PATDroid: Permission-Aware GUI Testing of Android

Anonymous Author(s)

ABSTRACT

Recent introduction of a dynamic permission system in Android, allowing the users to grant and revoke permissions after the installation of an app, has made it harder to properly test apps. Since an app's behavior may change depending on the granted permissions, it needs to be tested under a wide range of permission combinations. At the state-of-the-art, in the absence of any automated tool support, a developer needs to either manually determine the interaction of tests and app permissions, or exhaustively re-execute tests for all possible permission combinations, thereby increasing the time and resources required to test apps. This paper presents an automated approach, called PATDroid, for efficiently testing an Android app while taking the impact of permissions on its behavior into account. PATDroid performs a hybrid program analysis on both an app under test and its test suite to determine which tests should be executed on what permission combinations. Our experimental results show that PATDroid significantly reduces the testing effort, yet achieves comparable code coverage and fault detection capability as exhaustively testing an app under all permission combinations.

1 INTRODUCTION

Access control is one of the key pillars of software security [37]. Many access control models exist for selectively restricting access to a software system's security-sensitive resources and capabilities. Among such models, *permission-based* access control has gained prominenace in recent years, partly due to its wide adoption in several popular platforms [23], including Android.

In Andriod, permissions are granted to apps. The Android runtime environment prevents an app lacking the proper permissions from accessing both sensitive system resources (e.g., sensors) as well as other protected applications. Initially, Android employed a *static* permission system, meaning that the users were prompted to consent to all the permissions requested by an app prior to its installation, and the granted permissions could not be revoked afterwards. To provide the users more control over their device, in 2015, starting with API level 23, Android switched to a *dynamic* permission system, allowing users to change the permissions granted to an app at run-time [3].

The introduction of a dynamic permission system, however, poses an important challenge for testing Android apps. A test executed on an app may pass under one combination of granted permissions, yet fail under a different combination. As recommended by Android's best practices: *"Beginning with Android 6.0 (API level 23), users grant and revoke app permissions at run-time, instead of doing so when they install the app. As a result, you'll have to test your app under a wider range of conditions."* [3].

At the state-of-the-art, properly testing an Android app with respect to its permission-protected behavior entails re-execution of each test on all possible combination of permissions requested by an app, as there are no tools available to assist the developers with determining the interplay between tests and permissions. Such an *exhaustive* approach is time consuming, and often impractical, particularly in the case of regression testing, where the execution of an entire test suite needs to be repeated for an exponential number of permission combinations.

To mitigate this challenge, we have developed PATDroid, short for **P**ermission-**A**ware GUI **T**esting of An**Droid**. The insight guiding our research is that a given test may not interact with all the permissions requested by an app, meaning that some permissions, regardless of whether they are granted or revoked, may not affect the app's behavior under a particular test. By excluding the permissions that do not interact with tests, we can achieve a significant reduction in testing effort, yet achieve a comparable coverage and fault detection capability as exhaustive testing.

PATDroid leverages a hybrid program analysis approach to determine the interactions between an app's GUI tests and its permissions. It first dynamically pinpoints the entry-points of the app exercised by each test case. It then statically examines the parts of code that are reachable from the identified entry points to find the permission-protected code fragments. Afterwards, it statically determines the app inputs (i.e., GUI widgets) that control the execution of permission-protected code fragments. Finally, it statically identifies usages of the app inputs in the test scripts. Employing a sufficiently precise, yet scalable technique, PATDroid is able to effectively determine which tests should be executed under what permission combinations for an app.

Our experiments indicate that PATDroid is able to reduce both number of tests and their execution time by 71% on average, while maintaining a similar coverage as exhaustive execution of tests on all permission combinations. In addition, using PATDroid, we were able to identify several defects in real-world apps, as confirmed by their developers, that can only be exposed under certain permission settings, further demonstrating the usefulness of PATDroid in practice.

The paper makes the following contributions:

- *Theory*: To the best of our knowledge, the first approach that considers the dependencies between a program, its test suite, and access control model for the reduction of testing effort;
- *Tool*: A fully automated environment that realizes the approach for Android programs, and made available publicly [10];
- *Experiments*: Empirical evaluation of the approach on a large number of real-world android apps demonstrating its efficacy.

The remainder of this paper is organized as follows. Section 2 introduces an illustrative example to motivate the research. Section 3 provides an overview of PATDroid, while more details are presented in Sections 4-7. Section 8 presents the experimental evaluation of the research. Finally, the paper outlines the related research and concludes with a discussion of future work.

2 ILLUSTRATIVE EXAMPLE

We use a simplified version of an Android app, called *Suntimes*, to motivate the research and illustrate our approach. Suntimes calculates and displays sunrise, sunset, and twilight times for a particular location. It is developed to target Android version 6. Sample screen shots of this app are captured in Figure 1.

Since the app requires access to GPS data, it asks for *Location* permission once launched for the first time (Figure 1a). If a user grants the *Location* permission, the app periodically calculates and updates sunrise, sunset, and twilight times based on the current user location. Alternatively, the user can update her current location on demand from the option menu (Figure 1b), either by manually providing specific latitude and longitude, or using GPS to obtain location data (Figure 1c). However, *Suntimes* crashes when a user, who has previously denied the requested location permission, tries to update the current location using GPS, as the app at that point is neither granted the required permission (i.e., *Location*) to accomplish this task, nor it asks for it again.

To validate its behavior, *Suntimes* comes with a GUI test suite, a subset of which is shown in Figure 2. In contrast to unit tests, these tests run on a hardware device or emulator and commonly referred to as *instrumented tests* [4]. Regardless of the testing framework (e.g., Espresso [5], Robotium [13]), instrumented tests are compiled and packed as a separate *apk* file and installed together with the apk of the main app. To distinguish these two software artifacts throughout the paper, we call the apk containing the test suit and testing libraries as *Test Harness App (THA)*, and the apk of the main app as *App Under Test (AUT)*.

In the test cases shown in Figure 2, `testSunTimesNavigation` (Test #1) verifies the smooth navigation between different suntimes and dates, `testSettingLocationToUserDefined` (Test #2) validates adding a new user-defined location based on GPS data, and `testExportLocations` (Test #3) ensures the correctness of exporting retrieved location information to storage. Since Android version 6, it is recommended to test an app with various combinations of granted and revoked permissions to ensure correct behavior of the app under different conditions [3]. For example, `testSettingLocationToUserDefined` can reveal the aforementioned crash only when the developer has revoked the *Location* permission before running the test.

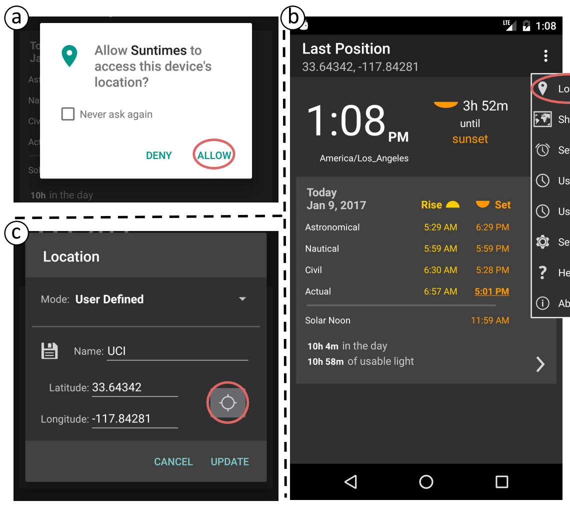

Figure 1: Screenshots of Suntimes app (a) Initially, asking user for the "Location" permission; (b) Main activity with available menu options, where the first option, i.e., location setting, is selected by the user; (c) Adding a new location to the app using GPS data

As another example, consider *Test #3* of Figure 2, which requires *Location* and *Storage* permissions to save user's location. Depending on the permissions granted to *Suntimes*, *Test #3* can exhibit different behaviors:

1) Both required permissions are already granted and *Suntimes* is able to successfully save the user's location on the external storage.
2) Only the *Location* permission is already granted. Hence, *Suntimes* asks for the *Storage* permission. In case of denial, *Suntimes* saves the location information in the app's internal storage, which does not require *Storage* permission.
3) Only the *Storage* permission is already granted. Hence, *Suntimes* asks for the *Location* permission. In case of denial, the app takes no action.
4) Neither of the required permissions have been previously granted. Hence, *Suntimes* asks for both of them. In case of denial, the app takes no action.

