Launch-Mode-Aware Context-Sensitive Activity Transition Analysis*

Yifei Zhang
UNSW Sydney, Australia

Yulei Sui
University of Technology Sydney, Australia

Jingling Xue
UNSW Sydney, Australia

ABSTRACT

Existing static analyses model activity transitions in Android apps context-insensitively, making it impossible to distinguish different activity launch modes, reducing the pointer analysis precision for an activity's callbacks, and potentially resulting in infeasible activity transition paths. In this paper, we introduce Chime, a launch-mode-aware context-sensitive activity transition analysis that models different instances of an activity class according to its launch mode and the transitions between activities context-sensitively, by working together with an object-sensitive pointer analysis.

Our evaluation shows that our context-sensitive activity transition analysis is more precise than its context-insensitive counterpart in capturing activity transitions, facilitating GUI testing, and improving the pointer analysis precision.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Object oriented languages**;

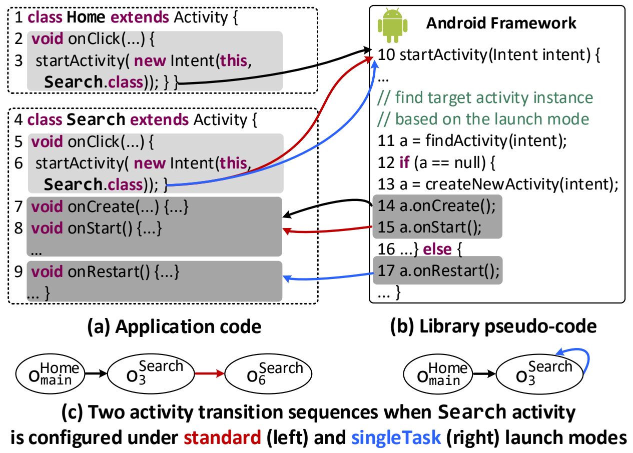
KEYWORDS

Android, Pointer Analysis, Activity Transition Analysis

ACM Reference Format:
Yifei Zhang, Yulei Sui, and Jingling Xue. 2018. Launch-Mode-Aware Context-Sensitive Activity Transition Analysis. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180188

1 INTRODUCTION

Activity, as a major type of Android components, lies at the heart of the Android programming framework due to its event-driven nature. An activity acts as a container consisting of various GUI elements (e.g., views and text boxes), through which, users interact with an app for activity navigations, i.e., transitions between different activities. Conceptually, an app executes along the activity transition paths and other callbacks are sprawled out of them.



```
class Home extends Activity {
  void onClick(...) {
    startActivity( new Intent(this,
      Search.class)); } }
```

```
class Search extends Activity {
  void onClick(...) {
    startActivity( new Intent(this,
      Search.class)); }
  void onCreate(...) {...}
  void onStart() {...}
  ...
  void onRestart() {...}
... }
```

```
startActivity(Intent intent) {
...
// find target activity instance
// based on the launch mode
a = findActivity(intent);
if (a == null) {
a = createNewActivity(intent);
a.onCreate();
a.onStart();
...} else {
a.onRestart();
... }
```

Figure 1: Three activity transitions, represented by ⟶, ⟶ and ⟶, via callbacks: `Home` **launches** `Search` **and** `Search` **launches itself with** `standard` **and** `singleTask` **modes.**

A fundamental static analysis for Android apps is to model activity transitions for event-driven callbacks. This serves as a cornerstone for many clients, such as vulnerability detection [5, 6, 9, 10, 14, 19, 27, 31], malware detection and mitigation [7, 8, 27], GUI model generation [32, 33], and GUI testing [2–4, 20, 23].

The core data structure used for an app is an *activity transition graph* (ATG) [2, 4, 23], which represents the activity transitions in the app. In an ATG, a node represents an activity instance and an edge between two nodes denotes an activity transition.

In Android, a parent activity can start a child activity by invoking, e.g., `startActivity()` as a form of an inter-component communication (ICC) call, passing it an intent that describes the child activity to be launched. In addition, an activity instance of a class, say, T can be launched in one of the four launch modes, `standard`, `singleTask`, `singleTop` and `singleInstance`, either configured in `AndroidManifest.xml` or specified in the intent passed to `startActivity()`. The first one is the default while the other three are known as special launch modes. These launch modes affect which activity instances are launched and their transitions. For example, `standard` always requests a new activity instance of T to be launched while `singleTask` requires an existing activity instance of T to be reused if it exists (by thus limiting only one instance of T in each activity transition path).

Figure 1 gives an example to demonstrate how we intend to use an ATG to keep track of activity transitions (in the presence of ICC calls) under different launch modes, with `Search` being configured once with `standard` and once with `singleTask`. We assume a navigation scenario where the user clicks a button on the

*Thanks to all the reviewers for their valuable comments. This research is supported by ARC grants, DP170103956 and DE170101081. Yifei Zhang is supported by an Australian Government Research Training Program Scholarship.

current Home activity to launch a Search activity via the ICC call `startActivity()` at line 3. Next, the user clicks another button on the launched Search activity, causing either a new Search activity (with `standard`) or the existing one (with `singleTask`) to be launched via the ICC call `startActivity()` at line 6.

Let us understand the internal working of ICC callbacks in the above navigation scenario for the three transitions as highlighted in black, red and blue arrows. There are three steps for each transition via the Android callback mechanism when an app (Figure 1(a)) interacts with the Android framework (Figure 1(b)). First, the app code passes a new intent object (line 3) to the framework (line 10). Second, the framework finds the corresponding activity instance *a* (line 11) based on the launch mode specified. As shown in Figure 1(b), a new instance is created (line 13) if the activity is launched for the first time, i.e., $o^{Home}_{main} \rightarrow o^{Search}_{3}$, or in `standard` mode with the transition $o^{Search}_{3} \rightarrow o^{Search}_{6}$, where o^{Search}_{3} and o^{Search}_{6} are two different instances of Search. Finally, the framework launches Search via the callbacks (lines 14-15) to their corresponding lifecycle methods (lines 7-8). If a special mode, `singleTask`, is used, the Android framework will retrieve the existing Search instance to restart the activity (line 17), $o^{Search}_{3} \circlearrowleft$.

