ABSTRACT

Ensuring the reliability and security of Android apps is important considering the large Android market and the critical usage of Android apps. To analyze and test Android apps, we need to know program paths, i.e., the control flow of callbacks implemented in the apps. One of the challenges to identify such information is the extensive use of the Android API methods in the apps. These methods can invoke multiple callbacks, and the control flow of these callbacks is context-sensitive in that different callback sequences may be invoked at different API call sites. To address this challenge, in this paper, we design a summary representation for an Android API method that aims to capture the control flow of callbacks in the API methods as well as the conditions under which the callbacks are invoked. We developed a static demand-driven analysis to automatically generate such summaries. To show the usefulness of the summaries, we construct the apps’ control flow graphs (CFGs) and apply infeasible path detection on the CFGs. Our experiments show that we are able to generate the API summaries that are compact and reusable, and by replacing the API calls with the summaries we generated, we obtained the apps’ CFGs with paths up to 10 callbacks for a set of Android apps under study. Comparing to the dynamic traces generated, we verified that such paths contain valid callback sequences. The API summaries and the CFGs of the apps computed for this work are all available at: http://www.cs.iastate.edu/~weile/research/lithium.html

1. INTRODUCTION

Android apps are implemented by extending the callbacks of the Android framework. Determining the control flow of the callbacks in the apps is important. With the order of callbacks and the program paths, developers can perform code inspection and predict whether the apps potentially behave as expected at runtime. Static analysis, testing and debugging tools can then rely on valid callback sequences to detect and diagnose bugs across multiple callbacks. In the previous research, FlowDroid has found vulnerabilities produced along a sequence of callbacks [4]. FlowDroid applied Android lifecycles to sequence callbacks in the app. We suspect that with a more precise control flow model for the callbacks, more of such bugs can be found. In addition to software assurance tools, control flow information of the callbacks is useful for verifying code transformations such as code refactoring. For example, recently, the researchers found that we do need to understand multiple callbacks to accomplish the refactoring tasks related to Android asynchronous constructs [15].

The execution order of callbacks in an Android app is mainly determined by the two factors. First, Android apps are event-driven systems, and a certain set of callbacks are designed to respond to the external events such as click a button on the GUI or report an update location by the GPS sensor. The order of such callbacks is determined by the order of events triggered when we use an app. Sequencing such callbacks is not the scope of this paper. There are also an extensive number of callbacks in the Android apps that are invoked by the Android API calls (APIs exported from the Android framework). We studied a total of 930 Android apps from F-Droid [1] and from the Google play market. We found that every app used Android API calls. On average, each app invokes 1204 Android API calls, and the maximum reaches 4029. On the other hand, we studied the top 100 frequently used Android API methods and found that 47% of the methods contain callbacks and 43% contain multiple callbacks. That said, much of the control flow of the callbacks in an app is determined internally by the API implementation. In this paper, we aim to summarize the API methods for such control flow and enable the construction of apps’ CFGs using the summaries. The summaries can be reused for different Android apps.

In the previous research, the static analysis tools mainly used Android lifecycles to construct the apps’ CFGs [4,14,17, 26]. The lifecycle models only represent the partial behavior implemented in the Android framework. The summaries we construct directly from the source code of the Android API methods are aimed to be more complete. In addition, the lifecycle models are built only for a few Android components such as Activity and Service. There are many more Android API methods that contain the control flow information of the callbacks, which we aim to uncover in this work. The researchers have made effort on building apps’ CFGs from analyzing GUI [24, 25]. Such control flow of callbacks is mainly relevant to external events. Thus, their techniques are complementary to our work. In fact, they built manual
models for a set of Android API methods, but the models only consider one callback per API method, which is incomplete. Since it is hard to build manual models for the entire Android framework, our automatically generated summaries can be used in their work to improve the coverage of the API methods. Considering the Android framework is update frequently, we need the automatic approaches to quickly generate new models for the new code. There is also work that automatically maps an Android API to the callbacks invoked in the API method [8], but it did not report the orders of these callbacks.

In this paper, we present techniques of automatically generating summaries for Android API methods that can be used for control flow analysis of Android apps. The summary extracts three pieces of information from the implementation of the API method: 1) the callbacks invoked and their orders 2) the conditions under which a callback is invoked, and 3) the updates of program states with regard to callback invocation conditions. The callback orders are used to construct program paths for the apps. The conditions and the updates relevant to the conditions can be used to determine the feasibility of the callback sequences; such callback sequences can be located within one or across multiple API methods. The conditions also can be used, e.g., in debugging or code inspection, to understand which callbacks should be invoked under a particular program context of interest. The summaries should be compact, so that the developers can easily understand them, and the tools can efficiently use them. The summaries should be able to be directly plugged into any app’s API call site for control flow analysis without further recomputation. It requires that we abstract away all the information local to the Android framework and only present in the summary the information visible to the apps.

Based on the above objectives, we design the summary representation as a graph that contains three types of nodes: callback nodes, specifying which callbacks are invoked, predicate nodes, indicating when a callback is invoked, and condition update nodes, pinpointing when and how the program state changes with regard to callback invocation conditions. Summarizing an API method, we perform three main tasks: 1) select program statements that can be the three types of nodes, 2) replace all the local variables in the selected program statements with the abstract variables defined to only contain variables visible at the API client side, and 3) construct edges between the three types of nodes based on the original control flow of the selected program statements.

The main research challenge we encountered is scalability, as Android API methods are large and typically invokes many internal calls in the Android framework. We integrated a set of techniques to address the scalability. First, we found that the naive approach of exhaustively traversing all the paths of an API method for summarization is not scalable. As an alternative, we perform a backward demand-driven traversal [7,11] starting at the callback call sites for summarization. Second, to identify and summarize the predicate nodes, we first identified all the conditional branches a callback is transitively control dependent on. We then perform a path-insensitive backward symbolic substitution from each branch node to resolve the local variables. Third, we allow to use the return values of the framework calls to specify the predicated nodes and condition updated nodes instead of unfold these calls. To ensure such summaries are still useful, we define templates to match pairs of the framework calls located in the predicate nodes and the condition update nodes for determining the feasibility of callback sequences.