In any case, if the user denies any of the requested permissions, *Suntimes* should not crash.

The problem of testing an app's behavior under different permission settings becomes more complicated as the number of permissions defined in the app configuration file, a.k.a. *Manifest*, increases. One approach is to randomly grant and revoke permissions and run the test suite. Though simple, this approach fails to thoroughly test the app's behavior and is prone to miss important defects. Alternatively, a developer could manually review the test scripts and source code of an app to determine which tests should be executed under what app permissions. Such an approach, however, is quite cumbersome, especially considering that every time the app's source code changes, the developer needs to manually establish the relationships between the app's tests and its permissions.

Another approach is to exhaustively run the test suite under all possible combinations of requested permissions. In this approach, if an application requires p permissions, each test should be executed 2^p times, since each permission takes two values of

```java
@Test //Test#1
public void testSunTimesNavigation() {
 onView(withId(R.id.info_note_flipper)).perform(click());
 onView(withId(R.id.info_note_flipper)).perform(click());
 onView(withId(R.id.info_time_nextbtn)).perform(click());
 onView(withId(R.id.info_time_prevbtn)).perform(click());
 // Check the navigation between suntimes is correct ...
}
@Test //Test#2
public void testSettingLocationToUserDefined() {
 onView(withId(R.id.action_location_add)).perform(click());
 onView(withId(R.id.appwidget_location_edit)).perform(click());
 onView(withId(R.id.appwidget_location_getfix)).perform(click())
 onView(withId(R.id.appwidget_location_name)).perform(
        replaceText("My Location"));
 onView(withId(R.id.appwidget_location_save)).perform(click());
 onView(withId(android.R.id.button1)).perform(click());
 // Check the newly added location is shown properly ...
}
@Test //Test#3
public void testExportLocations() {
 openContextualActionModeOverflowMenu();
 onView(withId(R.id.action_settings)).perform(click());
 onData(withKey(configLabel_places)).perform(click());
 onData(withKey(configLabel_places_export)).perform(click());
 // Check the locations are saved correctly ...
}
```

Figure 2: A subset of Espresso [5] tests embedded in the THA to verify the behavior of Suntimes app. The test assertions are not shown here

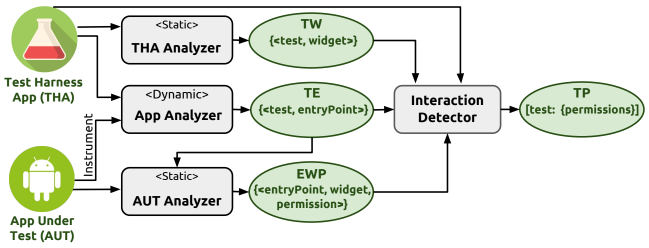

Figure 3: Overview of the approach

$\{granted, revoked\}$.[1] For instance, *Suntimes* requests four permissions in its manifest file (i.e., Location, Storage, Alarm, and Internet). Considering the three tests in Figure 2, exhaustive approach runs each test $2^4 = 16$ times. For only the 3 test shown in Figure 2, we would need a total of $3 \times 16 = 48$ test runs. Clearly, such an approach does not scale as the number of requested permissions and the size of test suite increase.

The insight guiding our research is that exhaustive execution of tests for all permission combinations is overly conservative. For instance, we found that *Test #2* requires only *Location* permission, as the code executed by this test does not require access to capabilities guarded by other permissions. As a result, this test can only be executed twice—with and without the *Location* permission—rather than the 16 times required under the exhaustive scenario.

3 APPROACH OVERVIEW

As mentioned in the previous section, in all popular Android testing frameworks (e.g., [5], [13]), a test suite is compiled and packed to produce the Test Harness App (THA), which is installed together with the App Under Test (AUT). Given a pair of THA and AUT, PATDroid identifies the minimum number of permission combinations for AUT that should be tested for each of the test cases embedded in THA. Figure 3 depicts an overview of PATDroid, consisting of four major components.

PATDroid first identifies those parts of AUT that could be exercised by the test cases embedded in THA. However, this is a challenging task as the test suite and test subject are realized in the form of two separate software artifacts (apk files). Moreover, THA is composed of *instrumented* test cases that require more involved analysis compared to, for example, *unit* tests. In contrast to unit tests that have no Android framework dependencies and directly invoke AUT's methods, instrumented tests run on a hardware device or emulator, and indirectly trigger a sequence of actions via GUI events.[2] The triggered GUI events are handled initially by the testing framework, then Android run-time environment, and eventually delegated to certain methods in AUT, called *entry-points*. Due to such implicit dependency, static analysis cannot resolve the parts of the AUT executed by THA.

To mitigate the difficulties of resolving the relationships between AUT and THA statically, PATDroid leverages a hybrid (static and dynamic) approach that traces the dependencies between AUT and

[1]In Android, only the dangerous permissions are configurable at run-time, while normal permissions are automatically granted at installation time. Without loss of generality, we consider all the permissions can be granted/revoked at run-time. for the evaluation however, we distinguish between dangerous and normal permissions.
[2]Instrumented tests can also trigger other events, such as sending *Intents*. Those events, however, are outside the scope of this research, which focuses on GUI testing.

```
1  @Override
2  public boolean onOptionsItemSelected(MenuItem item){
3    switch (item.getItemId()){
4      case R.id.action_alarm:
5        scheduleAlarm();
6        return true;
7      case R.id.action_settings:
8        showSettings();
9        return true;
10     case R.id.action_location_add:
11       configLocation();
12       return true;
13     case R.id.action_location_refresh:
14       refreshLocation();
15       return false;
16     case R.id.action_timezone:
17       configTimeZone();
18       return true;
19     // other options
20     default:
21       return super.onOptionsItemSelected(item);
22     }
23 }
```

Figure 4: An entry-point of Suntimes app that handles the event corresponding to the selection of menu items shown in Figure 1b (A subset of options are shown here)

THA at two levels of granularity. First, at the method level, dynamic analysis identifies the entry-point methods of AUT that are exercised as a result of running the tests embedded in THA (represented as the set *TE* in Figure 3). Second, at the sub-method level, static analysis components narrow the entry-points discovered by dynamic analysis down to the blocks executable by a particular test case. The selected blocks of the entry-point methods are the targets for further static analysis.

To appreciate the need for restricting the scope of analysis, recall Suntimes app and the test suite shown in Figure 2. The second test (`testSettingLocationToUserDefined`) triggers an event by selecting the Location option from the main menu (line 11), which is eventually handled by an entry-point method shown in Figure 4. This method is among the entry-point methods identified by dynamic analyzer for Test #2. However, inspecting `onOptionsItemSelected` method more carefully, it is clear that only the third case of the *switch* statement (i.e., lines 10-12 in Figure 4) is executable by Test #2, since other cases are intended to handle the other options never triggered by this test. Including the entire method, instead of focusing on lines 10-12, in the search for relevant permissions would increase the false-positive rate of our analysis.

