicability: How applicable are the transformations?

RQ2: Effort: How much programmer effort is saved by DP-Transform?

RQ3: Accuracy: How accurate is DP-Transform when performing a transformation?

A. Experimental Setup

To answer these questions, we apply DP-Transform on the 71 open-source Android projects in Corpus-2 from Sect. III. The apps in this corpus have been migrated to Android 6 and represent a variety of different types of apps, including categories such as *Communication*, *Education*, *Entertainment*, *Finance*, *Shopping*, and *Tools* on Google Play [14].

Using apps already migrated to Android 6 allows us to use permission code introduced by developers as oracle for our transformations. Each app is first reverted to a version immediately prior to Android 6 migration.

During reversion process we found that 5 apps in the corpus do not have version history prior to Android 6. We dropped them from the corpus, reducing it to 66 apps.

Using the version after migrating to Android 6 as an oracle, we applied DP-Transform on the prior version to every location for which the app developers introduced a `checkSelfPermission()` block.

In certain cases in which the commits for particular app were not fine-grained enough to allow for reversing just the introduction of permission checks, we manually reverted the introduction of `checkSelfPermission()`, `onRequestPermissionsResult()`, and any related field, method, or local variable refactorings within the immediate vicinity. These reversions were done based on the developer changes indicated in the version history for each particular app.

App Name	SLOC	Permission Locations	Passed	Conditional Passed	Failed	Lines Modified	Methods Extracted	Local Variables Extracted	Method Parameters Extracted	Guards Inserted	Callback Inserted/ Modified
Open-Keychain	63,476	1	1	0	0	17	1	0	1	1	1
ExoPlayer	46,282	1	1	0	0	153	0	0	0	1	1
SeriesGuide	42,520	3	0	3	0	69	0	0	0	3	1
Andstatus	42,175	2	2	0	0	36	0	0	0	2	0
Xabber	38,031	5	3	2	0	100	1	0	2	5	1
Conversations	36,283	7	0	5	2	858	0	0	0	5	0
FPlayAndroid	32,888	3	0	2	1	22	0	0	0	2	0
Web-Opac-App	32,233	2	1	1	0	73	1	1	0	2	1
RedReader	27,775	1	0	1	0	230	0	1	0	1	0
Kore	26,042	4	1	3	0	85	0	0	0	4	0
... (71 apps total)											
Totals	916,650	144	49	63	32	5,479	25	18	15	113	48

Fig. 4: Applicability & Effort evaluations for transformations.

We recorded several metrics for each transformation. Table 4 shows the result of applying DP-TRANSFORM on the permission locations in our corpus of 66 apps. We simplify the table to show the top ten largest apps, based upon lines of code (SLOC), but provide the full results on [13].

*B. **RQ1: Applicability: How applicable are the transformations?***

***Methodology**:* To measure applicability, we counted how many instances are in a supported Android component and thus DP-TRANSFORM could apply a suitable transformation template. We also analyzed the reasons why DP-TRANSFORM cannot change the remaining instances.

***Results**:* We applied DP-TRANSFORM on 144 locations in the 66 apps. Columns 4, 5, and 6 show the number of instances that passed (112), conditionally passed with minor alterations (63), or failed (32) to be in one of the supported components.

We counted the following cases as conditionally passed: (1) in some cases when permissions are not granted, the developer inserted `return` statement to skip executing the remaining of the method; since this change require domain knowledge about the logic of the code, we have inserted a similar `return` statement. (2) inserting a `Context` into `checkSelfPermission` when the parent class has access to a `Context` in a non-standard way (e.g. parent class accepts a `Context` within its constructor), and (3) permission utility classes/methods were created by developer to handle permissions.

DP-TRANSFORM failed to properly transform 32 permission locations. We examined each location and categorized them as following: (1) the parent class does not implement or derive from any Android components (in this case, DP-TRANSFORM cannot determine which template to apply). (2) multiple permissions are handled within a single `checkSelfPermission` permission guard instance (this is simply engineering work and can be easily added in a future release of our tool). (3) the app uses of 3rd-party permission library (e.g., *EasyPermissions* [15], *Dexter* [16], or *Nammu* [17]) which has a drastically different behavior than the standard one. Since DP-TRANSFORM does not have this domain knowledge, it cannot make use of these libraries.

