Asynchrony-Aware Static Analysis of Android Applications

Ashish Mishra
Computer Science and Automation,
Indian Institute of Science, India.
ashishmishra@csa.iisc.ernet.in

Aditya Kanade
Computer Science and Automation,
Indian Institute of Science, India.
kanade@csa.iisc.ernet.in

Y. N. Srikant
Computer Science and Automation,
Indian Institute of Science, India.
srikant@csa.iisc.ernet.in

***Abstract*—Software applications developed for the Android platform are very popular. Due to this, static analysis of these applications has received a lot of attention recently. An Android application is essentially an asynchronous, event-driven program. The Android framework manages the state of the application by invoking callbacks, called lifecycle callbacks, in pre-defined orders. Unfortunately, the existing static analysis techniques treat the callbacks synchronously. Additionally, they do not model all possible orderings of lifecycle callbacks. These may result in unsound analysis results. In this work, we present a precise representation of control flow of Android applications called Android inter-component control flow graph (AICCFG). In this representation, the asynchronous nature of the callbacks is modeled accurately. Further, all interleavings of callbacks of different components of an Android application are modeled in AICCFG. We use this representation to design a typestate analysis of Android applications. Android applications use a rich set of resources such as camera and media player whose safe usage is governed by some state machines. Using the typestate analysis, we can verify whether an application uses a resource safely or not. We have implemented the construction of AICCFG and the typestate analysis in the Soot framework. We have also implemented a variant of typestate analysis which uses the unsound control flow model used commonly in the literature. To compare our AICCFG based analysis with this, we present a benchmark of Android applications called AsyncBench. It comprises applications that use various resources in both safe and unsafe manner. The experiments over this benchmark demonstrate the benefits of our more precise control flow model and the typestate analysis.**

I. Introduction

Android is the heaviest used and rapidly growing mobile system with more than 2 million paid and free applications on Google's Play store currently [2]. Many of these applications are rigged with benign and harmful bugs and vulnerabilities. Unfortunately, analyzing these applications is a herculean task. There are numerous static [4], [9], [10], [15] and dynamic program analysis [5] works which try to find bugs in Android applications.

Android applications are typically made up of four types of elements called *components*. These are *Activity, Service, BroadcastReceiver, ContentProvider*. Each of these components have sets of methods or callbacks which are called by the Android framework on various user or system events. This differentiates these applications from normal Java programs and makes the analysis of these applications challenging. The control and data flow in Android applications is further complicated due to extensive use of *asynchronous calls*. These calls makes it convenient and efficient to model events and the interactions between various *components*, and permits the Android framework to interleave the execution of several event handlers and other callback methods. Although, many of the static analysis works [4], [9], [10], [15] for Android applications model these interactions with the framework, none of these works correctly model the asynchronous behavior of the event handling mechanism and inter component communications in Android. Moreover, they also lack a correct modeling of the framework's interaction with the application components called *component lifecycle*. These limitations bring unsoundness and imprecision in these analyses.

We in this work present the first model of asynchronous control flow semantics in Android applications and a sound and precise model of *component lifecycle* of Android applications. We explicitly model all the asynchronous calls and framework callbacks by creating an *application environment model* for the application and create a precise lifecycle state machine for each component. We define an Android inter component control flow graph (AICCFG) which integrates these two to soundly model all possible control flow interactions between various units. We present an algorithm to generate such an AICCFG for the application. We implement and solve a client typestate [13] analysis to verify Android resource API usages as an instance of the asynchronous interprocedural finite distributive subset (AIFDS) problem [8] and compare the results of our analysis against an asynchrony-unaware version of the analysis on the program representation used by other state-of-the-art analyses for Android applications. We demonstrate empirically that these asynchrony-unaware techniques are both unsound and imprecise with respect to our asynchrony-aware analysis. We present a set of 19 benchmark applications in five different resource categories, called AsyncBench. This comprises of test applications that use various resources in both safe and unsafe manner. A sound verification of these test applications requires an asynchrony-aware modeling of the applications. We were able to verify all the typestate violations in these applications with a precision of 78% and recall of 100%. The comparison of these results against the asynchrony-unaware analysis over unsound program representation used

by other works clearly demonstrates the soundness and effectiveness of our application modeling and analysis.

The major contributions of our work are as following-

- We present the first sound model of asynchronous control flow and component lifecycle semantics in Android applications.
- We present an intermediate representation of Android applications called AICCFG based on the above model.
- We develop a typestate analysis over AICCFG and find resource API violations for a variety of resource types.
- We present a set of benchmark applications called AsyncBench, comprising of applications whose analysis requires a sound modeling of asynchronous and lifecycle semantics of Android applications.
- We finally compare our typestate analysis against an asynchrony-unaware analysis and demonstrate the benefits of our modeling and analysis against these.

II. Motivating Example

Consider the example application FileReader in Figure 1. The application has two activities *SelectActivity* and *ReadFileActivity*. The application allows the user to select a file and open the file in ReadFileActivity. The FileReader object, line 2, is a global static reference which is accessible through both the components. A typestate [13] analysis checks the possible runtime states of a resource object against a given typestate property finite automaton.

To verify a typestate property "The application never reads from a closed FileReader", the analysis needs to verify and guarantee that the FileReader.read() is never called when the FileReader has been closed using FileReader.close() and not re-opened again. The analysis starts with the SelectActivity's onCreate() where the FileReader is instantiated and File is read at lines 8 and 9 respectively. This read operation is safe as the FileReader instantiation switches the object to open state. Line 12 makes an inter-component comunication (ICC) call to the ReadFileActivity. Now the correct semantics of Android on such an ICC call is as follows-

- The ICC call is treated as an asynchronous call. An *asynchronous* call is a method call which instead of being dispatched and executed at the call site, is stored in a task queue (associated with the thread) and is dispatched for execution at some later time.
- The lifecycle callback onCreate has an atomic execution semantics, thus the control stays in onCreate and the next instruction is scheduled for execution.
- Once the onCreate finishes execution, the Android framework (the *ActivityManagerService*) schedules onStart, onResume and onPause callbacks in that order (if present), before the control could escape to another component.
- Once the onResume finishes execution and since the application is not overriding onPause, the call to the target Activity at line 12 is scheduled for dispatch , and ReadFileActivity's onCreate is called.