The above example demonstrates that the execution flow of the GUI event handlers is controlled by the widgets triggering those events. Hence, a precise analysis should also take the GUI widgets affecting the control-flow of the app into account, otherwise it would over-approximate the code segments that could be exercised by each test. To that end, *THA Analyzer* determines the widgets used in each test case (represented as the set *TW* in Figure 3), While *AUT Analyzer* determines the permissions needed for executing each block of code in AUT, if any, along with the widgets affecting the reachability of those blocks (represented as the set *EWP* in Figure 3).

Finally, *Interaction Detector* integrates the outputs of the static and dynamic components and generates the relevant permissions for each test case (represented as the map *TP* in Figure 3). The following sections describe the four components of PATDroid in more details.

4 DYNAMIC ANALYSIS

Unlike the conventional Java program with a single `main` method, Android apps comprise several methods that are implicitly called by the framework, usually referred to as *entry-points*. Entry-points are responsible for handling various events, including GUI events (e.g., `onOptionsItemSelected` shown in Figure 4 that handles the selection of a menu option), as well as changing the status of the application, a.k.a. life-cycle events (e.g., `onResume` to activate a paused app).

As a result of running a test, an app's entry-points are invoked by the Android framework. These are identified by the *Dynamic App Analyzer* component. For this purpose, PATDroid first automatically instruments the given AUT and injects *loggers* at the beginning of every possible entry-point of the app, which are distinguishable by the virtue of implementing specific interfaces of the Android framework (e.g., `onOptionsItemSelected`, `onResume`, etc.). For a comprehensive list of Android's entry-point interfaces, we have relied on the results of prior research [19, 38, 39, 49].

PATDroid subsequently runs the entire test suite on the instrumented app with an arbitrary permission setting. Since the invocation of entry-points are independent of the permission settings, our approach effectively finds the THA-dependent entry-points in the AUT. Unlike the test script, the code covered inside the entry-points depends on the permission settings during the test execution. Thus, we use static analysis technique, described in Section 6, to further explore the logic inside the entry-point methods.

Finally, the log obtained through the instrumentation of app's entry-points is processed to capture the executed entry-points for each test case. The generated output of this phase, called *TE*, is a set of tuples $\langle test, entryPoint\rangle$, where the first element is the test identifier and the second element is an exposed entry-point during the test execution. Figure 5 provides a subset of generated output for Test #2 of Figure 2.

5 STATIC ANALYSIS OF TEST HARNESS APP

As briefly discussed in Section 3, PATDroid traces the dependencies between AUT and THA at two levels of granularity. At a high-level of granularity, the dependencies at the method-level are identified by dynamic analysis, as described in the previous section. At a low-level of granularity, within the entry-point methods, the dependencies are refined through static analysis.

To statically trace the dependencies between AUT and THA, PATDroid resolves the app inputs, namely GUI widgets, that are the target of actions performed by test scripts. In the running example, action_location_add is a widget identifier used in both THA and AUT artifacts (lines 11 and 10 in Figures 2 and 4, respectively). For this purpose, PATDroid's static analysis component extracts the widget information from both AUT and THA. The extracted information should uniquely identify the widget throughout the entire app's implementation, and thus, usually includes a widget *identifier* or a *key*. While this section focuses on extracting widgets from THA, Section 6.2 describes how our approach applies to AUT.

Each Android testing framework (e.g., Espresso [5], Robotium [13], etc.) encodes the widget interactions in its own unique way, based on the framework's APIs and patterns. To generalize the problem of finding the used widgets and make our approach test-framework-agnostic, we define this problem as a general *data-flow analysis*. Accordingly, our goal is to find the flow of data within the test programs, from certain *source*s to *sink*s. For this purpose, data sources are defined as the set of testing framework APIs for retrieving a widget by a specific property, e.g., finding widgets based on ID using `ViewMatcher.withId(int)` and `Solo.findViewById(String)` APIs in Espresso and Robotium frameworks, respectively. Similarly, data sinks are defined as the set of testing framework APIs for performing an action on the widgets, e.g., a click action defined by `ViewActions.click()` and `Solo.clickOnButton()` APIs.

Defining the problem in this way allows us to perform the static analysis independent of the testing framework. To support a new testing framework, it is only needed to provide the list of framework's APIs for retrieving and performing actions on widgets. A slightly faster, yet less precise approach to find the widgets is to only look for widget retrieval APIs (i.e., source set only) and simply return the extracted information. This approach, however, can increase the false-positive rate, since some widgets might be retrieved for purposes other than performing an action (e.g., making an assertion). For this reason, we opted for a precise analysis.

To solve the data-flow problem, we employed an Android-compatible data-flow analysis framework, *FlowDroid* [19], but with a significant modification that allows us to perform the analysis on a THA. By default, FlowDroid is intended to analyze apps that comply with the conventional structure expected by the Android framework, e.g., to be composed of Android components. In contrast to AUT, THA does not follow such conventional structure and thus, is not supported by FlowDroid. Therefore, we replaced FlowDroid's default entry-point creator with a customized creator specifically tailored for THA analysis. For each THA, PATDroid creates a dummy `main` method, which is responsible for preparing the test environment encoded in `@Before` methods, and then invoking the `@Test` methods embedded in THA. Recall the use of such annotations in the test script example shown in Figure 2.

Solving the data-flow problem, *THA Analyzer* generates the output, *TW*, which is a set of tuples $\langle test, widget\rangle$, where the first element is the test identifier and the second element is a widget that is the target of an action performed by the test. Figure 6 provides a subset of the analysis output generated for Test #2 of Figure 2.

```
{
<test:testSettingLocationToUserDefined,
 entryPoint:"SuntimesActivity:boolean onOptionsItemSelected(MenuItem)">,
<test:testSettingLocationToUserDefined,
 entryPoint:"LocationConfigDialog$3$1:void onClick(View)">,
<test:testSettingLocationToUserDefined,
 entryPoint:"LocationConfigDialog:void onResume()">,
}
```

Figure 5: A subset of Suntimes app's entry-points exercised by Test #2 of Figure 2

```
{
<test:testSettingLocationToUserDefined,
 widget:action_location_add(2131624168)>,
<test:testSettingLocationToUserDefined,
 widget:appwidget_location_getfix(2131624120)>
}
```

Figure 6: A subset of widgets extracted from Test #2 of Figure 2

Algorithm 1: AUT Analysis

```
Input: AUT: App under test, TE: Tests to entry-points set
Output: EWP: {⟨entryPoint, widget, perm⟩}
1  EWP ← ∅
   // ▶ Permission Analysis - see Section 6.1
2  permSummaries ← PermissionAnalysis(AUT, TE)
   // ▶ Widget Analysis - see Section 6.2
3  widgetSummaries ← WidgetAnalysis(AUT)
4  foreach entryPoint ∈ TE do
5      foreach stmt ∈ entryPoint.statements do
6          if stmt.type is METHOD INVOCATION then
7              perms ← permSummaries[stmt.targetMethod]
8              foreach perm ∈ perms do
9                  widgets ← widgetSummaries[stmt]
10                 if widget = ∅ then
11                     EWP ← EWP ∪ ⟨entryPoint, ∅, perm⟩
12                 else
13                     foreach widget ∈ widgets do
14                         if ⟨entryPoint, widget, perm⟩ ∉ EWP then
15                             EWP ← EWP ∪ ⟨entryPoint, widget, perm⟩
16                     end
17             end
18     end
19 end
```

6 STATIC ANALYSIS OF APP UNDER TEST

Running under an arbitrary permission settings, *Dynamic App Analyzer* partially explores the AUT code executable by each test. Subsequently, PATDroid leverages *AUT Analyzer* to statically examine all parts of the code that could be exercised by each test.