*C. **RQ2: Effort: How much programmer effort is saved by* DP-TRANSFORM*?***

***Methodology**:* As a proxy for measuring the transformation effort, we recorded the number of method extractions, local variable extractions, parameter reference extractions, check block insertions, callback method insertions, and callback method augmentations that were necessary for both app developers and DP-TRANSFORM to execute. We also counted the number of lines of code changed.

***Results**:* For permission locations that passed or conditionally passed, we include the total number of source lines of code modified during the transformation of permission locations within each app (column 7 in Fig. 4). On average, each transformation changes 61.74 SLOC.

We also record the number of methods extracted (column 8), local variables extracted to fields (column 9), method parameters extracted to a field (column 10), permission checks inserted (column 11), and permission request callback handler methods inserted (column 12). In total, 25 method extractions were required to reduce both the number of variables and the number of lines of code duplicated. In total, 18 local variables had to be extracted to class fields in order maintain access to a single variable across multiple scopes (permission check block and permission request callback handler method). In total, 15 method parameters required creating a new local variable that references the parameter. Then DP-TRANSFORM

extracted it to a class field to maintain access and scope. In total, 113 permission checks had to be inserted to determine whether permission has been granted prior to calling a sensitive, and providing a call to `requestPermissions` if not granted. In total, 48 permission request callback handler methods had to either be inserted or an existing `onRequestPermissionsResult` method needed to be modified to include a new block to handle an additional permission request.

D. *RQ3: Accuracy: How accurate is* DP-TRANSFORM *when performing a transformation?*

Methodology: To verify the accuracy, we manually inspected that code that DP-TRANSFORM transformed code is semantically equivalent to the transformations of the developer oracle. We also executed the transformed code in the *Android Virtual Device* (AVD) included with Android Studio IDE, and through the *Android Support* plugin [18] for IntelliJ IDEA IDE.

Results: We analyzed each of the 112 permission locations that either passed or conditionally passed, and found that 6 locations were not semantically equivalent to the code from the developer oracle. In 3 of the cases, the developer extended functionality and behavior with additional code inside of the block guarded by `checkSelfPermission`. In the other 3 cases, the developer inserted a call to `shouldShowRequestPermissionRationale` to determine whether to display a UI element to the user with rationale for requesting a permission. The extension of functionality and behavior, including the introduction of `shouldShowRequestPermissionRationale`, require a deep domain knowledge and understanding the logic of the app. Therefore, without such domain knowledge, DP-TRANSFORM has an accuracy of 94.64%.

VIII. RELATED WORK

A. Tools for static analysis of permissions

The most comprehensive effort of mining permission specifications to date in PScout [19]. Similarly to DP-DETECT it constructs a call graph and performs reachability traversal. The analyzed code is, unlike DP-DETECT, that of Android framework, and the code it tries to detect is that for Android internal permission requests. We investigated the possibility to use PScout result as permissions database, but chose to use Android SDK metadata in the end. The reasons were that PScout does not support Android 6, and we found the produced specifications to contain a significant number of false positives.

A closely related project to DP-DETECT is revDroid [20], a tool and empirical study that analyzes whether Android 6 apps continue to run without crashing when permissions are revoked. Similar to DP-DETECT it uses Soot and FlowDroid under the hood. Curiously, revDroid uses PScout as sensitives database. However, this tool does not customize entry point generation. Nor does it use context-sensitive points-to refinement. Thus we suspect the call graph they use, and consequently the results, are both less precise and less complete.

B. Android evolution

We based our initial understanding of the landscape of API usage in Android on the work of McDonnell et al. [21], which found that API changes quickly outpace app developers. We found similar trends in our formative study, which confirms the need for tools.

The recent work of Karim et al. [22] aimed to solve a related problem: to locate and determine which permissions sensitive APIs are being called most often by developers. In contrast, we focus on solving the problem of finding sensitive locations and introducing permissions guards.

C. Refactoring tools for Android

Some researchers provided tooling support for the static permissions model found in Android 5 and earlier. Jeon et. al [23] introduced a finer-grained permissions model that enhances the standard permissions model found in Android 5. This work does not handle the dynamic and transitive nature of permissions found in Android 6's runtime permissions model, thus it is different than our current work.