There are three major features of Android framework which govern the above semantics. (i)- All the component callback

```
1  class SelectActivity extends ActionBarActivity{
2   public static FileReader myFileReader;
3   protected void onCreate(Bundle savedInstanceState){
4     ...
5   setContentView(R.layout.activity_select);
6   try{
7     String filePath = this.getFilesDir() + '/' + "exFile.txt";
8     myFileReader = new FileReader(...);
9     int data = myFileReader.read();
10    Intent targetIntent = new Intent(this, ReadFileActivity.class);
11    // asynchronous call to the ReadFileActivity
12    startActivity(targetIntent);
13    }catch (FileNotFoundException e){
14      e.printStackTrace();
15    }
16  }
17  protected void onStart(){
18    super.onStart();
19  }
20
21  protected void onResume(){
22    super.onResume();
23    Log.d(TAG, "onResume");
24    try{
25      myFileReader.close();
26    }catch(IOException e){
27      e.printStackTrace();
28    }
29  }
30 }
```

```
31 class ReadFileActivity extends ActionBarActivity {
32  protected void onPause(){
33    super.onPause();
34  }
35  ...
36  protected void onStop(){
37    super.onStop();
38    try{
39      ...
40      int data = SelectActivity.myFileReader.read();
41      Log.d("ReadFileActivity", "data " +data);
42    }catch (IOException e){
43      e.printStackTrace();
44    }
45  }
46 }
```

Fig. 1. FileReader Application

methods have atomic execution, thus they finish executing before another callback of the same or different component could be scheduled. (ii)- Each component has a control flow protocol (lifecycle), a set of ordering relations which governs the calling order and control flow between callbacks in and across components. (iii)- The ICC calls like the one at line 12 are asynchronous in semantics and hence the actual dispatch of such calls is separated in time and is managed by the framework. In a typical synchronous call the call is executed at the call site and the caller blocks itself and waits for the callee to return. Compared to this, since the caller is not blocked in an asynchronous call, the state of the system can possibly change between the time of call and the time of dispatch of the callee for execution. Thus a sound static analysis should correctly model this control flow semantics else the analysis

results will be unsound.

Following the above semantics, the control reaches line 40 only after executing the call to myFileReader.close() at line 25 and there are no more *open* operations between 25 and 40. Hence the call to myFileReader.read() happens on a closed object and is a typestate property violation. Equipped with the correct control flow semantics, let us see how the state-of-the-art static analysis works for Android handle such a scenario. All the static analysis works for Android [9], [15] capable of handling ICCs treat call at line 12 synchronously, executing it right at the call site, thus blocking and passing control to line 31 and then to line 40. Since the global myFileReader object is in *open* state before line 12, and there is no myFileReader.close() operation between line 31 to 40, the object is in *open* state at line 40. Thus the call to myFileReader.read() is marked as a valid operation and they miss capturing the property violation.

Existing static analysis works for Android applications fail to correctly model the asynchronous semantics of ICCs and callbacks handling and lack precision and soundness in modeling the lifecycle semantics of components. In this work, we provide a detailed model of this semantics. We model this semantics as an Android application environment and present an intermediate program representation called the Android inter component control flow graph (AICCFG). The AICCFG soundly captures the control flow in Android applications. We show the effectiveness of our AICCFG by developing a client typestate analysis as an AIFDS [8] instance using this as program input. We effectively capture this and other typestate violations and verify important typestate properties over a variety of applications.

III. Android Application Environment Modeling And AICCFG Creation

Android applications execute inside the Android framework which asynchronously interacts with the application components on various user and system events. There exists a system dispatcher, the ActivityManagerService class of the framework which schedules an event-handler or a component lifecycle callback method based on the event fired and the state of the application. To soundly model the control flow of Android applications we model the framework over-approximating its asynchronous semantics and dispatch logic.

Our Android application environment model is an asynchronous control flow graph for the whole application and is called Android inter component control flow graph (AICCFG). This modeling contains three major structures namely the ambiance, an init codeblock for each component instance and a set of intra and interprocedural edges. A node of the AICCFG models a control location. The edges represent the sequence of program statements or simply a flow of control between nodes.

The ambiance is the core of the AICCFG, modeling the asynchronous call semantics of applications. Figure 2 shows the ambiance (middle) and lifecycle statemachine of two Activities (on sides) for the example FileReader application.

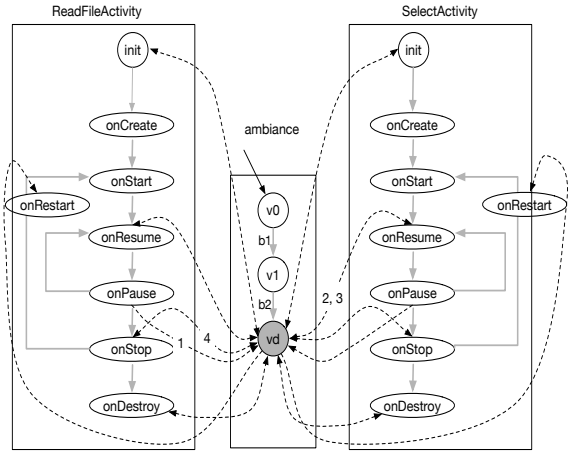

Fig. 2. Ambiance and lifecycle state machines for FileReader Application

There are two kinds of edges- the dashed black edges representing the asynchronous call dispatch and return edges, and solid gray edges representing other inter and intraprocedural control flows.