As depicted in Figure 3, the *AUT Analyzer* receives the AUT and *TE* as input and generates *EWP* as output. The generated output is a set of tuples, each containing three elements $\langle entryPoint, widget, permission\rangle$, indicating an entry-point method invoked during the execution of a test, a widget that can affect the reachability of permission-protected code within that entry-point, and the corresponding permission. *AUT Analyzer*'s main procedure is summarized in Algorithm 1.

The analysis procedure performs several steps to generate the output. Initially, *PermissionAnalysis* sub-procedure (line 2) identifies the required permissions for executing each statement, if any, for all of the app's entry-point methods exercised by the test suite. The details of this sub-procedure are described in Section 6.1. Subsequently, *WidgetAnalysis* procedure is invoked in line 3 to determine the statements that are controlled by each widget, the details of which are described in Section 6.2.

For each entry-point method (line 4) and each statement within it (line 5), the algorithm determines whether it is a method invocation statement (line 6) that is permission protected (lines 7–8). These could be either Android API calls or user-defined methods. For each permission-protected method invocation statement, all the widgets that control the execution of this statement are retrieved (line 9). Finally, the algorithm adds tuples consisting of method, widget, and permission information to set EWP, unless they already exist in this set (lines 14–15). If a permission-protected statement is not controlled by any widget, the widget element is set to Null in the corresponding generated tuple (lines 10–11).

Algorithm 2: Permission Analysis

```
Input: AUT: App under test, TE: Tests to entry-points set
Output: PS: Permission Summaries
1  PS ← ∅
   // PS is a map with method signature as its key and the
      corresponding set of required permissions as its value
2  CG ← constructCG(AUT)
3  foreach API ∈ AUT.AndroidAPICalls do
4      method ← caller(API)
5      perm ← perm(API)
6      PS[method] ← perm
7  end
8  repeat
9      foreach method ∈ BFS.next(CG, TE) do
10         callerMethods ← G.edgesTo(method)
11         foreach callerMethod ∈ callerMethods do
12             perms ← PS[method]
13             PS[callerMethod] ← PS[callerMethod] ∪ perms
14         end
15     end
16 until PS reaches a fixed-point;
```

6.1 Permission Analysis

For each method defined in a given AUT, *Permission Analysis* procedure captures all permissions required for executing that method, called *Permission Summaries (PS)*, through performing an *inter-procedural fixed-point* analysis, summarized in Algorithm 2.

In the first step, *Permission Analysis* constructs a call graph (CG) of the entire application (line 2). However, due to the event-driven structure of the Android platform, the traditional CG generation methods do not connect the call sites corresponding to implicit invocations. The challenges of generating call graph for Android apps are widely discussed in the prior research and several techniques are suggested for this purpose [19, 45], which are employed by PATDroid. Figure 7 depicts a subset of the call graph for the Suntimes app. In this graph, the implicit calls are denoted by dashed lines. For instance, the method `GetFixHelper.getFix()` starts an `AsyncTask`, namely `GetFixTask`, by invoking the `execute()` interface. Consequently, the method `doInBackground()` of the task class is invoked indirectly by the Android framework.

Permission Analysis iterates over all Android framework APIs that are called throughout the given app (lines 3–7) and adds the required permission for the API to the permission summaries (*PS*) of the methods where that API is called.[3] We have relied on permission-API mappings produced in the prior work [20, 22] to determine the required permission for Android APIs.

Finally, Permission Analysis traverses the constructed call graph (*CG*) using breadth-first search (BFS) method (lines 9–15). Starting from the given entry-point methods (*EE*), it propagates the permission in the graph. In each iteration, the algorithm updates the permission summaries (PS) of all methods calling the current method, by augmenting their PS with the PS of the callee method (line 13). This procedure is repeated until a fixed point is reached

[3]In addition to the Android framework APIs, certain *Intents* and queris on *Content Providers* need specific permissions. For brevity, however, only the iteration over the APIs is shown in Algorithm 2.

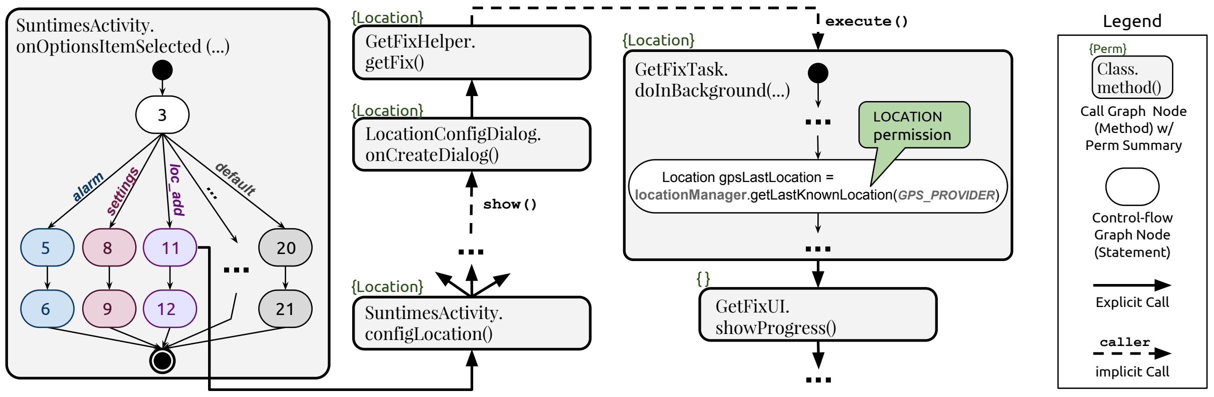

Figure 7: A sub-graph of inter-procedural control-flow graph for Suntimes app. The collapsed parts of the sub-graph are denoted by "…". The method call in node (=line) 11 of the control-flow-graph for `SuntimesActivity.onOptionsItemSelected()` method eventually leads to calling an Android framework API that requires Location permission (i.e. `getLastKnownLocation`). Since this permission is used under the branch with widget id `location_add`, it is inferred that Location is a relevant permission for a GUI test that exercises this entry-point method (`onOptionsItemSelected`) by performing an action on `location_add` widget

for the permission summaries (line 16), meaning that *PS* does not change in further iterations. In Figure 7, the permission summaries are shown at the top-left corner of each call-graph node.

6.2 Widget Analysis

Recall the entry-point method presented in Figure 4. To handle a selected menu option, this method (`onOptionsItemSelected`) invokes several other methods, each one under a `case` corresponding to the menu option. For instance, *Set Alarm* (third option in Figure 1b) is handled by the first `case` statement shown in Figure 4, where `scheduleAlarm()` method is called consequently (line 5). Therefore, if a GUI test only clicks on Set Alarm option, it does not execute the methods called in other cases, and thereby the summarized permissions for other methods (e.g., `showSetting()`, `configLocation()`, etc.) are irrelevant to this test. To exclude the irrelevant permissions, we need to determine which widgets affect the control-flow of which program statements, particularly the statements that invoke methods with non-empty permission summaries. *Widget Analysis* procedure, summarized in Algorithm 3, provides this capability.