Building ATGs statically to reason about activity transitions for Android apps is challenging. Unlike a Java program with a dedicated main() method, an Android app can have multiple entry points, with some activities implicitly launched by the Android framework through nondeterministic user and system events. This can significantly complicate static activity transition analysis in the presence of a large number of callbacks. In addition, activity transitions through ICC make use of intent objects to specify target activities launched, requiring a precise pointer analysis to discover the contents in such intents. Finally, the heap allocation sites for activity instances are invisible in the app code, as they are distributed via deep call chains in the Android framework. Thus, performing a precise pointer analysis for an app over the entire Android framework (with millions of lines of code) is unrealistic [9].

Existing GUI testing techniques [2, 4, 23] use context-insensitive ATGs to model activity transitions in Android apps, implying that all instances of an activity class are abstracted with one single object. Thus, context-insensitivity makes it inherently impossible to distinguish different activity launch modes, reduces the pointer analysis precision for an activity's callbacks, and potentially introduces infeasible activity transition paths. To analyze the callbacks of an activity, existing static analyses [1, 9, 14, 24–26, 31] construct a "fat" harness `main()`, consisting of one allocation site per activity class (for allocating one single abstract object for the class) and the calls to all its callbacks (e.g., `onCreate()` and `onClick()`) on the abstract object. In Figure 1, $o^{Search} \circlearrowleft$ will be created regardless of the launch mode specified for Search. The transitions between activities are modeled context-insensitively, as discussed Section 2.

To address the above-mentioned challenges in modeling activity transitions and the limitations of the prior work, we present Chime, a launch-mode-aware context-sensitive activity transition analysis for Android apps that builds context-sensitive ATGs, together with an object-sensitive pointer analysis [12, 13, 15–17, 21, 22, 28–30].

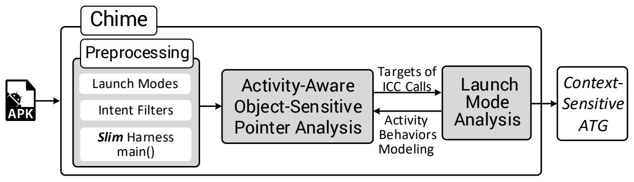


Figure 2: An overview of the Chime framework.

By maintaining context-sensitivity in tracking activity transitions, Chime can model launched activities based on their launch modes and reduce the number of infeasible transition paths (that would otherwise be introduced context-insensitively). By distinguishing different instances of an activity class based on their activity transition sequences as contexts, Chime avoids having to analyze the Android framework (e.g., to look for the heap allocation sites for activities) and improves the pointer analysis precision for the callbacks of an activity.

Figure 2 depicts our framework. The activity-aware object-sensitive pointer analysis and launch mode analysis are mutually dependent. Its preprocessing phase extracts metadata, including activity launch modes and intent filters for an Android app. Unlike existing static analyses [1, 9, 14, 24–26, 31], which use a fat harness `main()` for allocating all the activities and calling its callbacks (as mentioned earlier), Chime builds an ATG for an app by using a slim harness `main()`, consisting of initially the same information but restricted to the *entry activities*, i.e., the ones registered via intent filters or exported to the system. The other activities will be gradually introduced into the ATG during the analysis.

This paper makes the following contributions:

- We introduce a launch mode analysis for distinguishing launched instances of an activity class based on the launch mode specified.
- We introduce Chime, a launch-mode-aware context-sensitive activity transition analysis that tracks activity transitions context-sensitively, together with an object-sensitive pointer analysis.
- We have implemented Chime in the DroidSafe framework [9]. Evaluated with a set of 42 large real-world Android apps from Google Play, Chime is shown to be more precise than its context-insensitive counterpart in modeling activity transitions, guiding GUI testing, and improving the pointer analysis precision.

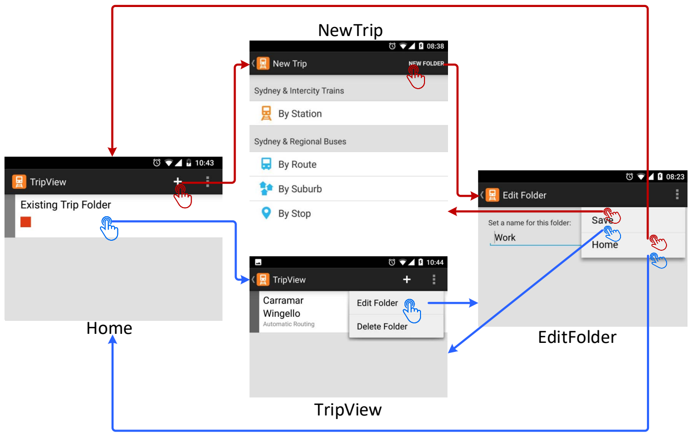
2 MOTIVATION

In Section 1, we have used an example in Figure 1 to explain why modeling activity transitions context-sensitively is essential in accounting for the effects of launch modes on launched activities. In this section, we use another example to demonstrate why such context-sensitive modeling is also essential in modeling accurately activity transitions, improving the pointer analysis precision, and potentially reducing the number of infeasible activity transitions.