A. Ambiance

Consider Figure 2, the ambiance (middle) comprises of three nodes v0, v1 and vd, and two code blocks b1 and b2, where:

- **Block b1**- The public and launcher components in an Android application are instantiated by Android framework when needed (e.g. launching of an application from the home screen creates an instance of the launcher component). The ambiance, statically instantiates each of these components explicitly. Block b1 in the figure represent such an instantiation code block.
- **Block b2**- For each inter component communication (ICC), explicit or implicit, the Android framework creates a new instance of the target component at runtime if this is the first ICC call to the target. Corresponding to this, the ambiance statically instantiates each possible targets of all the ICCs in the application. Block b2 in the figure denotes this instantiation code block.
- **Node vd**- Android framework asynchronously calls each component instance based on various user and system events, like launching of the application, pushing back button on the home screen, etc. The ambiance needs to soundly model these asynchronous calls and callbacks. Node vd in the figure, is called the dispatch node and is a special control location, which statically models these asynchronous calls and returns from the framework.

The lifecycle initialization method called init() in Figure 2 models the calls to a sequence of callback methods once the component is instantiated. For example, once an Activity is instantiated, its onCreate, onStart, onResume and onPause methods are invoked by the framework in that order, before the control can switch to another component. Edges from vd to SelectActivity's onCreate and other callback methods model asynchronous dispatch (and return where ever allowed

by lifecycle rules) edges to various lifecycle callback methods from the framework. Note that, these asynchronous call and

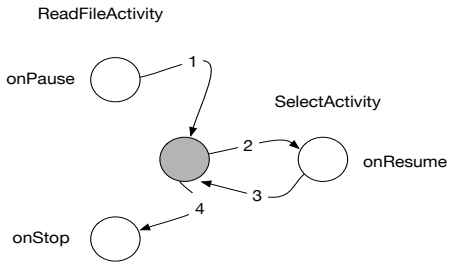

Fig. 3. An ICC control flow interleaving for FileReader application

return edges soundly model all possible interleavings between the callback methods of different components. Missing them may lead to unsoundness in the AICCFG, e.g., Figure 3 shows an execution sequence which requires such a correct modeling of these inter component flows. The execution sequence (1-2-3-4) in Figure 3 can be traced in Figure 2 and is shown by numbered edges (1-2-3-4). Other works modeling lifecycle callbacks as synchronous calls are unsound and miss such an interleaved execution path.

B. Android Inter Component Control Flow Graph, AICCFG

AICCFG is an asynchronous control flow graph $G_* = (V_*, E_*)$, for the whole Android application modeling the asynchronous calls, event handling and lifecycle callbacks invoked by the Android framework. The graph is based on the asynchronous program representation described in [8] extended to model the control flows specific to Android applications. In this section we formally define the AICCFG, illustrating its features using Figure 4 which shows a part of the AICCFG generated for the example FileReader application. The figure is a detailed version of Figure 2 and shows a more fine grained view of the control flows.

The graph in Figure 4 has nodes representing the control locations, with double edged circles representing the terminal locations for a method or callback. The graph can be divided into three subgraphs for the ReadFileActivity, SelectActivity and the ambiance in the middle. There are three major types of edges in the figure-

- intraprocedural gray edges, connecting successive statements (e.g. v3 $\rightarrow$ v4).
- interprocedural solid black edges, representing synchronous call and return (e.g. v2 $\rightarrow$ v3 or v2 $\rightarrow$ v6).
- and interprocedural dashed black edges, representing asynchronous dispatch edges from special dispatch node vd to certain lifecycle callback methods and lifecycle initialization method init() and their corresponding return edges (e.g. vd $\rightarrow$ v2 and v2 $\rightarrow$ vd).

To define the AICCFG, we begin with defining the substructures needed to define the AICCFG. Figure 4 is used to illustrate these definitions.

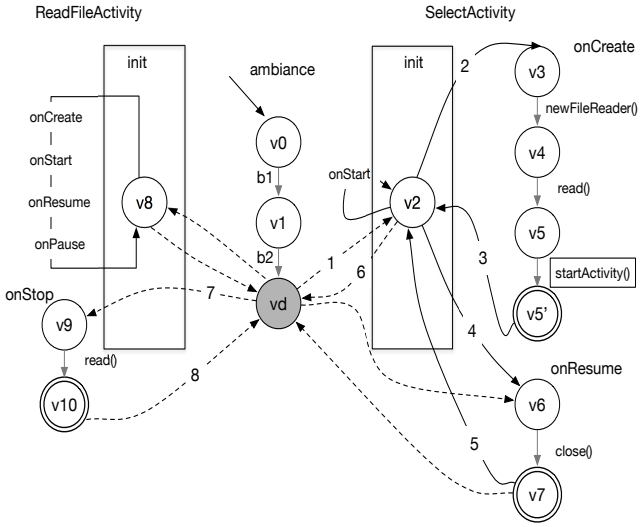

Fig. 4. Part of AICCFG for FileReader application

*a) **Lifecycle Callback Control Flow Graph**:* Each lifecycle callback method like onCreate, onStart, etc. is a Java method which is invoked by the framework on certain events and is executed atomically. The lifecycle callback control flow graph (LCCFG) defines the control flow graph for such a lifecycle callback method. It is a asynchronous control flow graph, $G_{lc} = (V_{lc}, E_{lc})$, where V_{lc} is the set of control locations in the callback method and E_{lc} is a set of normal control flow edges (e.g. edges like v3 $\rightarrow$ v4 , v4 $\rightarrow$ v5) along with two types of intraprocedural edges between nodes corresponding to following types of program statements-

- an intraprocedural edge from a call site to the return site corresponding to each synchronous call to a method *m* in the callback method. For example, edge (v4 $\rightarrow$ v5) with label *read* in the figure.
- an intraprocedural edge from a call site to the return site corresponding to each asynchronous call to a method or a component. For example, edge (v5 $\rightarrow$ v5') with a label *startActivity* in the figure.

Since an asynchronous call is stored in the task queue and not dispatched directly and the control stays with the caller the above intraprocedural edges corresponding to asynchronous calls correctly model the delayed dispatch semantics of asynchronous calls. These asynchronous pending calls will be dispatched later from the dispatch node vd.