For a given method, *Widget Analysis* procedure performs a *branch-sensitive, partial, inter-procedural* data-flow analysis and generates the *Widget Summaries (WS)* as the output. For this purpose, a trimmed version of inter-procedural control-flow graph (ICFG) is constructed first (line 3). An ICFG is a collection of control-flow graphs connected to each other at all call sites. Our analysis, however, targets app widgets exclusively and thus, only the call sites that pass a widget object are included in the trimmed ICFG, denoted as $ICFG^T$. Performing the analysis over $ICFG^T$, instead of ICFG, significantly improves the scalability of our approach, yet keeps the precision acceptable.

Afterwards, the *gen* set is populated through iterating over every statement of each method (lines 4–11). We are only interested in the conditional statements that affect the control-flow of the program, namely `IF` (lines 5–6) and `SWITCH` (lines 7–10) statements, with the widget as the condition. For instance, the `switch` statement in Figure 4 could be a target of our analysis, as it is (1) a conditional statement controlling the flow of the program, and (2) a widget, i.e., *MenuItem*, is used as the statement's condition.

Finally, the algorithm traverses the $ICFG^T$ in a breadth-first search manner and propagates the widget information through the graph. By doing this, at each statement we have the information of all widgets that can affect the control-flow of the program from the beginning to that statement. For example, as highlighted in the control-flow graph of `onOptionsItemSelected` method depicted in Figure 7, with `location_add` as the selected menu option, the

Algorithm 3: Widget Analysis

```
Input: AUT: App under test
Output: WS: Widget Summaries
1  $WS \leftarrow \emptyset$
2  $gen, in, out \leftarrow \emptyset$
   // WS, gen, in, and out are maps with program statement as its
      key and set of related widgets as its value.
3  $ICFG^T \leftarrow$ constructTrimmedICFG($AUT$)
4  foreach $stmt \in AUT.methods.statements$ do
5      if $stmt.type$ is IF & $stmt.condition.type$ is WIDGET then
6          $gen[stmt.target] \leftarrow stmt.condition$
7      else if $stmt.type$ is SWITCH & $stmt.condition.type$ is WIDGET then
8          foreach $case \in stmt.cases$ do
9              $gen[case.target] \leftarrow case.condition$
10         end
11 end
12 repeat
13     foreach $stmt \in BFS.next(ICFG^T)$ do
14         foreach $stmt' \in pred(stmt)$ do
15             $in[stmt] \leftarrow in[stmt] \cup out[stmt']$
16         end
17         foreach $stmt' \in succ(stmt)$ do
18             $out[stmt'] \leftarrow in[stmt] \cup gen[stmt]$
19         end
20     end
21     $WS \leftarrow out$
22 until $WS$ reaches a fixed-point;
```

```
{
<entryPoint:"SuntimesActivity:boolean onOptionsItemSelected(MenuItem)",
 widget: action_location_add(2131624168),
 permission: LOCATION>
}
```

Figure 8: A subset of *EWP* generated for Suntimes app

control-flow of the program will reach to lines 11 and 12. Hence, `location_add` is added to the widget summaries of the statements at nodes 11 and 12. The widget analysis terminates upon reaching a fixed point for the widget summaries (WS).

It is essential to note the difference between the precision and scope of two sub-procedures described in Sections 6.1 and 6.2, namely *Permission Analysis* and *Widget Analysis*. Due to flow and branch sensitivity, Widget Analysis is more costly than Permission Analysis. On the other hand, while Permission Analysis is performed on every method in the app through traversing its call graph, the scope of Widget Analysis is limited to a few entry-point methods exercised by running the tests. This distinction lets PATDroid keep the app analysis precise and yet, scalable.

Combining the outputs of *Permission Analysis* and *Widget Analysis* sub-procedures, the main procedure (Algorithm 1) generates the final output of *AUT Analyzer* component, i.e., *EWP*. A subset of generated EWP for Suntimes app is provided in Figure 8.

7 BUILDING PERMISSION COMBINATIONS

As shown in Figure 3, *Interaction Detector* generates the final output, TP, which is a map from tests to the set of relevant permissions. It does so by correlating the outputs of the other components, namely *TE*, *TW*, and *EWP*, as follows.

Interaction Detector procedure, summarized in Algorithm 4, iterates over the three input sets (*TE*, *TW*, *EWP*), and matches the tuple members of these sets based on the shared elements, i.e., *entry-Point*, *test*, and *widget*.[4] The only exception occurs when no widget

[4]Matching elements are distinguished by the same colors in Figures 5, 6, 8, and 9.

Algorithm 4: Interaction Detector

```
Input: TE= {⟨test, entryPoint⟩}, TW= {⟨test, widget⟩},
EWP= {⟨entryPoint, widget, permission⟩}, THA:Test harness app
Output: TP, A map with tests as the key and the set of relevant
        permissions as the value.
1  $TP \leftarrow \emptyset$
2  $testWithPerm \leftarrow \emptyset$
3  foreach $ewp \in EWP$ do
4      foreach $tw \in TW$ do
5          if $ewp.widget = \varnothing$ Or $ewp.widget = tw.widget$ then
6              foreach $te \in TE$ do
7                  if $ewp.entryPoint = te.entryPoint$ then
8                      if $te.test = tw.test$ then
9                          $TP[te.test] \leftarrow TP[te.test] \cup ewp.perm$
10                         $testWithPerm \leftarrow testWithPerm \cup te.test$
11             end
12     end
13 end
14 foreach $test \in THA.tests$ do
15     if $test \notin testWithPerm$ then
16         $TP[test] \leftarrow \emptyset$
17 end
```

```
[
testSunTimesNavigation:{},
testSettingLocationToUserDefined:{LOCATION},
testExportLocations:{LOCATION, STORAGE}
]
```

Figure 9: Relevant permissions for a subset of the tests listed in Figure 2

is found for an *EWP* (i.e., no widget is used to control access to permission-protected code in an entry-point), in which case it is conservatively assumed that the entire entry-point method could be executed by a single test and hence, the algorithm does not attempt to match $EWP.widget$ and $TW.widget$ (line 5). Based on the matched tuples, relevant permissions for a test are added to the output, TP (line 9). Finally, an empty set is assigned to those tests that have no relevant permissions (lines 14–17). The generated output, TP, for the test set of Suntimes app is provided in Figure 9.

The output of this algorithm enables efficient permission-aware testing of the given app. In total, for an app consisting of T tests and P permissions, the number of test-runs by PATDroid are calculated as follows:

$$\sum_{t=1}^{T} 2^{|TP[t].perms|}$$

where $TP[t].perms$ denotes the relevant permissions for test t identified by PATDroid. As our experiments will show, this number turns out to be significantly smaller than $|T| \times 2^{|P|}$ tests required for execution under the exhaustive approach.

8 EVALUATION

Our evaluation of PATDroid addresses the following questions:

RQ1. *Efficiency*: How does PATDroid compare against alternative approaches with respect to test-run size and test-execution time?

RQ2. *Coverage*: How does PATDroid compare against alternative approaches with respect to code coverage?

RQ3. *Effectiveness*: Is PATDroid able to reveal defects in real-world apps, particularly those that are only exposed under certain permission settings?

RQ4. *Performance*: How does PATDroid scale in relation to the size of app?

8.1 Experiment Setup

To evaluate our approach on realistic subjects, we crawled Google Play and GitHub repositories and searched for Android apps with the following criteria:

Table 1: A subset (those with available source code) of subject apps.

App	Size	# of permissions		test-suite
	(KLOC)	all	dangerous	size
A2DP Volume [1]	9.1	17	9	17
AlwaysOn [2]	15.9	18	6	16
Budget Watch [6]	8.0	3	2	12
Dumbphone Assistant [7]	1.4	3	3	7
Notes [9]	5.7	3	2	29
RadioBeacon [11]	31.4	10	6	17
Riot [12]	55.2	15	6	20
SMS Scheduler [14]	1.5	4	2	6
Suntimes [15]	22.4	4	3	13
SysLog [16]	12.1	4	2	13

Table 2: Test size and time reduction achieved by PATDroid compared to other approaches. +, - indicate that the reduction achieved by the the alternative approach is greater, or less than PATDroid, respectively.