To demonstrate that the summary graphs are useful, we develop a client analysis to identify the callback implementation at the API call site and use the API summaries to generate the CFGs for the apps. We also applied infeasible path detection [7] on the apps’ CFGs to show the usefulness of the predicate nodes and the condition update nodes.

We implemented the tool Lithium and generated summaries for the top 100 frequently used API methods on the Android Framework 5.1. Our experiments show that generating summaries take 744 seconds on average. Compared to the ICFGs of the API methods, the summaries are 98% smaller on average. Using the summaries, we are able to generate callback sequences up to 10 callbacks.

To the best of our knowledge, this is the first work on summarizing control flow of callbacks in API methods. The summaries computed here can be used for developers to understand the Android APIs or used by static analysis and testing tools to construct CFGs for Android apps. In summary, we made the following contributions in the paper:

- defining a summary representation that specifies the control flow of callbacks in Android API methods,
- designing algorithms and a tool, Lithium, to compute such summaries,
- demonstrating that the summaries are useful by producing apps’ CFGs with summaries, and
- the engineering efforts and experience of analyzing real-world large libraries, and the experimental results that demonstrate the scalability, precision and usefulness of the techniques.

In the rest of the paper, we present the motivation of the work in Section 2. In Sections 3 to 5, we show how we define, compute and apply summaries. We present our implementation and experimental results in Section 6, followed by related work in Section 7, and the conclusions in Section 8.

2. MOTIVATION

In this section, we explain why Android API methods play an important role in determining control flow of the callbacks in Android apps.

2.1 Relations of Android APIs and Callbacks

Android apps are tightly coupled with the Android framework in that it implements many callbacks extended from the framework; meanwhile, the implementation of these callbacks invokes many Android API calls exported from the framework. For example, the code snippet in Figure 1 implements the callbacks onStart and onStop extended from the Activity class defined in the framework, and onCreate, onUnbind and onDestroy extended from the Service class. Here, we see that the source code of the app does not provide the control flow information of these callbacks. In the later sections, we will show that it is the two Android API calls startService and bindService invoked in OnStart (see lines 3 and 5) that determine the order of the callbacks onCreate, onUnbind and onDestroy implemented in TrackingRecordingService.
The Android framework contains over 20,000 APIs. We studied the top 100 frequently used API methods and found that 47 of them contain callbacks, and 43 contain multiple callbacks. Additionally, all the API methods with callbacks are sensitive to the context in which they are executed, and the sequence of callbacks executed may vary per call.

To collect the evidence that Android API calls are frequently used in the app, we studied 546 Android apps from F-Droid [1] and 384 apps from the Google play market. We found that every app used Android API calls. On average, each app invoked 1204 Android API calls. The app com.infraware.office.link.apk used the maximum number, reaching 4029. We also found that the Android API calls are commonly placed in different program contexts.

2.2 Bug Related to Context-Sensitive API Calls

Android bugs can be related to multiple callbacks. To determine such bugs, we need to correctly identify a feasible sequence of callbacks. Sometimes, the developers do not have a good understanding on the Android APIs, and as a result, they made mistakes and introduced bugs.

```java
class HostListActivity extends Activity {
    public void onStart() {
        this.startService(new Intent(this, TrackingRecordingService.class));
    }
    public void onStop() {
        super.onStop();
        this.unbindService(connection);
    }
}

class TrackingRecordingService extends Service {
    public void onCreate() {
        wifilock.acquire();
    }
    public boolean onUnbind(Intent intent) {
        if (bridges.size() == 0) this.stopSelf();
        return true;
    }
    public void onDestroy() {
        if (wifilock != null & wifilock.isHeld())
            wifilock.release();
    }
}
```

Figure 1: ConnectBot Bug

In Figure 1, we display a bug from the app ConnectBot. At lines 3–6, the app invokes `startService` and `bindService` in `HostListActivity` to start and bind the service `TrackingRecordingService`. As a result of the API calls, `onCreate` at line 14 is invoked, and the `WiFi lock` is acquired (see line 15). When the app needs to unbind the service, it invokes `unbindService` at line 10. This Android API method calls `onUnbind` if the service is started, or `onUnbind` followed by `onDestroy` if the service is not started. However, in this bug, the developer assumed the latter always happen and forgot using `stopService` or `stopSelf` (see line 18) to stop the execution of the service in `onUnbind`. As a result, `onDestroy` is never called and the service is never destroyed. The `WiFi lock` acquired at line 15 is never released (it was supposed to be released at line 23 in `onDestroy`). This type of bugs are called no-sleep bugs [17] and can drain the battery.

3. DEFINE SUMMARY

We design a summary representation with the goals of 1) helping developers understand the control flow of the callbacks invoked in the API implementation, and 2) enabling the automatic construction of the apps’ CFGs. Based on the two potential applications, we design a summary representation consisting of `callback nodes`, `predicate nodes` and `condition update nodes`. In this section, we first use an example to show what a summary graph looks like. We then introduce the definitions of the three types of nodes.

3.1 An Example

Figure 2 shows the summary graphs (simplified versions) for the two Android API calls `startService` and `unbindService` invoked at lines 3 and 5 in Figure 1. On the left, in the summary for `startService`, nodes 3, 4 and 5 are callback nodes, node 2 is the predicate node, and node 6 is the condition update node. Dependent on the conditions at the API call site, `g.thread == null` at node 2 can be `true` or `false`, which leads to two different callback sequences: along path (2, 3, 5, 6), the edges indicate that `onCreate` is invoked asynchronously, followed by another asynchronous call `onStartCommand`; along path (2, 4, 6), only `onStartCommand` is invoked. Node 6 is a condition update node because in the `bindService` summary shown on the right, there is a predicate node (node 3) that uses the value of `g.started`. The summary graphs indicate that as long as `startService` is invoked, `g.started` is set to true; as a result, node 3 in `unbindService` always returns false, and `onDestroy` is not invoked. If we can provide such summaries to the developers, they are unlikely to make the mistakes show in Figure 1.