*b) **Android Component Control Flow Graph**:* Android framework enforces the lifecycle rules associated with each component of an Android application. This defines a control flow semantics associated with a component. An Android component control flow graph (ACCFG), models the synchronous component of the control flows in a component. It is a synchronous, interprocedural control flow graph $G_c = (V_c, E_c)$, which comprises of a set of LCCFG, one for each callback defined in the component along with the init method. The nodes V_c is a union of the nodes V_{lc} for each LCCFG, and

edges E_c is a union of the edges of each LCCFG, along with a set of synchronous interprocedural edges defined as follows.- For each intraprocedural edge in a LCCFG, corresponding to a synchronous call, the ACCFG has -

- an interprocedural synchronous call edge from callsite to the callee's entry node corresponding to each synchronous call in the method.
- an interprocedural synchronous return edge from the exit of the callee to the return site.

Note that the ACCFG misses the asynchronous dispatch and return edges which will be modeled by the AICCFG defined next.

*c) **Android Inter Component Control Flow Graph**:* An AICCFG is an asynchronous control flow graph G_* = (V_*, E_*), for an Android application modeling the asynchronous calls, event handling and lifecycle callback invoked by the Android framework. Intuitively, it models the asynchronous dispatches corresponding to the asynchronous calls occurring in the application. An asynchronous dispatch is an interprocedural edge from the node vd (in the ambiance) to a method or a callback, for which there is a pending asynchronous call. For example, the edge (v5 $\rightarrow$ v5') labeled startActivity() adds a pending call to the ReadFileActivity's init method. This call will be dispatched later from the vd, represented by edge (vd $\rightarrow$ v8). Thus, the AICCFG integrates the ACCFGs and the ambiance together and models the correct asynchronous dispatches performed by the framework. The nodes V_* is a union of nodes for each of these elements and edges E_* is a union of edges for each ACCFG plus a set of following asynchronous dispatch edges corresponding to each pending asynchronous call in the application.

- An asynchronous dispatch edge to each init $\in V_c$ of the ACCFG set. For example edge (vd $\rightarrow$ v2) in Figure 4 represents such an edge from vd to SelectActivity's init method.
- An asynchronous return edge from the exit of init to vd, for example, the return edge from v2 and v8 to vd in the figure.
- Android framework can make asynchronous calls to certain lifecycle callback methods of a component. The AICCFG models these calls as asynchronous dispatch edges from vd to the entry of these lifecycle callback methods of each ACCFG instance. For example, edge (vd $\rightarrow$ v6) in Figure 4.
- Corresponding to each vd to entry of a method like (vd $\rightarrow$ v6), there is an interprocedural return edge from exit of callback, like (v7 $\rightarrow$ vd).

Notice that not all the paths in the AICCFG are valid execution paths of the application. For example, there is no possible execution sequence for the path (1-2-3-4-5-6-1-2-3-6), which calls the init for SelectActivity twice. This is due to the fact that, each asynchronous ICC (like startActivity in edge v5 $\rightarrow$ v5') corresponds to a single asynchronous call to target. Thus informally, the set of valid paths in AICCFG, include only those paths for which every asynchronous dispatch has a matching asynchronous call defined earlier in the path. We refer the reader to [8] for a more formal semantics of asynchronous procedure calls.

C. Ambiance and AICCFG Construction

Algorithm 1 presents an algorithm for constructing the AICCFG for a given application. An Android applications is compiled into *Dalvik* executables and packaged into an *.apk* file. The application also has a *manifest* file which provides essential information about the application to the Android system. Our algorithm takes the application's apk A, and the manifest M as input and emits the AICCFG, G_* = (V_*, E_*) as output. The algorithm's Main routine extracts the launchers and public components of the application from M, using auxiliary methods getLaunchers() and getPublicComps() at lines 3 and 4 respectively. The call to method getICCCalls(), at line 5 analyzes the application and the manifest and returns the set of inter component communication calls, iCCSet in the applications. The method call to getComponentCfgs() at line 6 generates an ACCFG for each public and launcher component extracted earlier. Line 7 makes a call to subroutine createAmbiance.

The createAmbiance function (lines 11-19) takes lists of public and launcher components as input and instantiates each of these components creating blocks b1 and b2 (similar to the blocks in Figure 4) at lines 14 and 15. It also takes the iCCSet as input and instantiates the target of each of these ICC calls and concatenates these to b2 at line 16. It creates a new dispatch node vd at line 17 and finally concatenates each of these to the empty ambiance at line 18, and returns the generated ambiance.

The createAICCFG subroutine takes the generated ambiance and a list of ACCFGs accfgs as input and returns the final G_* = (V_*, E_*) as output. It initializes the node and edge sets with empty sets (line 23) and extracts the vd from the ambiance (line 24). The outer while loop (lines 25-36) visits each ACCFG G_c = (V_c, E_c), and adds the nodes and edges to V_* and E_* respectively (lines 26-28). The inner while loop (lines 30-34), looks at each node $N_c \in V_c$, and creates an asynchronous dispatch edge (vd, N_c), iff N_c is an entry node of a callback (e.g. node v3 in Figure 4) and adds this edge to the edge set E_* (line 33). Else, it creates an asynchronous return edge (N_c, vd), iff N_c is an exit node of a callback (e.g. nodes v7 or v10 in Figure 4) and then adds this edge to the edge set E_* (line 37). The method returns the G_*, to the caller Main which finally return the generated AICCFG G_*.