App	Test-run size (% of difference compared to PATDroid)				Testing time in sec. (% of difference compared to PATDroid)			
	PATDroid $\sum_{t=1}^{\|T\|} 2^{\|TP[t].perms\|}$	Exhaustive $\|T\| \times 2^{\|P\|}$	Pairwise	All&None $2 \times \|T\|$	PATDroid	Exhaustive	Pairwise	All&None
A2DP Volume	59	8,704(-99.32%)	136(-56.62%)	34(+73.53%)	314	39,591(-99.21%)	619(-49.32%)	155(+102.74%)
Always On	29	1,024(-97.17%)	96(-69.79%)	32(-9.38%)	208	7,133(-97.09%)	669(-68.92%)	223(-6.75%)
Budget Watch	15	48(-68.75%)	48(-68.75%)	24(-37.50%)	67	221(-69.47%)	221(-69.47%)	110(-38.94%)
Dumbphone Assist	56	56(0%)	28(+100.00%)	14(+300.00%)	447	447(0%)	224(+100.00%)	112(+300.00%)
Notes	35	116(-69.83%)	116(-69.83%)	58(-39.66%)	162	531(-69.50%)	531(-69.50%)	266(-38.99%)
RadioBeacon	66	1,088(-93.93%)	102(-35.29%)	34(+94.12%)	462	6,200(-92.55%)	581(-20.50%)	194(+138.50%)
Riot	48	1,280(-96.25%)	120(-60.00%)	40(+20.00%)	398	10,379(-96.16%)	973(-59.05%)	324(+22.84%)
SMS Scheduler	7	24(-70.83%)	24(-70.83%)	12(-41.67%)	34	110(-69.11%)	110(-69.11%)	55(-38.23%)
Suntimes	32	104(-69.23%)	52(-38.46%)	26(+23.08%)	317	931(-65.92%)	465(-31.84%)	233(+36.32%)
SysLog	27	52(-48.08%)	52(-48.08%)	26(+3.85%)	144	299(-51.77%)	299(-51.77%)	149(-3.54%)

In the presented formulas used for calculating the size of the test-runs, T is the set of tests, P is the set of app's permission, and $TP[t].perms$ is the set of relevant permissions for the test t generated by PATDroid.

Table 3: Test coverage achieved by PATDroid compared to other approaches. +, - indicate that the coverage of the alternative approach is greater, or less than PATDroid, respectively.

App	Statement Coverage (% of difference compared to PATDroid)				Branch Coverage (% of difference compared to PATDroid)			
	PATDroid	Exhaustive	Pairwise	All&None	PATDroid	Exhaustive	Pairwise	All&None
A2DP Volume	49.55%	49.55%(0%)	49.55%(0%)	47.18%(-5%)	23.87%	23.87%(0%)	23.87%(0%)	21.61%(-9%)
Always On	45.31%	45.31%(0%)	10.54%(-77%)	45.31%(0%)	25.58%	25.58%(0%)	1.69%(-93%)	25.58%(0%)
Budget Watch	72.24%	72.24%(0%)	72.24%(0%)	56.52%(-22%)	51.04%	51.04%(0%)	51.04%(0%)	37.35%(-27%)
Dumbphone Assist	64.90%	64.90%(0%)	7.56%(-88%)	64.90%(0%)	43.10%	43.10%(0%)	11.21%(-74%)	43.10%(0%)
Notes	77.89%	77.89%(0%)	77.89%(0%)	62.24%(-20%)	61.30%	61.30%(0%)	61.30%(0%)	48.54%(-21%)
RadioBeacon	49.22%	49.22%(0%)	49.22%(0%)	43.24%(-12%)	25.76%	25.76%(0%)	25.76%(0%)	22.41%(-13%)
Riot	50.40%	50.40%(0%)	46.49%(-8%)	46.92%(-7%)	42.28%	42.28%(0%)	40.24%(-5%)	39.81%(-6%)
SMS Scheduler	65.25%	65.25%(0%)	65.25%(0%)	65.25%(0%)	45.32%	45.32%(0%)	45.32%(0%)	45.32%(0%)
Suntimes	50.23%	50.23%(0%)	50.23%(0%)	44.14%(-12%)	32.95%	32.95%(0%)	32.95%(0%)	27.23%(-17%)
SysLog	71.33%	71.33%(0%)	71.33%(0%)	65.66%(-8%)	48.75%	48.75%(0%)	48.75%(0%)	42.19%(-13%)

(i) Should target Android API level ≥ 23; otherwise, the app does not support run-time permission modification, and thereby does not suffer from the problems that are the focus of our work.
(ii) Should define at least two *dangerous* permissions in the manifest file, because other types of permissions are not adjustable at run-time and solving the problem with less than two adjustable permissions is trivial.

In accordance with the above criteria, we collected 110 apps: (1) 100 popular apps from Google Play, and (2) 10 open-source apps from Github (listed in Table 1), since investigating RQ2, i.e., measuring code coverage, requires the availability of source code.

For the open-source apps, we manually created or extended the existing GUI tests using Espresso [5] or Robotium [13] frameworks to achieve at least apprixımately 50% statement coverage. For Google Play apps we used Monkey [8] to generate black-box GUI tests.

We have compared PATDroid against three alternative strategies, as follows:

Exhaustive—exhaustively includes all permission combinations.

Pairwise—generated according to pairwise technique [43]; that is, for any two permissions, all possible pairs of permission settings (i.e., granted, revoked) should be in the output set.

All-and-None—includes two combinations, one with all permissions granted, the other with all permissions revoked.

As we will discuss in Section 9, none of the existing test suite reduction tools support Android framework, nor consider its access control model, therefore, are not included in our evaluation.

8.2 Efficiency

To answer RQ1, we compare the test-run size and test-execution time of PATDroid with *exhaustive*, *pairwise* and *all-and-none*, as shown in Table 2. *Test-run size* indicates the cumulative number of tests required to run for each technique. This number is calculated by the formulas shown in Table 2, under the corresponding columns. In addition, the table shows the percentage of decrease or increase for each reported metric in comparison to PATDroid.

The results in Table 2 confirm that PATDroid can significantly reduce the number of test-runs and test-execution time. On average, PATDroid requires 71.35% and 41.78% less number of test executions than *exhaustive* and *pairwise*, respectively. Similarly, on average, PATDroid takes 71.07% and 39.07% less execution time than *exhaustive* and *pairwise*, respectively. In comparison to *all-and-none*, however, the results are mixed, where in some cases PATDroid achieves a higher reduction (e.g., *Budget Watch*), while in other cases PATDroid achieves a lower reduction (e.g., *RadioBeacon*). Although *all-and-none* achieves a higher reduction in some cases, the next section shows that it does not maintain the same coverage as other approaches.

Figure 10 plots the test-execution time for all of the 110 subject apps. As illustrated in the figure, test-execution time grows exponentially with respect to the number of permissions in exhaustive approach. Therefore, the reduction rates compared to *exhaustive* approach are higher in apps with more permissions. For example, the reduction in the case of *A2DP Vol* app with 9 permissions is above 99%, while the reduction in the case of *Budget Watch* app with 2 permissions is close to 70%.

Table 4: A subset (those with public issue tracker) of defects in real-world Android apps identified for the first time by PATDroid.