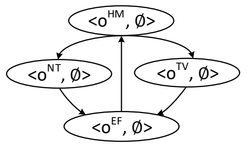
With this second example in Figure 3, we show how Chime models context-sensitively the activity instances launched for Home, NewTrip, TripView, and EditFolder, and their transitions, by considering their launch modes. In Figure 3(a), we see a code snippet from *TripView*, a real public transportation trip plan tool. Home is the main activity. Two launch modes, `singleTask` and `standard`, will be considered. The other three activities, configured under `standard` launch mode, are not exported (to the system).

```
1  class Home extends Activity {
2 // The “+” button registered with the following callback
3   void onClick(View view) {
4    Intent HMtoNT = new Intent(this, NewTrip.class);
5    startActivity(HMtoNT); }
6 // A list registered with the following callback
7   void onItemSelected(..., int pos, long row) {
8    if(getItem(pos) instanceof TripFolder) {
9     Intent HMtoTV = new Intent(this, TripView.class);
10    startActivity(HMtoTV); } ... }
11  void onNewIntent(..., Intent EFtoHM) {...} }
12 class NewTrip extends Activity {
13 // The “New Folder” button registered with the following callback
14  void onClick(View view) {
15   Intent NTtoEF = new Intent(this, EditFolder.class);
16   startActivityForResult(NTtoEF, ...); }
17  void onActivityResult(..., Intent EFtoNT) {...} }
18 class TripView extends Activity {
19 // A menu registered with the following callback
20  void onContextItemSelected(MenuItem menuItem) {
21   if(menuItem.getItemId() == EDITFOLDER) {
22    Intent TVtoEF = new Intent(this, EditFolder.class);
23    startActivityForResult(TVtoEF, ...); } ... }
24  void onActivityResult(..., Intent EFtoTV) {...} }
25 class EditFolder extends Activity {
26 // A menu registered with the following callback
27  void onContextItemSelected(MenuItem menuItem) {
28   if(menuItem.getItemId() == SAVE) {
29    Intent ret = new Intent();
30    ret.putExtra("folder_name", name);
31    setResult(..., ret);
32   } else if(menuItem.getItemId() == HOME) {
33    Intent EFtoHM = new Intent(this, Home.class);
34    startActivity(EFtoHM); } ... } }
```

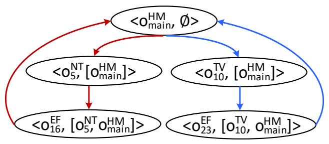
(a) App code abstracted from `TripView`



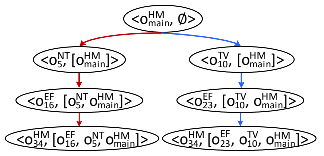
(b) Activity transitions (including GUI events)



(c) Context-insensitive ATG



(d) context-sensitive ATG (`singleTask` for `Home`)



(e) Context-sensitive ATG (`standard` for `Home`)

Figure 3: A motivating example. o^t_- is an instance of t, where $t \in$ {`Home (HM)`, `NewTrip (NT)`, `TripView (TV)`, `EditFolder (EF)`}. In (d) and (e), $\longrightarrow$ and $\longrightarrow$ denote two context-sensitive activity transition paths identified by CHIME.

2.1 Two Navigations Scenarios

Figure 3(b) shows two activity navigations highlighted in red and blue arrows, respectively. For the red navigation, Home launches `NewTrip` via the ICC call at line 5 once the "+" button for `NewTrip` is clicked. Then `NewTrip` launches `EditFolder`, once the "New Folder" button is clicked, to create a new trip folder via the ICC call `startActivityForResult()` at line 16, and subsequently, receives an intent object from the target activity as its result. If "Save" on the menu is clicked, `EditFolder` transits back to `NewTrip` so that `setResult()`, which is invoked at line 31, returns an intent object as the result to the previous `NewTrip` instance that has launched `EditFolder` through the Android callback `onActivityResult()` at line 17. If "Home" on the menu is clicked instead, `EditFolder` transits back to `Home` via the ICC call at line 34.

For the blue navigation, `Home` launches `TripView` to look up an existing trip at line 10 once the list item "`Existing Trip Folder`" is selected. `TripView` launches `EditFolder` to edit the name of an existing folder via the ICC call at line 23 once "Edit Folder" on the menu is clicked. Then `EditFolder` transits back to `TripView` if "Save" is clicked or returns to `Home` if "Home" is clicked.

2.2 Context-Insensitive Transition Analysis

Existing static analyses [1, 2, 4, 9, 14, 23–26, 31] model activity transitions context-insensitively. For our example, the context-insensitive ATG obtained is shown in Figure 3(c). For each app, a "fat" harness `main()` is created, responsible for allocating one abstract object for each activity class (to represent all its instances) and calling all its callbacks (e.g., `onCreate()`) on the abstract object. Thus, each abstract activity is parameterized with an empty context $\emptyset$. As there are only four context-insensitive activity objects, $\langle o^{HM}, \emptyset\rangle$, $\langle o^{NT}, \emptyset\rangle$, $\langle o^{TV}, \emptyset\rangle$ and $\langle o^{EF}, \emptyset\rangle$, the two `EditFolder` instances that are actually created at run time along two different transition paths are indistinguishable. In addition, regardless of the launch mode specified for `Home`, the same ATG is always built.

Finally, ATGs are essentially call graphs but used for modeling activity transitions. When constructed context-insensitively, ATGs will potentially exhibit infeasible transition paths.

2.3 Context-Sensitive Transition Analysis

For our example, CHIME will eventually build the context-sensitive ATG in Figure 3(d) if `Home` uses `singleTask` as its launch mode and the one in Figure 3(e) if `Home` uses `standard` as its launch mode. Initially, its slim harness `main()` allocates one instance, $o^{HM}_{\texttt{main}}$, of the main activity `Home` (and also calls all the callbacks registered). As before, $o^{HM}_{\texttt{main}}$ is parameterized with $\emptyset$ in the ATG.

Our activity-aware pointer analysis is then performed on the harnessed app to resolve its activity-related ICC calls. During the analysis, the pointed-to objects of the intent variables at the two ICC calls at lines 5 and 10 are queried in order to find their target activities. Once their class types, `NewTrip` (line 4) and `TripView` (line 9), are found, CHIME creates two activity instances, $\langle o^{NT}_5, [o^{HM}_{\texttt{main}}]\rangle$ at line 5 and $\langle o^{TV}_{10}, [o^{HM}_{\texttt{main}}]\rangle$ at line 10, and then adds the transition edges from $o^{HM}_{\texttt{main}}$ to o^{NT}_5 and o^{TV}_{10} as shown. Note that o^{NT}_5 and o^{TV}_{10} are parameterized with $[o^{HM}_{\texttt{main}}]$, a context representing currently the only activity transition sequence reaching $o^{HM}_{\texttt{main}}$. For `EditFolder`, however, there are two reaching transition paths, resulting in two different instances, $\langle o^{EF}_{16}, [o^{NT}_5, o^{HM}_{\texttt{main}}]\rangle$ and $\langle o^{EF}_{23}, [o^{TV}_{10}, o^{HM}_{\texttt{main}}]\rangle$, parameterized by two different transition sequences.