To understand the soundness of our AICCFG against the Android environment models used by other static analysis works, consider Figure 5 which presents a simplified partial environment model generated by IccTA [9] for the same example FileReader application. AmanDroid [15] has a similar ICC and asynchrony-unaware semantics as modeled in this graph. The figure has nodes and edges defined similar to Figure 4, but lacks any asynchronous features (dispatch node and dispatch edges). Consider a path in Figure 4 with numbered edges (1-2-...-8). This is a possible valid execution path for some

Algorithm 1: Algorithm AICCFG construction

input : Application *Manifest* M and apk A.
output: AICCFG $G_* = (V_*, E_*)$ for the application.

```
1  Function Main()
2      begin
3          launchers ← getLaunchers(M);
4          publicComps ← getPublicComps(M);
5          iCCSet ← getICCCalls(M, A);
6          accfgs ← getComponentCfgs(launchers,
             publicComps);
7          ambiance ← createAmbiance(launchers,
             publicComps, iCCSet);
8          aiccfg ← createAICCFG(ambiance, accfgs);
9          return aiccfg;
10     end
11 Function createAmbiance(launchers, publicComps, iCCSet)
12     begin
13         ambiance ← ∅;
14         b1 ← instantiate(launchers);
15         b2 ← instantiate(publicComps);
16         b2 ← v1.concat(instantiate(iCCSet));
17         vd ← newNode(); /*Creates a new empty node */
18         ambiance ← ambiance.concat(b1, b2, vd);
19         return ambiance;
20     end
21 Function createAICCFG(ambiance, accfgs)
22     begin
23         V* ← ∅; E* ← ∅;
24         vd ← (ambiance.vd);
25         while accfgs has Gc = (Vc, Ec) do
26             Gc ← remove(accfgs);
27             V* ← V* ∪ Vc;
28             E* ← E* ∪ Ec;
29             while Vc has Nc do
30                 Nc ← remove(Vc);
31                 if Nc is a callback method's entry then
32                     (vd, Nc) ← createEdge(vd, Nc);
33                     E* ← E*∪ {(vd, Nc)};
34                 end
35                 else if Nc is a callback method's exit then
36                     (Nc, vd) ← createEdge(Nc, vd);
37                     E* ← E*∪ {(Nc, vd)};
38                 end
39             end
40         end
41         return G*;
42     end
```

run of the application. The set of possible paths in Figure 5 lacks such a path due to unsound modeling of asynchronous calls as synchronous. Their graph dispatches the ICC call startActivity() (edge v4 → v5) synchronously and blocks till the callee returns. Thus the call to read in ReadFileActivity's onResume() (v8 → v9) is reached before the object is closed in SelectActivity's onResume is called (v5 → v11). As evident from this example, this makes the analyses built over their model inherently unsound. In this case, the analysis will be missing a possible typestate violation at (v8 → v9).

Another source of unsoundness comes from modeling component lifecycle as a synchronous control flow block (e.g. the ReadFileActivity's lifecycle block), which lacks the ability to model the interleaved control flows between component callbacks. For example, consider the execution sequence shown earlier in Figure 3. There is no possible path in Figure 5 to model such a sequence, while our AICCFG can easily capture

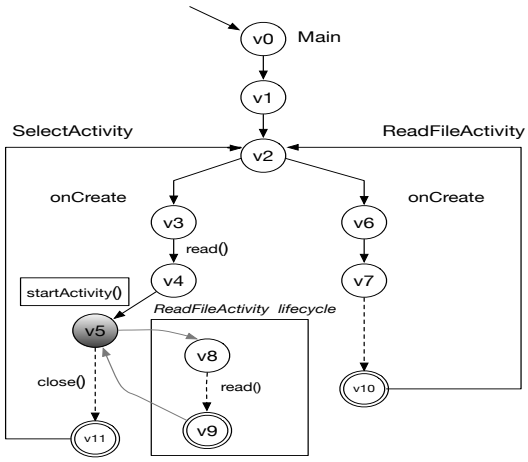

Fig. 5. Partial Android environment model generated by IccTA for the FileReader application

such a path (4-5-6-7-8) in Figure 4.

IV. Typestate Analysis

Typestate [13] is a refinement over the *type*. Whereas, the type of a data object defines the set of operations that are ever permitted on the object, *typestate* defines the subset of these operations that are valid in a given context. For example, Java Collections class allows to get the next element from the collection (call to Collections.next()) only if the collection has another element in the collection, else throws an *IllegalStateException*. Static typestate analysis could be highly useful in catching programs which might be syntactically legal but meaningless or semantically invalid [6]. In the absence of typestate analysis, the programmer needs to perform runtime checks and adhere to the the API usage rules which is not desirable and error prone.

Android framework provides a large set of resources like Camera, MediaPlayer, Databases, etc. to be used by the applications through APIs. Some of these resource APIs, e.g., Android MediaPlayer has a fairly complex protocol [1], making it highly burdensome and error prone to be enforced by the programmer. The violations of these protocols could have effects ranging from benign application crash to providing attack surfaces to attackers breaching application and user security.

Apart from the resource APIs, many other important safety properties in Android applications (like, granting and revoking of UriPermissions) could be modeled and verified using a typestate analysis. The control flow soundness and precision requirements of typestate analysis make it challenging for Android applications.

A. Android Typestate Analysis

This section defines a typestate analysis for Android over the control flow semantics and the AICCFG defined earlier. The analysis is the first typestate analysis for Android applications. We model our typestate problem as an asynchronous interprocedural finite distributive subset (AIFDS) problem [8].

Although our asynchronous analysis derives from that work in theory, there are the following challenges which are specific to asynchronous inter component analysis on Android applications-

- Android ICC calls have much complex runtime semantics and lack explicit asynchronous calls and returns. The asynchronous calls are either due to ICC or callbacks from the framework to the application. Moreover, these are asynchronous calls to a collection of callback methods rather than a single target method, which needs to be resolved either explicitly or using manifest. Once the target is resolved, all valid paths should be invoked based on the called component type and its lifecycle.
- Contrary to the definition of valid paths in [8], *valid* paths for the AICCFG discussed in section III, are a function of both pending asynchronous calls and application's component lifecycle rules. Thus we need a modified version of the original asynchronous data flow analysis (ADFA) algorithm to soundly calculate the meet-over-valid-path (MVP) solution over these valid paths of the application.

Formally an AIFDS formulation of the Android typestate analysis problem is defined as follows-

Definition IV.1 (**Typestate Property, FA**)**.** A typestate property to be verified, is represented as a finite automaton FA = $\langle \Sigma, Q, \delta, S, Q \backslash \{err\} \rangle$ where Σ is the set of operations possible on objects, Q is the set of typestates a resource object might exist in, δ is the *transition function* mapping a typestate and a $\sigma \in \Sigma$ to a target typestate, *err* is a single error state, *S* is a unique start state and finally all the states other than the error state are final states.