Our group has previously automated several refactorings [24], [25] for helping developers convert synchronous, blocking code, into asynchronous code that improves responsiveness. Our current work has a very different scope and uses different techniques.

IX. CONCLUSIONS

A static permission model that asks users to grant permissions at install time spells trouble. Malware can easily abuse the user-granted permissions to compromises the privacy and security of apps. To solve these problems, Android 6 overhauled its permission model to use runtime permissions. However, app developers have to pay a high maintenance premium for it.

Our large scale formative study of a corpus of real-world apps that developers migrated to the Android revealed surprising findings. First, app developers still do not have a consistent way to insert permissions, resulting in a proliferation of techniques, of which some are even dubious. This points to a great need for tools like our DP-TRANSFORM to transform the code consistently and correctly.

Second, app developers insert permission guards in Android components that were not natively designed for permission requests. This points to a need for tools like our DP-DETECT that recommends locations where users can be interrupted and asked to grant permissions.

Third, the sheer number of third party libraries built just to handle permissions indicates that app developers find the Android 6 API for permissions is not intuitive. This again points to the need for tools like ours. We hope that our paper serves as a call to action for other researchers to work on these important problems that ultimately affect 3 billion end users [26].

ACKNOWLEDGMENTS

We would like to thank Jacob Lewis and George Harder for their help with the empirical evaluation. This research was partially funded through a Google Faculty Research Award and NSF grant CCF-1439957.

REFERENCES

[1] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 499–514. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831175

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[3] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2295136.2295141

[4] J. Sellwood and J. Crampton, "Sleeping android: The danger of dormant permissions," in *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, ser. SPSM '13. New York, NY, USA: ACM, 2013, pp. 55–66. [Online]. Available: http://doi.acm.org/10.1145/2516760.2516774

[5] (2016, Aug) F-droid. [Online]. Available: https://f-droid.org/repository/browse

[6] (2016, Aug) Github: Speed of sound. [Online]. Available: https://github.com/jpeddicord/speedofsound

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[8] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, ser. CASCON '10. Riverton, NJ, USA: IBM Corp., 2010, pp. 214–224. [Online]. Available: http://dx.doi.org/10.1145/1925805.1925818

[9] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 153–169. [Online]. Available: http://dl.acm.org/citation.cfm?id=1765931.1765948

[10] X. Xiao and C. Zhang, "Geometric encoding: Forging the high performance context sensitive points-to analysis for java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 188–198. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001443

[11] (2016, Aug) Github: Mdapp. [Online]. Available: https://github.com/olejon/mdapp

[12] Intellij platform software development kit (sdk). [Online]. Available: http://www.jetbrains.org/intellij/sdk/docs/

[13] (2016, Aug) Droidperm study. [Online]. Available: http://cope.eecs.oregonstate.edu/DroidPerm

[14] (2016, Aug) Google play store. [Online]. Available: https://play.google.com

[15] (2016, Aug) Github: Easypermissions. [Online]. Available: https://github.com/googlesamples/easypermissions

[16] (2016, Aug) Github: Dexter. [Online]. Available: https://github.com/Karumi/Dexter

[17] (2016, Aug) Github: Nammu. [Online]. Available: https://github.com/tajchert/Nammu

[18] (2016, Aug) Android support plugin for intellij idea. [Online]. Available: https://plugins.jetbrains.com/plugin/1792

[19] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[20] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. S. Wang, Z. Qian, and H. Chen, "revdroid: Code analysis of the side effects after dynamic permission revocation of android apps," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 747–758. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897914

[21] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 70–79.

[22] M. Y. Karim, H. Kagdi, and M. D. Penta, "Mining android apps to recommend permissions," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 427–437.

[23] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: Fine-grained permissions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381938

[24] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 224–235. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.50

[25] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 341–352. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635903

[26] S. Matt, "Google io by the numbers: every stat mentioned at the event." [Online]. Available: http://www.techradar.com/us/news/phone-and-communications/mobile-phones/google-io-by-the-numbers-every-stat-mentioned-at-the-event-1321710