Finally, $\langle o^{\text{EF}}_{16}, [o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$ at line 34 launches a `Home` activity. If `Home` is configured with `singleTask` (Figure 3(d)), Chime retrieves its existing instance $\langle o^{\text{HM}}_{\text{main}}, \emptyset\rangle$ along the transition path reaching $\langle o^{\text{EF}}_{16}, [o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$, and adds an edge from $\langle o^{\text{EF}}_{16}, [o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$ to $\langle o^{\text{HM}}_{\text{main}}, \emptyset\rangle$. Similarly, the context-sensitive transition from $\langle o^{\text{EF}}_{23}, [o^{\text{TV}}_{10}, o^{\text{HM}}_{\text{main}}]\rangle$ to $\langle o^{\text{HM}}_{\text{main}}, \emptyset\rangle$ is added. If `Home` is configured with `standard` instead (Figure 3(e)), two new instances of `Home` will be created, $\langle o^{\text{HM}}_{34}, [o^{\text{EF}}_{16}, o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$ and $\langle o^{\text{HM}}_{34}, [o^{\text{EF}}_{23}, o^{\text{TV}}_{10}, o^{\text{HM}}_{\text{main}}]\rangle$, one per each transition path, resulting in the following two new edges added: $\langle o^{\text{EF}}_{16}, [o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle \longrightarrow \langle o^{\text{HM}}_{34}, [o^{\text{EF}}_{16}, o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$ and $\langle o^{\text{EF}}_{23}, [o^{\text{TV}}_{10}, o^{\text{HM}}_{\text{main}}]\rangle \longrightarrow \langle o^{\text{HM}}_{34}, [o^{\text{EF}}_{23}, o^{\text{TV}}_{10}, o^{\text{HM}}_{\text{main}}]\rangle$.

2.4 Object-Sensitive Pointer Analysis

Consider the class `EditFolder` with `onCreate()` shown in Figure 4. At line 4, its `intentEF` is known to point to only one of the two intent objects passed from lines 15 and 22 in Figure 3(a), respectively.

```
1 class EditFolder extends Activity {
2  protected void onCreate(...) {
3   ...
4   Intent intentEF = getIntent();
5   ... } }
```

Figure 4: Context-sensitive pointer analysis of callbacks with context-insensitive and context-sensitive ATGs.

Based on the context-insensitive ATG in Figure 3(c), existing context-sensitive pointer analyses for Android apps [9, 31] will analyze all the callbacks of an activity context-insensitively. As there is only one instance of `EditFolder`, $\langle o^{\text{EF}}, \emptyset\rangle$, in Figure 3(c), `onCreate()` will be analyzed only once on the receiver object $\langle o^{\text{EF}}, \emptyset\rangle$. As a result, `intentEF` at line 4 in Figure 4 will point to the two intent objects passed from lines 15 and 22 in Figure 3(a).

Chime builds one of the two context-sensitive ATGs in Figures 3(d) and (e) depending on whether `Home` is configured with `singleTask` or `standard` launch mode. There are two instances of `EditFolder`, $\langle o^{\text{EF}}_{16}, [o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$ (representing the instance of `EditFolder`, started at line 16 in Figure 3(a)) and $\langle o^{\text{EF}}_{23}, [o^{\text{TV}}_{10}, o^{\text{HM}}_{\text{main}}]\rangle$ (representing the instance of `EditFolder`, started at line 23 in Figure 3(a)). Our activity-aware object-sensitive pointer analysis is able to analyze `onCreate()` separately. With $\langle o^{\text{EF}}_{16}, [o^{\text{NT}}_{5}, o^{\text{HM}}_{\text{main}}]\rangle$ as its receiver object, `intentEF` will point to the intent object pointed to by `NTtoEF` at line 15 in Figure 3(a). With $\langle o^{\text{EF}}_{23}, [o^{\text{TV}}_{10}, o^{\text{HM}}_{\text{main}}]\rangle$ as its receiver object, `intentEF` will point to the intent object pointed to by `TVtoEF` at line 22 in in Figure 3(a).

3 APPROACH

First, we review briefly Android's intents and its ICC mechanism (Section 3.1) and introduce our notations used (Section 3.2). Next, we discuss how Chime preprocesses an app to enable our subsequent analyses (Section 3.3). Then, we introduce our activity-aware object-sensitive pointer analysis (Section 3.4). Finally, we start with a context-insensitive approach (Section 3.5) and then move to a context-sensitive one (Section 3.6) for modeling activity transitions.

3.1 Android Intents and ICC Calls

The `intent` at an ICC call, e.g., `startActivity(intent)` determines the target activity to be opened, where `intent` is either *explicit* or *implicit*. With an explicit intent, the class name of the target activity is explicit (e.g., as in line 3 in Figure 1). With an implicit intent, an action, together with some data with which to perform the action, is provided. The Android framework will find a target class with a registered intent filter (in its app's `AndroidManifest.xml`) that has advertised its ability for performing the action.

The Android framework provides various APIs to store the information regarding a target activity in the fields of an intent object, such as `setClassName()` and `setAction()`, whose parameters are target class name and action name, respectively. For an explicit intent, the target class name is stored into the field `Intent.mComponent.mClass`, where the type of `Intent.mComponent` is `ComponentName` and the type of `ComponentName.mClass` is `String`. We will use the action field, `Intent.mAction`, to resolve an implicit intent. Therefore, the target activity of an ICC call can be determined by querying the pointed-to values in the fields of an intent object (Section 3.4).