App	Reported issue link	Defect Type	Status
Open Food	https://goo.gl/4eIm3E	Crash	Fixed
Budget Watch	https://goo.gl/8XBvkf	Unexpected Behavior	Fixed
A2DP Volume	https://goo.gl/9sfS09	Unexpected Behavior	Fixed
RadioBeacon	https://goo.gl/80Mb5j	Crash	Verified
Riot	https://goo.gl/MNEdkx	Unexpected Behavior	Fixed
OpenNoteScanner	https://goo.gl/yKNiRZ	Crash	Reported

8.3 Coverage

To answer RQ2, we compare the statement and branch coverage achieved by PATDroid against that of achieved by the alternative techniques. As shown in Table 3, PATDroid achieves the same exact coverage as *exhaustive* in all subject apps. The fact that PATDroid achieves the same coverage as *exhaustive* is particularly important, as it shows that PATDroid does not produce many false negatives, i.e., failing to execute a test with a relevant permission combination for an app.

Moreover, on average, PATDroid achieves 14% and 10% higher coverage than *pairwise* and *all-and-none* techniques, respectively. It is worth noting that while in 7 apps *pairwise* achieves the same coverage as PATDroid, in 3 apps it achieves significantly lower coverage. A closer look at the apps where PATDroid outperformed *pairwise* showed that these situations occur when certain capabilities provided by an app depend on more than two permissions. For instance, *AlwaysOn* app asks for four permissions, and if any of those permissions are not granted, the app's functionally is significantly downgraded. Since the *pairwise* technique does not include a combination with all four permissions granted, it achieves 77% lower statement coverage and 93% lower branch coverage than PATDroid.

In summary, the results of RQ1 and RQ2 confirm that PATDroid is able to significantly reduce the number of tests without trading-off code coverage.

8.4 Effectiveness

To answer RQ3, we investigate the power of our approach in identifying permission-related defects in real-world apps. To that end, we carefully analyzed Android log, and the output of the tests executed under the permission combinations generated by PATDroid.

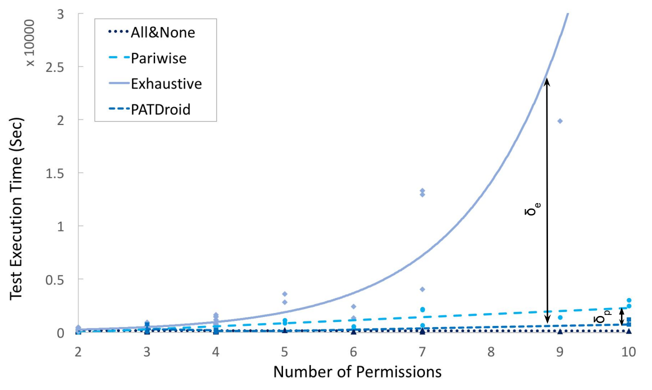

Figure 10: Test execution time based on the number of permissions. δ_e and δ_p represent the reduction in test execution time, achieved by PATDroid, compared to exhaustive and pairwise approaches, respectively

Particularly, we were interested in crashes or unexpected behaviors that could only be verified by running the tests under certain permission combinations.

Running PATDroid on the set of 110 apps, we found 14 apps (i.e., 13%) with defects that are due to inappropriate handling of dynamic permissions. We reported the identified defects for the open-source apps to their developers through GitHub issue tracker, along with information to reproduce the faults and suggestions for fixing the defects. Table 4 provides a summary of the reported defects and the current status of each issue for the apps that provide a public issue tracker. As of the date of this paper submission, most of the defects are verified and fixed by the app developers.

Note that *exhaustive* and *pairwise* approaches are also able to identify the reported defects, except they take significantly longer time to execute as shown in Section 8.2. *all-and-none* on the other hand, is not able to reveal these issues. For instance, in *Open Note Scanner* app, which asks required permissions initially, revoking the Storage permission while granting Camera permission would make the application crash. Such behavior is not reproducible using *all-and-none* technique. Furthermore, *exhaustive* approach was not able to find a defect that PATDroid missed, further demonstrating the efficacy of PATDroid in revealing permission-related defects.

8.5 Performance

To answer RQ4, we measured the performance of running PATDroid over the subject apps. The experiments are run on a PC with an Intel Core i7 2.4 GHz CPU processor and 16 GB of main memory. According to the experimental results, the average time spent on identifying the relevant permissions is 356 seconds, which is negligible compared to the time saved due to reducing the test-run size (See Section 8.2).

Figure 11 shows the performance measurements of running PATDroid. The analysis times for each phase of PATDroid, i.e., static and dynamic analyses, are plotted separately in the figure. On average, static and dynamic analyses take 97 and 259 seconds, respectively. According to the figure, the static analysis time increases as the app size increases, while there is no correlation between the dynamic analysis time and the app size. Dynamic analysis time depends on the logic and workload of the subject app.

9 RELATED WORK

Test reduction has been the goal of research efforts in several domains. In this section, we provide a discussion of such efforts in light of our research.

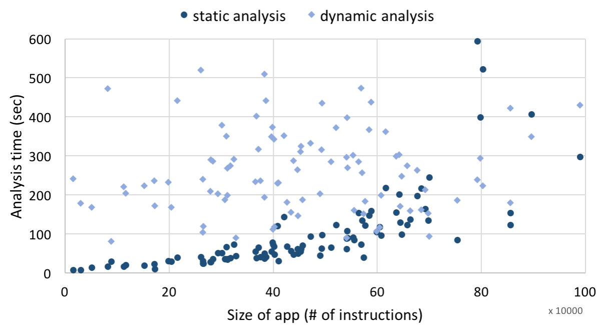

Figure 11: Performance measurements of PATDroid

Combinatorial Interaction Testing (CIT). Combinatorial interaction testing proposes a set of techniques to reduce the test space of a software system while maintaining the effectiveness of the whole test space [46, 54]. CIT approaches can be categorized as greedy [26, 28, 29], heuristic-based [30, 44], genetic [50] and search-based algorithms [51].

Another thrust of research in CIT includes identifying and removing *constraints* from configuration. While the main body of research [24, 27, 53] has focused on the *hard constraints*–those making the configuration infeasible or not permitted, the problem of *soft constraints*–valid, yet undesirable or irrelevant configuration, is unexplored by the prior research. To the best of our knowledge, PATDroid is the first attempt at reducing the test space by addressing soft constraints in the context of access control, i.e. excluding app permissions that are irrelevant to the test cases.

Testing Software Product Lines (SPL). Systematic testing of programs constituting software product lines (SPL) is expensive, as it requires examining combinations of the features for testing. Therefore, several prior works [33–36] have attempted to reduce the test space of SPL. For instance, *SPLat* [36] and its predecessor [34] leverage dynamic and static program analysis techniques to identify and exclude irrelevant features from each test case.

Despite having similar objectives, the proposed techniques for SPL test reduction are not applicable to permission-aware GUI testing of Android apps for several reasons. In contrast to the SPL features that are explicitly specified in the code, app permissions are not embedded in the app code. Moreover, the target of SPL test reduction techniques is *unit* tests. While unit tests are traceable from the program, there is no explicit dependency between the source code and the GUI test suite in the case of Android programs, as discussed in Section 5.

Regression Test Selection (RTS). A large body of prior research has focused on speeding up the regression testing in continuous integration processes [31, 47, 48, 55]. In contrast to these RTS techniques that track dependency at the file, class, or method level, PATDroid captures the dependency at the control-flow level. Applying a precise yet scalable analysis, PATDroid is able to significantly reduce the testing time. Despite the similarity of the techniques applied for tracking the dependencies between tests and system under test, the goal of our approach is rather different from RTS. While RTS techniques aim to identify the tests relevant to the changes introduced in the codebase due to software revisions, the goal of PATDroid is to find the relevant permissions for GUI testing.