![Figure 2: Summaries for startService, unbindService](image)

3.2 Summary Representation

As shown in Figure 2, a summary representation is a directed graph that consists of three types of nodes. The nodes are selected from the statements of the API methods. The summarization process removes the local variables in these statements and represent them using abstract variables.

Preliminary: Abstract Variables. The goal of designing abstract variables to represent summaries is to enable a

direct usage of the summaries for apps’ control flow analysis. As an example, see the code snippet of the API method initLoader in Figure 3. At line 16, a callback onReset is invoked through an object receiver info.c, and the condition is haveData is true (see line 15). However, both info.c and haveData are local variables of initLoader. Presenting such information in the summary did not help determine at the API call site which callback implementation we should use to instantiate onReset and whether haveData is true.

Therefore, in a summary representation, we design the abstract variables to replace the local variables of interest in the API method and only present the information that can be directly consumed at the API client side. Take haveData at line 15 in Figure 3 as an example. During summarization process, we perform backward symbolic substitution for haveData starting at line 15 and resolve it to the disjunction of the two abstract variables [calling object, LoaderManager, createLoader()].mHaveData ∨ calling object, LoaderManager, mLoaders.get().mHaveData.

The abstract variable consists of the three pieces of information. The first element specifies the visibility and scope of the variable and has the domain of the static variable (from the classes of the Android framework), the API calling object, the input parameters of the API method, and constants such as a boolean, numeric or null constant. The second element of an abstract variable specifies the type of the variable. The third element provides the details on how to compute the values for the variable, e.g., it can be an access path of a field or a return of a method call defined in the Android framework.

For example, in the abstract variable [calling object, LoaderManager, createLoader()].mHaveData shown above, calling object indicates that the callback invocation condition at line 15 in Figure 3 is related to the object receiver of the API call at the client side. The type of the object receiver is LoaderManager. Specifically, it is the field mHaveData in the return object of the createLoader method that determines whether or not the callback onReset will be invoked at a particular API call site. The disjunction ∨ between the two abstract variables means that either when the first or when the second abstract variable get the value of true, the callback invocation condition is satisfied.

class LoaderManager {
  Loader<D> initLoader(int id, Bundle args) {
    LoaderManager.LoaderCallbacks<D> callback) {
      LoaderManager r0 = this;
      boolean creatingLoader = r0.mCreatingLoader
      if (creatingLoader == true) {
        throw new IllegalStateException("...");
      }
      LoaderInfo info = mLoaders.get(id);
      if (info == null) {
        info = createLoader(id, callback);
        mLoaders.put(id, info);
      }
      boolean haveData = info.mHaveData;
      if (haveData == true) {
        info.c.onReset();
      }
  }
}

Figure 3: LoaderManager.initLoader

Callback Nodes. Callback nodes represent the callback call sites in the API implementation. The two key pieces of information specified in the callback nodes are 1) the object receiver represented using an abstract variable, and 2) the callback signature. With the information, we are able to find the corresponding implementation of the callbacks at the API call site.

There are several ways for an app to pass the object receiver of a callback into the framework. The object receiver can be the calling object of the API call in the app. The object receiver can also be passed through the input parameters of the API call. For example, in Figure 2, the object receiver of onCreate and onStartCommand at nodes 3, 4 and 5 are passed through the input parameter of startService. At the call site of startService at line 3 in Figure 1, we found that the actual parameter has a type TrackingRecordService. We then go to the definition of the class TrackingRecordService at line 13 to find the implementation of the two callbacks. Finally, the object receiver of a callback can be a static variable. The app can assign the values to the static variable through other Android API calls, and the static variable is later used as an object receiver to invoke the callback in the Android API method. Currently, in our tool, we handled the first two types.

Predicate Nodes. Predicate nodes provide the conditions satisfying which a callback or a sequence of callbacks will be executed in the API method. The conditions in the predicate nodes are specified using the arithmetic (+, −, /, ×), the logical (∨, ∧) and the boolean (==, >, <, ≥, ≤) operations on the abstract variables. For instance, in the summary, the statement at line 15 in Figure 3 will be converted to the predicate node [calling object, LoaderManager, createLoader()].mHaveData ∨ [calling object, LoaderManager, mLoaders.get()].mHaveData == true.

Predicate nodes provide the developers and tools the information on when a callback (or a sequence of callbacks) of interest will be invoked. It also can be used with the condition update nodes for determining the feasibility of a sequence of callbacks along the program paths of an app.

Condition Update Nodes. The condition update nodes are the ones that can update the program state and impact the invocations of succeeding callbacks located in the current API method or the succeeding API methods. A condition update node is either an assignment whose left side is an abstract variable and whose right side is an expression of the abstract variables, or a framework call whose object receiver is the abstract variable—in this case, the modification is through the side effect of the framework call.

The nodes can update the static variables (including their fields) that are visible to all the methods in the Android framework, the calling objects (including their fields) that are visible to other methods invoked using the same calling objects, or the parameters of the current API method which then can be passed into the succeeding API methods.

Using static variables as an example. At runtime, each app runs on an instance of the Android framework. Static variables record the status of the global data structures in the framework. Such variables can be communicated across a set of relevant API methods invoked at the different program points in an app. For example, the Android framework keeps a map of all the Service objects running for an app. The map is created when the app starts, and it can be modified and accessed throughout the app’s lifetime using the static variable. When any Service object is going to be started, the app invokes startService. This API call first
checks if the service object is already started by inquiring the static variable. If the object does not exist, the app creates the corresponding service by calling the onCreate callback and also updates the state of the service stored in the static variable.

**The Edges.** The edge in the summary graph can connect two callback nodes, which represents the order of the two callbacks. The edge also connects a predicate node and a callback node to indicate that the callback is guarded by the condition, or a condition update node and other nodes to specify the scope and order of the update. To specify the control flow between callbacks, we also distinguish the edge for synchronous or asynchronous invocation of a callback (or a sequence of callbacks).