3.2 Notations

Figure 5 gives our notations. The first part lists the domains used. Abstract objects are created at object allocation sites. To achieve object-sensitivity, a context is a sequence of abstract objects.

variable	$intent, cpnt, a \in \mathbb{V}$
class type	$\mathsf{t} \in \mathbb{T}$
abstract object	$o_i, o_t, o_i^t \in \mathbb{O}$
context	$c \in \mathbb{C} = \emptyset \cup \mathbb{O} \cup \mathbb{O}^2 \cup \ldots$
launch mode	$\{\mathsf{Std}, \mathsf{Top}, \mathsf{Task}, \mathsf{Inst}\} = \mathbb{LM}$
$pt:$	$\mathbb{V} \times \mathbb{C} \to \mathcal{P}(\mathbb{O} \times \mathbb{C})$
$fpt:$	$\mathbb{O} \times \mathbb{C} \times \mathbb{F} \to \mathcal{P}(\mathbb{O} \times \mathbb{C})$
$newActObj:$	$\mathbb{T} \to \mathbb{O}$
$heapCtxSelector:$	$\mathbb{O} \times \mathbb{C} \to \mathbb{C}$
$mtdCtxSelector:$	$\mathbb{O} \times \mathbb{C} \to \mathbb{C}$
ATG	$G = (V, E)$
ATG Node	$\langle o_i, c\rangle \in V \subseteq \mathbb{O} \times \mathbb{C}$
ATG Edge	$\langle o_i, c\rangle \hookrightarrow^{p} \langle o_j, c'\rangle \in E \subseteq V \times V \times \mathbb{N}$

Figure 5: Notations.

The middle part lists several functions used in our inference rules. $pt(p, c)$ gives the points-to set of a pointer p under context c. $fpt(o_i, c, \mathsf{f})$ gives the points-to set of the field f of an object o_i under context c. $newActObj(\mathsf{t})$ creates an instance of an activity class t in `main()` (while $new(\mathsf{t})$ applies to every other type t). Following [12, 21, 22, 29], we use *mtdCtxSelector* (*heapSelector*) to generate a method (heap) context required for analyzing (identifying) a virtual method call (a non-activity object) to enable k-object-sensitive pointer analysis. Note that *heapCtxSelector* and *mtdCtxSelector* are recursively defined. For *heapCtxSelector*, the object o_i at allocation site i is modeled context-sensitively by a heap context $[o_{i_1}, \ldots, o_{i_{k-1}}]$ (of length $k-1$), where i_j is the allocation site for the receiver object o_{i_j} of the method that contains i_{j-1} (with $i_0 = i$). For *mtdCtxSelector*, if x points to an object o_i modeled under a heap context $[o_{i_1}, \ldots, o_{i_{k-1}}]$, then the k-object-sensitive method context used for analyzing a virtual call $x.foo()$ is $[o_i, o_{i_1}, \ldots, o_{i_{k-1}}]$.

The bottom part defines an ATG as a directed multi-edge graph for an app. A node $\langle o_i, c\rangle$ represents a context-sensitive instance of an activity. An edge $\langle o_i, c\rangle \hookrightarrow^p \langle o_j, c'\rangle$ from a parent activity $\langle o_i, c\rangle$ to a child activity $\langle o_j, c'\rangle$ represents an activity transition, where $\langle o_i, c\rangle$ starts $\langle o_j, c'\rangle$ via an ICC call, say, `startActivity()` at line p. In Figures 1 and 3, the line numbers on their edges are elided.

We will present our formalism for a Java-like language consisting of five types of statements: object allocation (`x = new t()`), copy (`x = y`), load (`x = y.f`), store (`x.f = y`) and virtual call (`x = y.g(arg`$_1$`,...,arg`$_n$`)`). Static variables and static method calls are excluded but handled context-insensitively.

In addition, we handle four types of ICC-related statements specially. We resolve an explicit intent at `intent.mComponent = cpnt` and an implicit intent at `intent.mAction = action`. For activity-related ICC calls, we consider `startActivity(intent)` (for starting an activity) and `setResult(intent)` (for receiving a result from an activity). Note that `startActivity()` represents all the APIs that can start an activity, including, e.g., also `startActivityForResult()` and `startActivityIfNeeded()`. By handling `startActivity()`, `onNewIntent()` is also modeled. By handling `setResult()`, `onActivityResult()` is also modeled.

3.3 Preprocessing

During this preprocessing step, an app is decompiled to extract its code and metadata, such as the launch mode of and intent filters registered by each activity class in its `AndroidManifest.xml` file. To analyze an app, we must generate a slim harness `main()`.

For the user-defined Android components of class types, `Service`, `BroadcastReceiver` and `ContentProvider` in an app, we proceed as in [1, 9, 31], by including in the harness (1) one heap allocation site for each component class (for allocating one abstract object from this class) and (2) a call to each of its callbacks registered for the object. These components can start an activity but do not otherwise contribute to activity transitions.

For the user-defined Android components of class type, `Activity`, the harness for an app will include the same information pertaining only to its entry activities, i.e., the ones registered via intent filters or exported to the system. All the other activities that are started by these entry activities will be added later to the ATG of the app context-sensitively during the analysis.

3.4 Activity-Aware Object-Sensitive (Whole-Program) Pointer Analysis

Given an app, Chime builds a context-sensitive ATG $G = (V, E)$ while performing an object-sensitive pointer analysis to the app, starting from its harness `main()`. Figure 6 gives our rules. Let $contexts(m)$ be the set of all the contexts reaching method m from `main()`, where $contexts(\texttt{main()}) = \emptyset$. Initially, $G = (V, E)$ is empty.

Let us consider the first six rules. We handle an object allocation site depending on whether it is for an activity class in `main()` ([P-NewAct]) or otherwise ([P-NewOther]). In the former case, the activity instance created is added to the ATG. No edges are available at this stage yet. In each case, o_i^{-} uniquely identifies the abstract object created as an instance of `t` at allocation site i.