Definition IV.2 (**Typestate Mapping Function,** α)**.** We define $\alpha : Ref \mapsto 2^Q$, a typestate mapping function from the object reference set Ref, to the powerset of Q. $\alpha(r_i)$ represents the possible typestates a given reference $r_i \in Ref$ could have at a given program point.

B. Typestate as an AIFDS Problem

We describe the Android typestate verification problem as an AIFDS [8] instance. An AIFDS problem is an asynchronous version of an interprocedural finite distributive subset (IFDS) problem [12] for calculating MVP solutions over asynchronous programs. An AIFDS instance is a six tuple, A = $(G^*, D_g, D_l, CMap, F, \sqcup)$, which we solve using a modified ADFA algorithm to track typestate violations for a given application and a typestate property. We define each of these tuple of A now in detail.

1) ***The Program Representation,*** G^**:* The AICCFG G_*=(V_*, E_*), defined in section III, acts as the input program for our AIFDS analysis instance.

The typestate analysis also requires a typestate property automaton *FA* for the resource object as an input. In our current prototype implementation the user needs to provide this property automaton for the resource following the resource APIs documentation. We have provided such automata for important resource types for Android like Camera, MediaPlayer, File, SqliteDatabase, etc. in our implementation.

2) ***Data Flow Facts****:* Since the AICCFG is an asynchronous program representation, we need to split each typestate data flow fact into global and local component rather than having a single global data flow fact. Such a division is necessary because at the point of the asynchronous call, we need to capture incoming data flow facts, passed to the called procedure. We then store this asynchronous call as a pending call with these facts to be dispatched later. We cannot use a single global set of facts to represent the input fact for the pending call because operations that get executed between the asynchronous call and actual dispatch may change the global fact, leaving the local fact unchanged at the callsite.

The data flow facts D = ($D_l \times D_g \times$ *CMap*), where:

- D_g and D_l are global and local typestate data flow facts. Each data flow fact $d_g \in D_g$ or $d_l \in D_l$ is a pair of the form $(so_i, \{s_i, s_j ..., s_m\})$ where $so_i \in SO$. The $SO \subset Ref$ and is the set of symbolic object references for the resource objects, while Ref is the set of object references in the application. Each $s_i \in Q$, represents the possible set of typestates a given resource object so_i could be in, at a given program point.
- *CMap* : $(Ref \times Methods \times D_l) \mapsto N$, where Ref is the set of object references as described earlier, $Methods$ is the set of methods (callbacks, event handlers and normal methods), and N is the set of natural numbers. Intuitively the map $CMap$, captures the number of pending asynchronous calls not yet dispatched for a given method m $\in Methods$, of an object reference $r_i \in Ref$ with a given local data flow fact d_l. If there are no pending calls for the triple consisting of a given reference, method and local data flow facts, CMap maps the triple to 0.

3) ***Transfer Functions F for Typestate Analysis****:* Table I defines the set of transfer functions F for the problem. The columns give the rule number, the statement type, the transfer function and the side condition in which this function is applied. Each row for a given statement type is subdivided into subrows, defining the functions for a given side condition. Rule 1 applies to an object instantiation statement. The rule checks if the class being instantiated is a resource class (e.g. Android's Camera class), then it creates a new symbolic object so_i for the resource, initializes its typestate to the FA's start state *S* and updates the local or the global data flow facts depending upon the fact whether the statement defines a method local reference or an application level global reference. The call to Android resource initialization APIs are handled in a similar way. Rule 4 defines the transfer functions which applies the FA's typestate transition δ based on the incoming typestates and the operation performed on the symbolic object. The auxiliary function transferTS(), takes a statement (an operation), a data flow fact d_g/d_l (containing the current state) and the typestate property automata FA and returns an output data flow fact d'_g/d'_l (containing the target state). These operations do not alter the pending calls and hence the output CMap is same as the incoming CMap.

TABLE I
TYPESTATE TRANSFER FUNCTIONS

Transfer Functions F			
S.No.	Statement, st	$(d_g, d_l$, CMap) $\rightarrow$ Out	Side Conditions
1	new C()	$(d_g, d_l$, CMap) $\rightarrow$ $(d_g, d_l \cup \{(so_i, \{FA.start\})\}$, CMap)	(C $\in$ resourceclasses $\wedge$ Statement st defines a method local reference. $\wedge$ $so_i \notin SO$)
		$(d_g, d_l$, CMap) $\rightarrow$ $(d_g \cup \{(so_i, \{FA.start\})\}, d_l$, CMap)	(C $\in$ resourceclasses $\wedge$ Statement st defines an application level referenc. $\wedge$ $so_i \notin SO$)
		$(d_g, d_l$, CMap) $\rightarrow$ $(d_g, d_l$, CMap)	otherwise
2	startActivity(target) | startXYZ(target)	$(d_g, d_l$, CMap) $\rightarrow$ $(d_g, d_l$, CMap[$(r_t$, init(), $d_l) \leftarrow$ (CMap(r_t, init(), d_l)++), (r_t, onX(), $d_l) \leftarrow$ (CMap(r_t, onX(), d_l)++)]	$r_t \leftarrow$ getTartget(st)
3	dispatch(m(), d_l)	$(d_g, d_l$, CMap) $\rightarrow$ $(d_g, d_l$, CMap[$(r_t$, init(), $d_l) \leftarrow$ (CMap(r_t, init(), d_l)$--$), (r_t, onCreate(), $d_l) \leftarrow$ (CMap(r_t, onCreate(), d_l)$--$)]	(m = r_t.init() $\wedge$ CMap(r_t, init(), d_l) > 0)
		$(d_g, d_l$, CMap) $\rightarrow$ $(d_g, d_l$, CMap)	(m = r_t.onX() $\wedge$ CMap(r_t, onX(), d_l) > 0 $\wedge$ onX() $\neq$ onDestroy())
		$(d_g, d_l$, CMap) $\rightarrow$ $(d_g, d_l$, CMap[$(r_t$, onX(), $d_l) \leftarrow$ (CMap(r_t, onX(), d_l)$--$)]	(m = r_t.onDestroy() $\wedge$ CMap(r_t, onDestroy(), d_l) > 0)
4	Normal statement	$(d_g, d_l$, CMap)) $\rightarrow$ d'_g, d'_l, CMap)	$d'_g \leftarrow$ transferTS(st, d_g, FA) $d'_l \leftarrow$ transferTS(st, d_l, FA)