4. **COMPUTE SUMMARY**

In this section, we present our algorithms to compute summary graphs from the ICFGs of the Android methods. As shown in Figure 4, our static analysis framework Lithium performs four phases: identifying callback nodes, computing predicate nodes, computing condition update nodes, and generating the summary graph. The two key ideas are 1) control flow analysis to identify the three types of nodes and 2) backward demand-driven symbolic substitution to resolve local variables in the nodes to be abstract variables. In the following, we describe the four phases in a detail.

![Figure 4: An Overview of Lithium](image)

**4.1 Identifying Callback Nodes**

The goal of this step is to locate callback call sites in the API methods. Specifically, we aim to identify the program paths that can reach the callbacks from the entry of the API method. Along these paths, we then can generate summaries for predicate nodes and condition update nodes. Once a callback call site is identified, we apply pointer analysis and type inference to resolve object receivers of the callbacks and represent them using abstract variables. The two challenges we addressed in this phase are 1) to achieve the scalability of the static analysis and 2) to handle asynchronous callbacks. The following two sections provide our solutions.

**4.1.1 Demand-Driven Analysis for Scalability**

Our initial study found that the API method can be large and invoke many internal framework calls. These calls are frequently invoked through dynamic dispatch, which dramatically increases the size of the call graphs. We found it is not scalable to exhaustively traverse the ICFG of the API method to perform summarization.

We designed a demand-driven analysis. In the first step, we analyze the class hierarchy and the visibility of the methods in the Android framework and identify a list of potential callback methods. Specifically, we find every non-static, non-final method of a non-final class that has a visibility of public or protected. Additionally, we also identify all the methods of public interfaces in the Android framework. These methods can be overridden by the apps and executed in the Android framework via dynamic dispatch, and thus possibly can be callbacks.

In the second step, we scan the ICFG of an API method and use the call signatures to match the nodes in the ICFG and potential callback methods identified above. We perform a backward traversal from the callback call site towards the entry of the API method to find the paths from the entry of the API method to the call site of the callback \( c \). The result is the call chain \( \{ m_1, m_2, ..., m_i \} \), where \( m_i \) is the caller of \( m_{i+1} \) and \( m_1 \) is the API method. These call chains are used by the next phases to ensure that our expensive analysis only traverses the paths of interest for scalability.

**4.1.2 Handling Asynchronous Callbacks**

Android implements a message passing mechanism using the Handler class. When `sendMessage` is invoked from a `Handler` object, a message is posted into the event queue. When the message is dispatched, `handleMessage` from the same `Handler` class is invoked to handle the message. The `handleMessage` method can invoke a set of callbacks. To include such asynchronous callbacks into the summary, we accomplish two tasks.

First, we determine the type of the object receiver for every asynchronous call `sendMessage`. We then find the `sendMessage` method in the class that defines the type. For example, in Figure 5, according to line 17, the object receiver of `sendMessage` at line 21 has a type `ActivityThreadHandler`. Thus, we identify `handleMessage` implemented in the class `ActivityThreadHandler` at line 4 is a match. Any callbacks invoked in this method should be linked to the asynchronous call site of `sendMessage` at line 21.

Second, we prune any infeasible code at the asynchronous call site. As an example, in Figure 5, `handleMessage` in the `ActivityThreadHandler` class handles messages related to both binding a service (lines 9-7) and launching an activity (lines 9-10). However, at the call site line 21, of the interest is to launch an activity (see the parameter `LAUNCH_ACTIVITY` passed into `sendMessage`). Therefore, we can prune the infeasible code at lines 9-10 and only link the callbacks implemented in the `handleStartActivity` method to the asynchronous call site at line 21.

**4.2 Computing Predicate Nodes**

Ideally, to compute predicate nodes, we need to identify the control slices for each callback node and summarize the conditions along the control slices in a path-sensitive way. In our feasibility study, we found such slices can be large and perform path-sensitive analysis is very slow. Our approach is to first perform control flow analysis to identify conditional branches that a callback node transitively control dependent on and report them as predicate nodes (Section 4.2.1). We then resolve the local variables contained in the predicate nodes in a path-insensitive way (Section 4.2.2). The tradeoff is that the conditions reported in the predicate nodes may not be as strict as the ground truth; when our summary determines a callback can be invoked, the real execution actually may not be able to reach the callback.
class ActivityThreadHandler extends Handler {
    static final int BINDSERVICE = 1;
    static final int STARTACTIVITY = 2;
    public void handleMessage(Message m) {
        switch (m.what) {
            case BINDSERVICE:
                handleBindService (...);
                break;
            case LAUNCHACTIVITY:
                handleStartActivity (...);
                break;
        }
    }
}

class Activity {
    ActivityThreadHandler threadHandler;
    public void startActivity (...) {
        ...
        threadHandler.sendMessage(LAUNCHACTIVITY);
    }
}

Figure 5: handleMessage in ActivityThreadHandler

4.2.1 Identifying Predicate Nodes

Predicate nodes are the conditional branch statements in the ICFG of the API method that decide whether a callback should be executed. For each method m_i appeared in the call chain \{m_1, m_2, ..., m_l\} computed from the phase Identifying Callback nodes, we applied a control flow analysis shown in Algorithm 1 to identify the predicate nodes in the method.

The input of the algorithm is the CFG of the method m_i, and p is the program point of interest. Dependent on m_i, it can either be the callback c or the call site of the method m_i+1 in the call chain. The algorithm reports a set of branch nodes where p is transitively control dependent on.

At line 2, the algorithm traverses every conditional branches in the method. At line 3, Influence(b) returns all the statements in the CFG that are transitively control dependent on the conditional branch statement b \[22\] \[23\]. If the statement of interest p is one of such statements, the branch nodes will be stored in the results B.