In [P-Copy], [P-Load], [P-Store] and [P-Call], we handle all the other non-ICC-related statements. In [P-Copy], the points-to facts flow from RHS to the LHS of a copy statement. In [P-Load] and [P-Store], the fields of an abstract object o_i are distinguished. In

m: the containing method for each statement being analyzed

$$\frac{\begin{array}{c} i : x = \texttt{new}\ t() \quad m = \texttt{main()} \\ t\ \textit{is a subtype of}\ \texttt{Activity} \end{array}}{\begin{array}{c} \langle o^t_{\texttt{main}}, \emptyset\rangle \in pt(x, \emptyset) \\ \langle o^t_{\texttt{main}}, \emptyset\rangle \in V \end{array}} \quad \text{[P-NewAct]}$$

$$\frac{\begin{array}{c} i : x = \texttt{new}\ t() \quad c \in contexts(m) \\ \neg(m = \texttt{main()}\ \wedge\ t\ \textit{is a subtype of}\ \texttt{Activity}) \end{array}}{\langle o_i, heapCtxSelector(o_i, c)\rangle \in pt(x, c)} \quad \text{[P-NewOther]}$$

$$\frac{x = y \quad c \in contexts(m)}{pt(y, c) \subseteq pt(x, c)} \quad \text{[P-Copy]}$$

$$\frac{x = y.\mathrm{f} \quad c \in contexts(m) \quad \langle o_i, c'\rangle \in pt(y, c)}{fpt(o_i, c', \mathrm{f}) \subseteq pt(x, c)} \quad \text{[P-Load]}$$

$$\frac{x.\mathrm{f} = y \quad c \in contexts(m) \quad \langle o_i, c'\rangle \in pt(x, c)}{pt(y, c) \subseteq fpt(o_i, c', \mathrm{f})} \quad \text{[P-Store]}$$

$$\frac{\begin{array}{c} x = y.g(arg_1, ..., arg_n) \\ c \in contexts(m) \quad \langle o_i, c'\rangle \in pt(y, c) \\ g' = dispatch(o_i, g) \quad c'' = mtdCtxSelector(o_i, c') \end{array}}{\begin{array}{c} c'' \in contexts(g') \quad \langle o_i, c'\rangle \in pt(g'_{this}, c'') \\ \forall\ 1 \leqslant k \leqslant n : pt(arg_k, c) \subseteq pt(g'_{pk}, c'') \\ pt(g'_{ret}, c'') \subseteq pt(x, c) \end{array}} \quad \text{[P-Call]}$$

$$\frac{\begin{array}{c} intent.\texttt{mComponent} = cpnt \quad c \in contexts(m) \\ \langle o_i, c'\rangle \in pt(intent, c) \quad \langle o_j, c''\rangle \in pt(cpnt, c) \end{array}}{fpt(o_j, c'', \texttt{mClass}) \subseteq fpt(o_i, c', \texttt{tgtClassName})} \quad \text{[I-Ex]}$$

$$\frac{\begin{array}{c} intent.\texttt{mAction} = action \quad c \in contexts(m) \\ \langle o_i, c'\rangle \in pt(intent, c) \quad \langle o_j, \_\rangle \in pt(action, c) \end{array}}{actionToCpnt(o_j) \subseteq fpt(o_i, c', \texttt{tgtClassName})} \quad \text{[I-Im]}$$

$$\frac{\begin{array}{c} p :\ this.startActivity(intent) \quad c \in contexts(m) \\ this\ \texttt{instanceof Activity} \quad \langle o_i, c'\rangle \in pt(this, c) \\ \langle o_j, c''\rangle \in pt(intent, c) \quad \langle o_k, \_\rangle \in fpt(o_j, c'', \texttt{tgtClassName}) \\ t = getType(o_k) \quad \texttt{lm} = getLM(t) \end{array}}{(\langle o_i, c'\rangle, \langle o_j, c''\rangle, \texttt{lm}, p) \rightsquigarrow t} \quad \text{[I-Act]}$$

$$\frac{\begin{array}{c} this.setResult(intent) \quad c \in contexts(m) \\ this\ \texttt{instanceof Activity} \quad \langle o_k, c'\rangle \in pt(this, c) \\ \langle o_j, c'''\rangle \in pt(intent, c) \quad \langle o_i, c''\rangle \hookrightarrow^{-} \langle o_k, c'\rangle \in E \end{array}}{\texttt{modelSetResult}(\langle o_i, c''\rangle, \langle o_j, c'''\rangle)} \quad \text{[I-Ret]}$$

Figure 6: Rules for activity-aware object-sensitive pointer analysis (`tgtClassName` is a pseudo field for an intent object).

[P-Call], the function $dispatch(o_i, g)$ is used to resolve the virtual dispatch of method g on the receiver object o_i to be g'. We assume that g' has a formal parameter g'_{this} for the receiver object and $g'_{p1}, ..., g'_{pk}$ for the remaining parameters, and a pseudo-variable g'_{ret} is used to hold the return value of g'.

Finally, let us consider the last four rules for handling ICC-related statements. [I-Ex] and [I-Im] resolve explicit and implicit intents, respectively (passed via ICC calls). To handle both uniformly, a pseudo field of type `String`, `tgtClassName`, for an intent is used.

In [I-Ex], we analyze `intent.mComponent = cpnt` by storing the name of each target activity class in `cpnt.mClass` directly into `intent.tgtClassName`. In [I-Im], we do likewise in analyzing `intent.mAction = action` except that $actionToCpnt(o_j)$ gives rise to the set of target classes (identified again by their names) that have registered an intent filter for performing action o_j.

[I-Act] and [I-Ret] handle the two ICC APIs, `a.startActivity()` and `a.setResult()`, respectively. For simplicity, we assume that `a = this`, where `this` is an instance of `Activity`. This captures the two mostly commonly used cases. The other cases can be handled similarly, involving a simple search for any parent activity that may possibly start a child activity and possibly receive a result from the child activity in the app.

In [I-Act], every call to `this.startActivity(intent)` at line p induces a fact $(\langle o_i, c'\rangle, \langle o_j, c''\rangle, \texttt{lm}, p) \rightsquigarrow t$, meaning that a parent activity $\langle o_i, c'\rangle$ starts an instance of activity class t with its launch mode lm at line p, with $\langle o_j, c''\rangle$ passed as the intent object. Here, $getType(o_k)$ maps the name of a class o_k to its corresponding class type and $getLM(t)$ obtains the launch mode of an activity class t.