Rule 2 defines the transfer functions for asynchronous calls like *startActivity, startService* or *startXYZ* in general representing an ICC call. Android framework invokes a sequence of calls (modeled by init() in our AICCFG) on component creation, to model this semantics, this rule increments the pending calls for (r_t, init(), d_l). Once a component is created, the framework may invoke any lifecycle callback of the component, hence we also increment the pending calls for other callback methods (r_t, onX(), d_l). We have used 'onX()' for a generic callback method name (e.g. onStart(), onResume(), etc.). Rule 3 defines the transfer functions for the asynchronous dispatch of method m with data flow fact d_l from the special dispatch node of the AICCFG. It checks if the dispatched method m is an initialization method init() and has a pending call with the given d_l, then it dispatches the call and decrements the corresponding CMap cells for (r_t, init(), d_l) and (r_t, onCreate(), d_l). The onCreate() is called only once during the lifetime of a component instance, hence we decrement its counter here at the initialization. If the call is neither to init() nor to a component destruction method like onDestroy(), the function just dispatches the call without modifying the CMap. Android Activity component callback methods other than onCreate() and onDestroy() (similarly, for creation and destruction callback methods of other component types) can be invoked any number of times during the life time of the component due to some user or system event. Hence we do not decrement the pending calls to these methods here. Finally, the call to component destruction method like onDestroy(), decrements the pending calls for all the asynchronously called callbacks and the init() method for the given object in the CMap as the object is now dead and none of its callbacks could be invoked by the framework.

The transfer functions for asynchronous calls and dispatches for other component types are similarly defined and based on similar logic and are not presented here in view of limited space.

*4) **The meet operation** $\sqcup$:* The meet operation defines the meet of data flow facts along two or more different paths in the AICCFG. Our typestate analysis is conservative in nature and thus performs a weak update to typestate associated with any abstract object so_i. At junction nodes with two or more different incoming paths we perform a set union operation over the $\alpha(so_i)$ for each $so_i \in SO$. Formally, for any merge node p in the AICCFG of the application and for each abstract object so_i, the meet $\cup$ is defined as a union over the $\alpha(so_i)$ over all the valid interprocedural paths p_i starting from the start of the ambiance and merging at p.

For our analysis, we modified the ADFA algorithm [8], to handle asynchronous calls and dispatch semantics of Android applications. We then ran the modified ADFA to calculate *MVP(I_1)* and *MVP($I_1{}^\infty$)*, refer [8]. We did not need to run for higher values of *k=2,3...* as the *MVP(I_1)* and *MVP($I_1{}^\infty$)* converge for our typestate analysis problem over the benchmark applications we ran on.

V. IMPLEMENTATION

The overall implementation of our approach has two major components- (1) AICCFG generator and (2) A typestate analysis for Android applications.

A. AICCFG Generator

The AICCFG generator generates a sound AICCFG for the application using the AICCFG algorithm 1. The algorithm requires certain input information which is collected via a preprocessing phase performing a string analysis, manifest mining and preliminary context and flow insensitive pointer analysis. We have implemented the AICCFG generation algorithm using *Soot* static program analysis framework for Java [14].

B. Typestate Analysis

The typestate analysis is modeled as an instance of AIFDS problem [8] and is solved using a modified ADFA [8] built using Soot's *heros* IFDS implementation [14]. The typestate

TABLE II
BENCHMARK APPLICATIONS

App-type	Salient Features
Type0	No typestate violation.
Type1	Single typestate violation, **requires** sound asynchronous semantics modeling.
Type2 and Type3	Single typestate violation, **requires** sound asynchronous semantics modeling and sound lifecycle interleaving across components. These two types model two different interleavings between callbacks.

analysis, uses the AICCFG generated by the AICCFG generation algorithm as the program input and implements the transfer functions presented in table I.

VI. EVALUATION

A. AsyncBench Benchmarks

We present a set of synthetic benchmark applications, AsyncBench [3], containing tests for typestate violations whose verification requires a sound modeling and tracking of control and data dependencies in Android applications including the asynchronous semantics and sound lifecycle modeling. Although our benchmarks are small applications, the sound asynchronous control flow modeling presented by us is generic and the AICCFG construction algorithm is scalable. We leave the scalability studies and evaluations of our client typestate analysis as future work. Table II concisely presents the salient features of each type of application for each resource category. The four types of applications in each resource category test the correct modeling of asynchronous calls and lifecycle interleavings as discussed earlier in Section III, and the precision of these models (via benign applications with no typestate violations).

The aim of these benchmarks is two-fold. First, they check the coverage and soundness of the analysis for asynchronous calls and lifecycle properties of applications. Second, we add applications in various categories using different Android resources, namely *Camera, SQLiteDatabase, MediaPlayer, Files, etc*. This checks the usability angle of our analysis and shows that the analysis is generic enough to capture typestate violations against a rich set of Android resource usage protocols. Towards the similar goal, we also have applications, modeling the safe granting and revocation of Android UriPermissions. Android framework provides UriPermissions, which allows an application to grant temporary read/write access permissions (for its resource) to other applications. These UriPermissions are useful for ContentProviders to grant permissions to other applications to temporarily access some or whole of their data. This is done either by setting an Intent flag like, Intent.FLAG_GRANT_READ_URI_PERMISSION or by invoking the grantUriPermission() method of the Android ContextWrapper class. One possible bug in usage of these temporary UriPermissions is the leak of these permissions when the granter forgets to revoke the permissions through corresponding revokeUriPermission() call. Typestate can easily model and check such permission leaks by checking any UriPermission granted temporarily is always revoked before the granting component is terminated and a URIPermission is not revoked without being granted before.