**Algorithm 1: Identifying Predicate Nodes**

Input: \( m_i : \text{CFG} = (N, E), p \in N : \text{callback c or call site of m}\_i+1 \)
Output: \( B \subseteq N : \text{the set of conditional branches such that for } b \in B, p \text{ is transitively control dependent on } b \)

1. \( B = \{ \} \)
2. foreach conditional branch \( b \in m_i \), do
   3. \( B \subseteq N : \text{the set of conditional branches such that for } b \in B, p \text{ is transitively control dependent on } b \)
   4. if \( p \in S \) then \( B = B \cup b \)
5. end
6. return \( B \)

4.2.2 Summarizing Predicate Nodes

The goal of this step is to convert any of the local variables in the predicate nodes to be abstract variables. The approach we used is demand-driven, backward symbolic substitution. The analysis starts at each predicate node. At any assignment that defines the local variables under tracking, we update the variables symbolically. If the assignment is through a method call, we do not traverse the callee; instead, we keep the method call as a part of the abstract variable. For example, in Section 3.2, we used createLoader().mHaveData to specify the abstract variable.

<table>
<thead>
<tr>
<th>Class</th>
<th>Predicate Nodes</th>
<th>Condition Update Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.List</td>
<td>isEmpty, size,</td>
<td>add*, remove*, set</td>
</tr>
<tr>
<td></td>
<td>get*, contains*</td>
<td></td>
</tr>
<tr>
<td>java.util.Set</td>
<td>isEmpty, size,</td>
<td>add*, remove*</td>
</tr>
<tr>
<td></td>
<td>contains*, get</td>
<td></td>
</tr>
<tr>
<td>java.util.Map</td>
<td>isEmpty, size,</td>
<td>put*, remove</td>
</tr>
<tr>
<td></td>
<td>contains*, set</td>
<td></td>
</tr>
<tr>
<td>android.util.ArrayMap</td>
<td>isEmpty, size,</td>
<td>setValueAt, put*,</td>
</tr>
<tr>
<td></td>
<td>value*, contains*</td>
<td>remove*</td>
</tr>
<tr>
<td>android.util.SparseArray</td>
<td>size, value*</td>
<td>setValueAt, put*,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remove*, delete</td>
</tr>
</tbody>
</table>

Table 1: Example Templates Used for Matching Predicate Nodes and Condition Update Nodes

The rationale is that we can be lazy and delay to perform interprocedural analysis till when we need to match predicate nodes and condition update nodes to determine the feasibility of the callback sequences. When the analysis reaches the entry of the method \( m_i \), we advance the analysis to its caller \( m_{i−1} \) reported in the call chain. The analysis stops when all the local variables are replaced with static variables, calling objects, input parameters, constant or their expressions.

4.3 Computing Condition Update Nodes

The goal of including condition update nodes in the summary graph is to determine under a particular condition, which callbacks (either in the current API method or in the succeeding API methods) should be invoked. To determine which statements in the API method can be condition update nodes, we need to 1) obtain all the abstract variables used in the predicate nodes for all the Android API methods, and 2) find the assignments and the framework calls that can potentially change the values of these variables.

We analyzed all the abstract variables collected from the predicate nodes, and found they can be either a field of an object or the return of a method call. For the first category, we identify any assignment to the same field of the same type of objects as condition update nodes. For the second category, we identify a set of frequently invoked methods in the predicate nodes and define a set of templates for finding condition update nodes.

In Table 1, we show a few example templates we developed. Under Class, we show in which class the methods belong to. Under Predicate Nodes, we list the method calls frequently used in predicate nodes. The column provides a set of methods used to test the state of a collection. For example, isEmpty tests if a collection is empty. Under Condition Update Nodes, we list the types of statements that potentially can answer questions regarding the conditions in the predicate nodes. The column shows that these are the methods that have side effects on the collections. Note that the methods whose names end with an star (*) represent a set of similar methods whose names start with the same prefix given. For example, we have the method contains to test if a member belongs to a collection, and containsAll tests if a set of values belong to a collection. We represent such calls using contain*.

4.4 Generating Summary Graphs

In the first three phases, we mark all the identified call-back nodes, predicate nodes and condition update nodes on the ICFG of the API method. To add edges between the marked nodes and generate the summary graph, we traverse the ICFG and determine the reachability between the marked nodes in the ICFG with the goal that the final summary graph should keep the original control flow between
the marked nodes.

Algorithm 2 takes an input the ICFG of the API method with the three types of marked nodes and generates a summary graph for the API method. The worklist at line 2 stores a pair of nodes, \( q \) is the marked node, and \( n \) is the node encountered during the traversal of the ICFG. This pair of nodes are always reachable from each other on the ICFG, as the two nodes initially are the same node (see line 3), and when creating a new worklist element at line 12, the successors of \( n \) are replaced, which are still able to reach \( q \). At lines 7 and 8, when we find that \( n \) is a marked node, an edge is added between \( n \) and \( q \).

A key function \texttt{Succ} at line 12 handles the challenges of interprocedural analysis. If \( n \) is the exit of the current method, the successors are located at the next statement of its call sites. If \( n \) is the call site, the successors can be found at the entry of its callees (only the ones contain marked nodes). A special case when a query reaches \texttt{sendMessage} in the ICFG, we find its successors in the inline callbacks at the call site of \texttt{sendMessage} (see Section 4.1).

**ALGORITHM 2: Generating the Summary Graph**

\begin{verbatim}
1 set SG to {}
2 set worklist to {}
3 set n0 = Entry(icfg); q = n0 \ // n0 is also the entry for summary
4 add (n0, q) to worklist
5 while worklist \( \neq \{\} \) do
6     remove pair \((n, q)\) from worklist
7     if \( n \) is the marked node or \( n \) is the exit of icfg then
8         add edge \((q, n)\) to SG
9         newq = n
10     else newq = q;
11     foreach \( s \in \text{Succ}(n, icfg) \) do add \((s, newq)\) to worklist;
12 end
13 return SG
\end{verbatim}

5. APPLY SUMMARY

In this section, we show how we use generated summaries to construct the CFGs for the Android apps and also detect the infeasible callback sequences on the CFGs.

5.1 Constructing CFGs for Android Apps

The goal here is to sequence callbacks implemented in the Android apps. As we mentioned in the introduction, there are two major factors that determine the control flow of the app. Here, our focus is to sequence callbacks related to the Android API calls in the app. The functionalities of these callbacks include, but are not limited to, the Android lifecycles and component interactions.