None of the aforementioned three categories of related research are applicable to Android. We next briefly provide an overview of the related work in Android GUI testing.

Android GUI Testing. Android GUI testing has received substantial attention in recent years [25]. Proposed approaches employ a variety of techniques, including random [17, 18, 32], hueristic-based [40], model-based [21, 52], and search-based [41] approaches.

Nonetheless, only a few research approaches have applied combinatorial test reduction techniques to Android testing domain. Most notably, *TrimDroid* [42] extracts dependencies among the widgets to reduce the number of combinations in GUI testing. TrimDroid, however, differs from our approach in several ways. First, TrimDroid performs *static* analysis over the app code to generate new test cases, while PATDroid employs *hybrid* analysis on both app under test and test harness app to determine a subset of permission combinations for running the existing tests. Second, TrimDroid captures dependencies among the widgets, while PATDroid tracks dependencies among app widgets and permissions. To the best of our knowledge, access control models, particularly permissions, are not considered in any of the prior research for test generation or reduction.

10 CONCLUSION AND FUTURE WORK

Recent introduction of a dynamic permission system in Android has made it necessary to test the behavior of Android apps under a variety of permission settings. Without an automated solution to reason about which tests should be executed under what permission combinations, the developers have to either manually make such determinations or exhaustively re-run each test under an exponential number of permission combinations. Both approaches are impractical, time-consuming, and cumbersome.

To overcome this problem and help developers efficiently test Android apps under various permission settings, we presented PATDroid. Through a hybrid program analysis of Android app and its test suite, PATDroid is able to identify relevant permissions for each test case. By excluding the irrelevant permissions, PATDroid is able to significantly reduce the number of test runs and execution time of tests without trading-off coverage and fault detection ability of tests. Our experimental results show that PATDroid can achieve 71% reduction in execution time of tests compared to the exhaustive approach, without any degradation in code coverage. Moreover, using PATDroid, we were able to identify several previously unknown permission-related defects in real-world apps.

Our current implementation of PATDroid has two limitations that will be the subject of our future work.

First, it is assumed that GUI widgets are retrieved by a single unique identifier, such as ID or key. Though this assumption is true in the vast majority of cases, widgets are rarely retrieved by a combination of multiple properties, such as position and parent's ID.

Second, it is assumed that the GUI-dependent flow of the program is controlled directly using the widgets. Meaning that, if the execution of a program block depends on a specific GUI event, developers specify a conditional guard on the widget triggering that event. However, another less common, yet plausible scenario is that a developer indirectly uses the widget. For example, by saving a reference to the widget in a global variable and using this reference to control the flow of program.

Addressing the aforementioned limitations is possible through the construction of more advanced static analyses. However, the additional precision is likely to reduce the scalability and performance of our approach for little gain in the overall efficacy. Studying such tradeoffs will be a focus of our future research.

We plan to extend our approach to include other configurable parameters in Android that can affect the behavior of programs, such as the settings for network and battery usage. Finally, we intend to investigate the applicability of our approach to other platforms that use permission-based security model.

REFERENCES

[1] 2017. A2DP Volume. (2017). https://github.com/jroal/a2dpvolume

[2] 2017. Always On. (2017). https://github.com/rosenpin/AlwaysOnDisplayAmoled

[3] 2017. Android Permissions Best Practices. (2017). https://developer.android.com/training/permissions/best-practices.html

[4] 2017. Android Studio User Guide: Test your app. (2017). https://developer.android.com/studio/test/index.html

[5] 2017. Android Testing Support Library : Espresso. (2017). https://google.github.io/android-testing-support-library/docs/espresso/

[6] 2017. Budget Watch. (2017). https://github.com/brarcher/budget-watch

[7] 2017. Dumbphone Assistant. (2017). https://github.com/yeriomin/DumbphoneAssistant

[8] 2017. The Monkey UI android testing tool. (2017). https://developer.android.com/studio/test/monkey.html

[9] 2017. Notes. (2017). https://github.com/SecUSo/privacy-friendly-notes

[10] 2017. PATDroid: Permission-Aware Testing for Android. (2017). https://sites.google.com/view/patdroid

[11] 2017. Radio Beacon. (2017). https://github.com/openbmap/radiocells-scanner-android

[12] 2017. Riot. (2017). https://github.com/vector-im/riot-android

[13] 2017. Robotium. (2017). http://robotium.com/pages/about-us

[14] 2017. SMS Scheduler. (2017). https://github.com/yeriomin/SmsScheduler

[15] 2017. Suntimes. (2017). https://github.com/forrestguice/SuntimesWidget

[16] 2017. SysLog. (2017). https://github.com/Tortel/SysLog

[17] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. 252–261.

[18] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. 258–261.

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 259–269.

[20] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 217–228.

[21] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 641–660.

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

[23] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. 73–84.

[24] Renée C. Bryce and Charles J. Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology* 48, 10 (2006), 960–970.

[25] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 429–440.

[26] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatiorial Design. *IEEE Trans. Software Eng.* 23, 7 (1997), 437–444.

[27] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. Software Eng.* 34, 5 (2008), 633–650.

[28] Charles J. Colbourn, Myra B. Cohen, and Renée Turban. A deterministic density algorithm for pairwise interaction coverage. In *IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 17-19, 2004*. 345–352.

[29] Jacek Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios, Portland, Oregon, October 14-15, 2008. In *Proceedings of 24th Pacific Northwest Software Quality Conference*. 419–430.

[30] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102.

[31] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. 211–222.

[32] Cuixiong Hu and Iulian Neamtiu. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*. 77–83.

[33] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *4th International Workshop on Feature-Oriented Software Development, FOSD '12, Dresden, Germany - September 24 - 25, 2012*. 1–8.

[34] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011*. 57–68.

[35] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don S. Batory. Shared Execution for Efficiently Testing Product Lines. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. 221–230.

[36] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don S. Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 257–267.

[37] Butler W. Lampson. 1974. Protection. *Operating Systems Review* 8, 1 (1974), 18–24.

[38] Li Li, Alexandre Bartel, Tegawend F. Bissyand, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 280–291.

[39] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 229–240.

[40] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 224–234.

[41] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 599–609.

[42] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 559–570.

[43] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2 (2011), 11:1–11:29.

[44] Kari J. Nurmela. 2004. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics* 138, 1-2 (2004), 143–152.

[45] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick D. McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 77–88.

[46] Alessandro Orso and Gregg Rothermel. Software testing: a research travelogue (2000-2014). In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. 117–132.

[47] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia C. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*. 432–448.

[48] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (1997), 173–210.

[49] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 515–526.

[50] Toshiaki Shiba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings*. 72–77.

[51] Charles Song, Adam A. Porter, and Jeffrey S. Foster. 2014. iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees. *IEEE Trans. Software Eng.* 40, 3 (2014), 251–265.

[52] Wei Yang, Mukul R. Prasad, and Tao Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings.* 250–265.

[53] Cemal Yilmaz. 2013. Test Case-Aware Combinatorial Interaction Testing. *IEEE Trans. Software Eng.* 39, 5 (2013), 684–706.

[54] Cemal Yilmaz, Sandro Fouché, Myra B. Cohen, Adam A. Porter, Gülsen Demiröz, and Ugur Koc. 2014. Moving Forward with Combinatorial Interaction Testing. *IEEE Computer* 47, 2 (2014), 37–45.

[55] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011.* 23–32.