In [I-Ret], we analyze `setResult(intent)`, where the intent is given by $\langle o_j, c'''\rangle$, which is invoked on a child activity $\langle o_k, c'\rangle$ in order to find its matching callback `onActivityResult(...,intentRES)` that is invoked on its parent activity $\langle o_i, c''\rangle$ (thanks to the ATG built on the fly). Now, $\texttt{modelSetResult}(\langle o_i, c''\rangle, \langle o_j, c'''\rangle)$ simply tells the pointer analysis to analyze `onActivityResult(...,intentRES)` invoked on $\langle o_i, c''\rangle$, where `intentRES` points to $\langle o_j, c'''\rangle$.

3.5 Launch-Mode-Unaware Context-Insensitive Activity Transition Analysis

Figure 7 gives the rule [Act-Ci] for building a context-insensitive ATG, $G = (V, E)$, and also performing the pointer analysis for the callbacks of every newly started activity at the same time.

$$\frac{(\langle o_i,\emptyset\rangle,\langle o_j,c\rangle,\_,p)\rightsquigarrow t}{\begin{array}{c} o_t = newActObj(t) \quad \langle o_j,c\rangle \in fpt(o_t,\emptyset,\texttt{mIntent}) \\ \langle o_t,\emptyset\rangle\in V \quad \langle o_i,\emptyset\rangle \hookrightarrow^p \langle o_t,\emptyset\rangle \in E \\ \texttt{modelCallBacks}(\langle o_t,\emptyset\rangle) \\ \texttt{modelNewIntent}(\langle o_t,\emptyset\rangle,\langle o_j,c\rangle)\end{array}}\;[\text{Act-Ci}]$$

Figure 7: A rule for context-insensitive activity transition analysis (as assumed in the state of the art).

According to the prior work [1, 2, 4, 9, 14, 23–26, 31], only one abstract object, $\langle o_t, \emptyset\rangle$, (with an empty context $\emptyset$) is created for each activity class t. In addition, the launch mode for t is ignored.

Given the new activity $\langle o_t, \emptyset\rangle$ started, $\texttt{modelActivity}(\langle o_t, \emptyset\rangle)$ tells the pointer analysis to analyze the callbacks of t invoked on $\langle o_t, \emptyset\rangle$. In addition, for the intent object $\langle o_j, c\rangle$, which is now added to o_t`.mIntent` (under context $\emptyset$) and can be retrieved later by calling `getIntent()` on $\langle o_t, \emptyset\rangle$, `modelNewIntent()` serves to inform the pointer analysis to analyze `onNewIntent()` with this intent on the receiver $\langle o_t, \emptyset\rangle$ conservatively (as this may be reused).

3.6 Launch-Mode-Aware Context-Sensitive Activity Transition Analysis

Figure 8 is an analogue of Figure 7 except that our analysis now is launch-mode-aware and builds a context-sensitive ATG, $G = (V, E)$.

The execution of an app is managed by one or more tasks. Every task has a *back stack*, which is used to manage activity navigations (adding a newly created activity to the stack) and backtracking (popping off the finished activity off the stack). A launch mode specifies how a new instance of an activity class is associated with the current task [11]. Developers select launch modes to provide a smooth and consistent user experience or to achieve design requirements such as the singleton pattern. In Figure 3, two different launch modes, `singleTask` and `standard`, for `Home` are illustrated.

With the help of a context-sensitive ATG, different launch modes can now be distinguished and effectively handled during the analysis. The ATG keeps track of activity transitions for every transition path, mimicking the back stack of a task. Let us examine the rules for handling four types of of launch modes: [L-Std] (`standard`), [L-TopReuse] and [L-TopNew] (`singleTop`), [L-TaskReuse] and [L-TaskNew] (`singleTask`), and [L-Inst] (`singleInstance`).

$$\frac{\begin{array}{c}(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Std},p)\rightsquigarrow t \\ hc = heapCtxSelector(o_i,c)\end{array}}{\begin{array}{c} o^t_p = newActObj(t) \\ \langle o^t_p,hc\rangle\in V \quad \langle o_i,c\rangle \hookrightarrow^p \langle o^t_p,hc\rangle \in E \\ \langle o_j,c'\rangle \in fpt(o^t_p,hc,\texttt{mIntent}) \\ \texttt{modelCallBacks}(\langle o^t_p,hc\rangle)\end{array}}\;[\text{L-Std}]$$

$$\frac{(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Top},p)\rightsquigarrow t \quad getType(o_i) = t}{\begin{array}{c}\langle o_i,c\rangle \hookrightarrow^p \langle o_i,c\rangle \in E \\ \texttt{modelNewIntent}(\langle o_i,c\rangle,\langle o_j,c'\rangle)\end{array}}\;[\text{L-TopReuse}]$$

$$\frac{(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Top},p)\rightsquigarrow t \quad getType(o_i) \neq t}{(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Std},p)\rightsquigarrow t}\;[\text{L-TopNew}]$$

$$\frac{\begin{array}{c}(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Task},p)\rightsquigarrow t \\ \langle o_k,c''\rangle \hookrightarrow^* \langle o_i,c\rangle \quad getType(o_k) = t\end{array}}{\begin{array}{c}\langle o_i,c\rangle \hookrightarrow^p \langle o_k,c''\rangle \in E \\ \texttt{modelNewIntent}(\langle o_k,c''\rangle,\langle o_j,c'\rangle)\end{array}}\;[\text{L-TaskReuse}]$$

$$\frac{\begin{array}{c}(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Task},p)\rightsquigarrow t \\ \langle o_k,c''\rangle \not\hookrightarrow^* \langle o_i,c\rangle \quad getType(o_k) = t\end{array}}{(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Std},p)\rightsquigarrow t}\;[\text{L-TaskNew}]$$

$$\frac{(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Inst},p)\rightsquigarrow t}{(\langle o_i,c\rangle,\langle o_j,c'\rangle,\texttt{Task},p)\rightsquigarrow t}\;[\text{L-Inst}]$$

Figure 8: Rules for Chime's launch-mode-aware context-sensitive activity transition analysis.

standard. For the default launch mode, we always create a new instance of t, identified by $\langle o^t_p, hc\rangle$, where p is the line number of its corresponding `startActivity()`, and then deliver its passed-in intent to this new activity object. While the ATG grows, $\texttt{modelCallBacks}(\langle o^t_p, hc\rangle)$ serves again (as in [Act-Ci]) to request the callbacks on the new activity $\langle o^t_p, hc\rangle$ to be analyzed now.