To construct the app’s CFG, we start at some top level function. It can be, for example, a handler for a GUI event. We first build the ICFG for this callback. We then traverse the ICFG, and when an Android API call is encountered, we add edges to connect the call site of the API call to the entry of the summary, and also from the exit of the summary back to the next statement after the call site. Next, we identify the implementation of the callbacks listed in the summary. To do so, we perform a pointer analysis for the app to identify all the possible types that the calling object of the API call and the actual parameters the API call can have at the API call site. In case the parameter is an \texttt{Intent}, we resolve it to a set of possible component types defined in the app. Based on the types, we find the implementation of the callbacks invoked in the API methods at the call site.

![Figure 6: CFG for HostListActivity.onStart](image)

In Figure 6, we show the CFG we built for \texttt{HostListActivity.onStart} in Figure 1. At node 3, \texttt{onStop} invokes \texttt{unbindService}. We construct the edges to connect the summary. In the summary, \texttt{onUnbind} and \texttt{onDestroy} are called, and the calling object of the callbacks is from the input parameter \texttt{connect} (which can be resolved to TrackRecordingService). We thus replace the callbacks in the summary using the implementations from TrackRecordingService.onUnbind and TrackRecordingService.onDestroy.

Using a similar approach, we also can compute a CFG for \texttt{HotListActivity.onStart} shown in Figure 1. Following the lifecycle of \texttt{Activity}, we can add the CFG of \texttt{onStart} as a predecessor of \texttt{onStop} to create a more complete CFG for the app. On this complete CFG, we can then find program paths along which the static variable \texttt{started} was set to true in HotListActivity.onStart, and as a result TrackRecordingService.onDestroy is never called; the no-sleep bug discussed in Figure 1 can be found. In the future, we plan to obtain further constraints on how the external events should be ordered, e.g., through analyzing the GUI, and compose the CFGs.

5.2 Detecting Infeasible Paths

To apply program analysis on the CFGs, we can use predicated nodes and condition update nodes in the summaries to prune the infeasible paths related to callback sequences. Such infeasible paths can be useful to reduce the number of false positives in static analysis and help better estimate the coverage of testing.

To compute the infeasible paths on the CFGs of the apps, we implemented a demand-driven branch correlation algorithm [7]. We raise a query at each predicate node in the summaries. We propagate this query backwards along the paths of the CFG. The query can be resolved at the condition update node within the same API method summary.
or in a different API method summary. The modification of the algorithm here is that the query contains the abstract variables obtained from the predicate nodes, and we need to use the information from the condition update nodes, e.g., the value obtained through the assignments or the method templates defined in Table 1, for resolutions.

6. EXPERIMENTAL RESULTS

The goals of our experiments are to show that 1) the summaries are compact; 2) the summaries can be computed with practical precision and scalability; and 3) the summaries are useful for control flow analysis of Android apps.

6.1 Experimental Setup

We implemented Lithium using Soot [21] both for summarizing the Android API methods and for computing the apps’ CFGs. To summarize the Android API methods, we used as input the byte code of the Android framework 5.1 implementation and applied Spark [13] to build the call graphs for the API methods. To use Spark, we built a dummy main function which contains the calls to each of the Android API method analyzed. To analyze apps, we used the .apk files as input and applied Dexpler [5] to convert them to Soot Jimple representation. We built the call graph for each callback in the app using Class Hierarchy Analysis (CHA).

To perform the experiments, we first evaluated close to 1000 Android apps from the Google Play Market and F-Droid [1]. Through analyzing the usage of the Android API methods in these apps, we found 100 most frequently invoked Android APIs and generated summaries for them. We then selected 15 random Android apps from the F-Droid repository and constructed the apps’ CFGs using the generated summaries. We also generated dynamic traces through manual and random testing (Monkey) to determine whether the paths in the apps’ CFGs can be found in real execution traces.

All of our experiments were run using a virtual machine (VM) with 4 cores and 40GB of memory. The VM runs on a machine with 16 cores of Quad Core AMD Operton 6204. We use a 64-bit JVM with a maximum heap size of 15GB. In the following, we provide the detailed experimental results.

6.2 Compactness of the Summaries

For all the 100 Android API methods analyzed, we counted the number of nodes in the ICFG of the API method as well as the ones in the summary graph. By comparing the two, we derived that on average, the summary graphs contain 98% less nodes than ICFGs. We also counted the number of callback, predicate and condition update nodes for each summary graph. We found that on average, each summary contains 101 callbacks, and it contains 3 times more predicate nodes than the callback nodes. Based on the total number of boolean expressions used in the predicate nodes and the total number of callback nodes, we computed that each callback node is guarded by 20 conditions on average, implying that the callback invocation logic can be complex.

We sorted the size of the ICFGs, summary graphs, callback nodes, predication nodes and condition update nodes for the 100 API methods analyzed, and we report the minimum, average and maximum number of nodes for each type in rows min, ave and max, under ICFG, Summary, Callback, P-Node and U-Node respectively, shown in Table 2. The results show that the size of the summary graphs ranges from 0 to 4553 nodes with an average of 482 nodes.

Table 2: No. of Nodes in ICFGs and Summaries

<table>
<thead>
<tr>
<th></th>
<th>ICFG</th>
<th>Summary</th>
<th>Callback</th>
<th>P-Node</th>
<th>U-Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ave</td>
<td>172235</td>
<td>482</td>
<td>101</td>
<td>330</td>
<td>49</td>
</tr>
<tr>
<td>max</td>
<td>341830</td>
<td>4553</td>
<td>1327</td>
<td>3334</td>
<td>305</td>
</tr>
</tbody>
</table>

6.3 Correctness of the Summaries

In this section, we report our studies on the correctness of the summaries. In the first study, we identified a total of 7 small size API methods for manual analysis. Our manual inspection focuses to find all the callback, predicate and condition update nodes that the summary graphs should have. We then compare this ground truth with the summary generated by Lithium to understand the capabilities of our tool.