TABLE III
TYPESTATE ANALYSIS ON ASYNCBENCH APPLICATIONS.

⊛ = correct warning, ⊖ = missed violation, ⋆ = false warning		
Application Name	Async-Aware	Sync-only (IccTA)
Camera Applications		
Cameraapp_0	-	⋆
Cameraapp_1	⊛	⊖
Cameraapp_2	⊛	⊖
Cameraapp_3	⊛, ⋆	⊖,⋆
MediaPlayer Applications		
MediaPlayer_0	-	⋆
MediaPlayer_1	⊛	⊖
MediaPlayer_2	⊛	⊖
MediaPlayer_3	⊛, ⋆	⊖,⋆
SQLiteDatabase Applications		
SQLiteDatabaseapp_0	-	⋆
SQLiteDatabaseapp_1	⊛	⊖
SQLiteDatabaseapp_2	⊛	⊖
SQLiteDatabaseapp_3	⊛, ⋆	⊖,⋆
File Applications		
Fileapp_0	-	⋆
Fileapp_1	⊛	⊖
Fileapp_2	⊛	⊖
Fileapp_3	⊛, ⋆	⊖,⋆
UriPermission Applications		
Permissionapp_0	-	⋆
Permissionapp_1	⊛	⊖
Permissionapp_2	⊛	⊖
Total, Precision and Recall		
⊛, higher is better	14	0
⋆, lower is better	4	8
⊖, lower is better	0	14
Precision = ⊛ / (⊛ + ⋆)	78 %	0 %
Recall = ⊛ / (⊛ + ⊖)	100 %	0 %

B. Results

Table III presents the static typestate analysis results for our asynchrony-aware approach on AICCFG against a synchronous-only typestate analysis built as an IFDS over the program representation used by other static analysis for Android (IccTA) applied to AsyncBench test applications. On these applications our asynchrony-aware typestate analysis captures all typestate violations and raises false warning in only one case in each resource category. Thus we achieve a precision rate of 78% and a recall rate of 100% over the AsyncBench benchmarks.

Compared to this the synchronous-only approach misses all the typestate violations in different categories due to the inherent unsoundness of its underlying program representation and its synchronous analysis. The high false negatives of synchronous only analysis shows the unsoundness in the state-of-the-art modeling of the Android environment which makes them miss many typestate violations. Moreover, a sound asynchronous and lifecycle modeling and an asynchrony-aware

analysis also increases the precision of our analysis, allowing us to give a 50% lower false warnings and higher precision compared to the synchronous-only analysis. Our asynchrony-aware typestate analysis runs smoothly on a normal desktop machine with a dual core Intel processor and a moderate memory size of 16 GB. The average time for analyzing an application in AsyncBench came out to be approximately 2-3 minutes. This shows the practical feasibility of our analysis on these applications.

VII. Related Work

A. Modeling and Static Analysis of Android Applications

Ours is the first work modeling the asynchronous semantics of Android application's control flow. Other works modeling the Android application environment and performing static analyses over applications are FlowDroid [4], IccTA [9] and AmanDroid [15]. FlowDroid and IccTA perform an information flow analysis over applications. Amandroid calculates points-to information for all objects in an Android application in a flow and context sensitive way across Android components. While IccTA and AmanDroid are capable of handling inter-component communications in applications, FlowFroid over-approximates these communications by tainting all flows across components. These works treat Android applications as synchronous programs, our work differs from them significantly as discussed throughout the paper. Epicc [11] computes Android ICC call targets using an IDE framework, their work is orthogonal to ours and our AICCFG construction could use the ICC targets resolved by their approach.

B. Analysis of Asynchronous Programs

Our analysis over Android applications leverages various concepts from the Jhala et. al. work [8]. Their work formalizes the problem for interprocedural data flow analysis for asynchronous programs as an AIFDS problem. We extend their work to formalize and model Android applications as asynchronous programs and then build a typestate analysis as an instance of the AIFDS problem discusses by them. Our work differs from theirs on several counts as discussed in section IV-A.

C. Typestate Analysis

Typestate analysis work from Fink et. al. [7] presents a sound typestate verification for real world Java programs. Since Android applications are inherently Java programs this work relates to our client typestate analysis. Despite this overlap, we differ greatly from their work which is at its core an IFDS instance of the typestate problem. They do not target asynchronous programs like us, nor do they handle Android applications.

VIII. Conclusions and Future Work

We presented a model of asynchronous control flow semantics and a sound and precise model of component lifecycle for Android applications. We presented an intermediate Android inter component control flow graph (AICCFG) for Android applications which soundly models these semantic features. We built a client typestate analysis for Android resource APIs protocol verification on our AICCFG. We presented a set of benchmark applications called AsyncBench and evaluated our work on these applications against the state-of-the-art static analysis for Android and showed empirically that other works are unsound, missing many important typestate violations and also lack precision as compared to our analysis.

We leave the formal modeling of Android control flow and the AICCFG and its soundness proof as future work. We also leave the scalability study of our client typestate analysis as a future direction of our work. Another interesting future direction could be the application of our modeling and analysis to other important static analyses (e.g. information flow analysis) over Android applications.

References

[1] Android media player. https://developer.android.com/reference/android/media/MediaPlayer.html.

[2] Appbrain. http://www.appbrain.com/stats/number-of-android-apps.

[3] Asyncbench. https://github.com/ashishtheaegis/AsyncBench.

[4] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[5] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[6] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, May 2008.

[7] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, May 2008.

[8] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. POPL '07, pages 339–350. ACM, 2007.

[9] L. Li, A. Bartel, T. F. Bissyand, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *ICSE*, volume 1, pages 280–291, 2015.

[10] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[11] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95, pages 49–61. ACM, 1995.

[13] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[14] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. CASCON '99, pages 13–. IBM Press, 1999.

[15] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. CCS '14, pages 1329–1341. ACM, 2014.