In Table 3, under Match, we report the number of nodes in the summary graph that match the ones in the ground truth. Under Miss, we show the number of nodes that are in the ground truth but missed by our summary graph (false negatives). Under Additional, we list the number of nodes reported in the summary graph but not present in the ground truth (false positives). The data show that the precision of the tool is 100% (we do not report any false positives), and the recall is 69% (we do miss some nodes). We missed the nodes in the summary graph, as in this experiment we terminate the analysis for the callback nodes whose call chains are longer than 20 methods. We had done experiments to tune the parameter to allow longer call chains to be analyzed. Although we observed the improved recall, the tool ran slower and generated more false positives. In this case, the analysis traversed deeper in the ICFGs of the API methods and encountered more virtual call sites.

Table 3: Compare to Manually Identified Ground Truth

<table>
<thead>
<tr>
<th></th>
<th>Match</th>
<th>Miss</th>
<th>Additional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback</td>
<td>20</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>P-Node</td>
<td>49</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>U-Node</td>
<td>30</td>
<td>32</td>
<td>0</td>
</tr>
</tbody>
</table>

We also verify the correctness of the object receivers identified for the callback nodes. For the 7 API methods under study, we report a total of 20 callbacks. We correctly identified the object receivers for the 13 callbacks. In our method, we use the type information to determine whether an object receiver should be the calling object of the API or the input parameters. The imprecision implies that it is not sufficient by only matching the types. As we mentioned in Section 3.2, the object receiver can also be assigned via static variables, which can be modified by the app through Android API calls. The 7 incorrect object receivers in our results belong to this case.

Besides comparing with the ground truth for the 7 API methods, in the second study, we built partial ground truth for additional 4 API methods whose ICFGs are large and hard to be analyzed manually. We used the documentation to identify the callbacks executed in these API methods. We determined whether the callbacks were in the summary graphs. In addition, we randomly picked a set of callback

---

nodes from the summary graphs and navigated to the corresponding part of the ICFG of the API method to verify manually if the callback nodes were false positives. Our results report 187 matched call back nodes, 1 false negative and 135 false positives. We found that 127 false positives are due to call graph imprecision, and 8 false positives are the cases where the object receiver of the callback is from some internal object of the framework, i.e., they are not the callbacks implemented in the apps.

6.4 Scalability of Generating the Summaries

In Figure 7a, we show the time used to build the summary graphs for all the 100 API methods in ascending order. For more than 60% of the methods the time used is in few seconds, and more than 90% of the methods were summarized under 4000 seconds. The overall performance of Lithium was considerable fast with a median of 13 seconds and an average of 744 seconds per API method. One of the aspects that contributes to the scalability was the demand-driven analysis that identified the call chains of the callbacks and discarded irrelevant methods for the expensive second phase.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7}
\caption{(a) Time in Ascending Order (b) Time vs Size of Summary Graph}
\end{figure}

In Figure 7b, we plot the performance with regard to the size of the summary graphs. The plot shows an almost linear correlation between the total time used and the size of the summary. The most performance-intensive task is to perform backward symbolic substitution to resolve local variables to abstract variables.

6.5 Usefulness of the Summaries

In this section, we report our results on control flow analysis of Android apps using the summaries. In Table 4 under App, we list 15 apps under study. Under Callback, we report a total number of callbacks for the app. Under API calls, we show the number of Android API calls we modeled for the apps. Under Apps’ CFGs, we report the size of the CFGs we built for the apps in terms of number of callbacks. In this experiment, we view any callbacks that do not have a caller in the app as top level functions. Each of such callback produces a CFG. For all such CFGs an app generates, we report the minimum, average and maximum number of callbacks integrated into the CFG. We traverse each CFG to find the longest paths in terms of the callbacks, shown under Longest Path. Finally, we show the time used (in seconds) to build the CFGs from the summaries under Time (s).

The construction of the apps’ CFGs is finished on average 10 seconds. The results show that by modeling API calls using the summaries, we are able to connect callbacks to the CFGs. The longest callback sequences reach up to 10 for org.openintents.filemanager. Note that we used the summaries for the 100 most frequently used Android API method, and not all the API calls are modeled. We expect that the larger CFGs and the longer callback sequences would be generated if we plug in more summaries.

In Table 5, we show the comparison of the traces generated from testing and the paths (the callback sequences) generated from the CFGs. Column Traces report the data obtained from analyzing the traces. Under covered-c, we report the number and the percentage of the callbacks covered during testing. Under total, we report the total number of traces we generated during testing. Under covered-p, we report how many of these traces contain the callback sequences obtained from the CFGs. Our results show that on average we covered 49% of the callbacks during testing. We covered 96 out of 97 traces from the 15 apps and just missed one trace in org.openintents.filemanager. We inspected the case and found that the trace does not reach any API calls we modeled. The longest path found in the traces consists of 5 callbacks.

Column Paths report the analysis of paths. Under total, we show the total number of acyclic paths generated from all the CFGs in the app. Under infeasible, in traces and unclassified, we report the distributions of the paths in three categories, the paths known to be infeasible, the paths found in the traces, and the paths not yet to be determined. The unclassified paths may be infeasible but not yet detected by our tool, as detecting infeasible paths is undecidable, or they are feasible but not covered by our current test cases.

Our results show that we have identified infeasible paths using the CFGs built based on the summaries, and the predicate nodes and condition update nodes thus are useful to determine valid callback sequences. Among 9 out of 15 apps, we are able to confirm that more than 50% of the paths are either infeasible or covered by the traces. This demonstrates that the paths generated on the CFGs are valid. For app such as org.openintents.filemanager that reports the low path coverage, we found that the tests did not reach the API calls we modeled.

7. RELATED WORK

Our work is closely related to the following three areas.

Interprocedural Analysis of Mobile Apps: Most of the current tools for interprocedural analysis of Android apps use manual models to define control flow between callbacks [4] [14] [24] [10] [16] [27]. These manual models typically define control flow constraints for a subset of the callbacks executed in the Android API methods. For instance, Yang et al. [24] [25] analyzed callbacks’ control flow graphs to identify a sequence of Android API calls, and used manual models to obtain the first callback invoked in the API method. Blackshear et al. [6] introduced the jumping framework, which allows to identify inter-callback control flow constraints via data dependencies on top of lifecycle of components. Using these models to construct CFGs for Android apps can miss important callbacks, which can affect the precision of interprocedural analyses for Android apps. Moreover, manual models need to be updated as newer versions of the Android APIs are released. Our automatic approach can generate summaries in minutes and can easily be used to generate summaries for new releases.

Pre-computed Summaries: EdgeMiner [8] is the closest work to our approach. Their goal is to map Android API methods to callbacks invoked in the methods. However, they do not report the control flow between callbacks. Our results show that Android API methods have complex im-
implement and generate different sequences of callbacks under different contexts. We are able to identify such fine-grained information, including the order of callbacks and the conditions under which the callbacks will be invoked.

Pre-computed summaries to solve other types of program analysis problems have been defined. Clapp et al. [9] mine information flow specifications for the Android API methods. Their specifications identify how values from the app can be tainted in the Android API methods. However, their summaries do not include any control flow information about callbacks. For data flow summaries of complete libraries, Routnev et al. [19] [20] extended Sharir and Pnueli’s [18]’ work to summarize library methods for whole-program analysis. Their extension created placeholders for the callback methods identified in the library. Their approach requires a whole-program analysis on the application to identify the receiver of the callbacks in the summaries. The summaries also do not keep control flow information which is vital to determine all the possible sequence of callbacks at a call site of an Android API method.

Ali et al. [2, 3] focus on analyzing the application code using a single summary node to represent all library methods and less conservative assumptions (separate compilation assumption) about how the library code can interact with application code. However, since Android apps depend most its entire execution on the Android Framework, with a great number of callbacks and framework method calls made by the app, this assumptions their technique would be too imprecise to analyze Android apps.

**Control Flow Analysis in the Presence of Callbacks:** Zhang et al. [28] proposed an algorithm to resolve library callbacks to improve application call graphs. Their approach uses a data reachability analysis to reduce spurious callback edges for every library call site. This can help to improve the precision when resolving the calling objects of the callbacks; however, they require to analyze the entire library at each call site of the API methods, which can make the analysis intractable even for small apps [12].

### 8. CONCLUSIONS AND FUTURE WORK

This paper presents the representation and computation of the Android API summaries with respect to the control flow of callbacks. We demonstrate that the summaries generated are compact, reasonably precise and useful for control flow analysis of Android apps such as constructing CFGs and detecting infeasible callback sequences. In the future, we plan to generate API summaries for the entire Android framework and also integrate the GUI analysis to compose the CFGs we generated.

---

**Table 4: Construct Apps’ CFGs Using Summary Graphs**

<table>
<thead>
<tr>
<th>App</th>
<th>Callbacks</th>
<th>API Calls</th>
<th>Apps’ CFGs</th>
<th>Longest Path</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.blippex.app</td>
<td>16</td>
<td>7</td>
<td>1 1 1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>net.sourceforge.andsys</td>
<td>21</td>
<td>17</td>
<td>2 2 2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>com.darknessmap</td>
<td>25</td>
<td>9</td>
<td>1 2 3</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>com.example.android.contactslist</td>
<td>41</td>
<td>7</td>
<td>1 8 16</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>de.onyxbits.remotekeyboard</td>
<td>41</td>
<td>10</td>
<td>1 2 3</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>info.staticfree.SuperGenPass</td>
<td>50</td>
<td>10</td>
<td>1 4 8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>com.markuspage.android.atimetracker</td>
<td>57</td>
<td>25</td>
<td>3 3 3</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>aarddict.android</td>
<td>57</td>
<td>9</td>
<td>2 3 4</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>de.ub0r.android.websms</td>
<td>69</td>
<td>21</td>
<td>1 1 2</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>com.google.exming.client.android</td>
<td>76</td>
<td>48</td>
<td>1 3 6</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>org.openintents.filemanager</td>
<td>164</td>
<td>17</td>
<td>1 4 8</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>org.connectbot</td>
<td>169</td>
<td>25</td>
<td>1 3 6</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>a2dp.Vol</td>
<td>171</td>
<td>35</td>
<td>3 7 17</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>com.evancharlton.mileage</td>
<td>204</td>
<td>48</td>
<td>1 1 2</td>
<td>2</td>
<td>13</td>
</tr>
</tbody>
</table>

**Table 5: Compare Dynamic Traces and Static Paths**

<table>
<thead>
<tr>
<th>App</th>
<th>Traces</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>covered-c</td>
<td>total</td>
</tr>
<tr>
<td>com.blippex.app</td>
<td>15 (94%)</td>
<td>11</td>
</tr>
<tr>
<td>net.sourceforge.andsys</td>
<td>17 (81%)</td>
<td>6</td>
</tr>
<tr>
<td>com.darknessmap</td>
<td>16 (64%)</td>
<td>6</td>
</tr>
<tr>
<td>com.example.android.contactslist</td>
<td>33 (80%)</td>
<td>6</td>
</tr>
<tr>
<td>de.onyxbits.remotekeyboard</td>
<td>27 (66%)</td>
<td>6</td>
</tr>
<tr>
<td>info.staticfree.SuperGenPass</td>
<td>32 (64%)</td>
<td>6</td>
</tr>
<tr>
<td>com.markuspage.android.atimetracker</td>
<td>27 (47%)</td>
<td>6</td>
</tr>
<tr>
<td>aarddict.android</td>
<td>37 (65%)</td>
<td>6</td>
</tr>
<tr>
<td>de.ub0r.android.websms</td>
<td>30 (43%)</td>
<td>6</td>
</tr>
<tr>
<td>com.google.exming.client.android</td>
<td>43 (57%)</td>
<td>6</td>
</tr>
<tr>
<td>org.openintents.filemanager</td>
<td>58 (35%)</td>
<td>10</td>
</tr>
<tr>
<td>org.connectbot</td>
<td>70 (41%)</td>
<td>6</td>
</tr>
<tr>
<td>a2dp.Vol</td>
<td>81 (47%)</td>
<td>10</td>
</tr>
<tr>
<td>com.evancharlton.mileage</td>
<td>85 (40%)</td>
<td>6</td>
</tr>
</tbody>
</table>

---
9. REFERENCES