singleTop. If the activity to be started has the same type as the top activity, then the top activity is reused ([L-TopReuse]), with its `onNewIntent()` reanalyzed (`modelNewIntent()`). Otherwise, we handle it identically as in the case of `standard` ([L-TopNew]).

singleTask. This mode is similar to `singleTop`, except that the activity instance closest to the top of the back stack will be reused if it has the same type as the new activity to be started ([L-TaskReuse]). Otherwise, we fall back to the case where `standard` is handled ([L-TaskNew]). Here, $\langle o_k, c''\rangle \hookrightarrow^* \langle o_i, c\rangle$ represents the standard graph reachability on a directed graph except that it disallows the existence of any intermediate node $\langle o_s, c'''\rangle$ such that $\langle o_s, c'''\rangle \hookrightarrow^* \langle o_i, c\rangle$ also holds, where $getType(o_k) = getType(o_s)$.

singleInstance. This mode is similar to `singleTask`, except that only one instance of its activity class resides in its task. Therefore, this mode is handled identically as `singleTask` ([L-Inst]).

3.7 Discussion

In our evaluation, we will compare Chime with CiChime, a version of Chime, with [ACT-CI] used for building the ATG for an app. Given the same object-sensitive pointer analysis applied to a given app, Chime's ATG is strictly non-less precise than CiChime's ATG as the former is context-sensitive but the latter is not.

Thus, the pointer analysis under Chime will also be strictly non-less precise than the one under CiChime. With a context-sensitive ATG, Chime enables an activity's callbacks to be analyzed context-sensitively. With a context-insensitive ATG, however, CiChime allows such callbacks to be analyzed only context-insensitively.

3.8 Example

Let us go through some of our rules for the example in Figure 3 (with `Home` configured for `singleTask` and the others for `standard`) to see how Chime builds the context-sensitive ATG in Figure 3(d). Initially, the ATG contains $\langle \mathrm{o}^{\mathtt{main}}_{\mathsf{HM}}, \emptyset \rangle$, an instance of `Home` allocated when its harness `main()` is analyzed ([P-NewAct]). Let o_4 an o_9 be the intent objects resolved at lines 4 and 9 respectively, so that we obtain `"NewTrip"` $\in fpt(o_4, \_, \mathtt{tgtClassName})$ and `"TripView"` $\in fpt(o_9, \_, \mathtt{tgtClassName})$ ([I-Ex]). By applying [I-Act] to lines 5 and 10, the two ICC call relations $(\langle o^{\mathtt{main}}_{\mathsf{HM}}, \emptyset \rangle, \langle o_4, \_ \rangle, \mathtt{Std}, 5) \rightsquigarrow$ `NewTrip` and $(\langle o^{\mathtt{main}}_{\mathsf{HM}}, \emptyset \rangle, \langle o_9, \_ \rangle, \mathtt{Std}, 10) \rightsquigarrow$ `TripView` are established. By further applying [L-Std] to these two relations, where $heapCtxSelector(o^{\mathtt{main}}_{\mathsf{HM}}, \emptyset) = [o^{\mathtt{main}}_{\mathsf{HM}}]$, we obtain two new activity objects o^{NT}_5 and o^{TV}_{10}. In addition, two ATG edges $\langle o^{\mathsf{HM}}_{\mathtt{main}}, \emptyset \rangle \hookrightarrow^5 \langle o^{\mathsf{NT}}_5, [o^{\mathsf{HM}}_{\mathtt{main}}] \rangle$ and $\langle o^{\mathsf{HM}}_{\mathtt{main}}, \emptyset \rangle \hookrightarrow^{10} \langle o^{\mathsf{TV}}_{10}, [o^{\mathsf{HM}}_{\mathtt{main}}] \rangle$ are discovered.

Similarly, we can resolve the ICC calls at lines 16 and 23. Let o_{15} an o_{22} be the intent objects resolved at lines 15 and 22, respectively, so that we obtain `"EditFolder"` $\in fpt(o_{15}, \_, \mathtt{tgtClassName})$ and `"EditFolder"` $\in fpt(o_{22}, \_, \mathtt{tgtClassName})$ ([I-Ex]). At line 16, we obtain $(\langle o^{\mathsf{NT}}_5, [o^{\mathsf{HM}}_{\mathtt{main}}] \rangle, \langle o_{15}, \_ \rangle, \mathtt{Std}, 16) \rightsquigarrow$ `EditFolder` ([I-Act]). By applying ([L-Std]) to this relation, where $heapCtxSelector(o^{\mathsf{NT}}_5, [o^{\mathsf{HM}}_{\mathtt{main}}]) = [o^{\mathsf{NT}}_5, o^{\mathsf{HM}}_{\mathtt{main}}]$, we obtain $(\langle o^{\mathsf{NT}}_5, [o^{\mathsf{HM}}_{\mathtt{main}}] \rangle \hookrightarrow^{16} \langle o^{\mathsf{EF}}_{16}, [o^{\mathsf{NT}}_5, o^{\mathsf{HM}}_{\mathtt{main}}] \rangle$. At line 23, we obtain $(\langle o^{\mathsf{TV}}_{10}, [o^{\mathsf{HM}}_{\mathtt{main}}] \rangle, \langle o_{22}, \_ \rangle, \mathtt{Std}, 23) \rightsquigarrow$ `EditFolder` ([I-Act]). By applying ([L-Std]) to this relation, where $heapCtxSelector(o^{\mathsf{TV}}_{10}, [o^{\mathsf{HM}}_{\mathtt{main}}]) = [o^{\mathsf{TV}}_{10}, o^{\mathsf{HM}}_{\mathtt{main}}]$, we obtain $(\langle o^{\mathsf{TV}}_{10}, [o^{\mathsf{HM}}_{\mathtt{main}}] \rangle \hookrightarrow^{23} \langle o^{\mathsf{EF}}_{23}, [o^{\mathsf{TV}}_{10}, o^{\mathsf{HM}}_{\mathtt{main}}] \rangle$.

Finally, we consider the `startActivity()` at line 34. Let o_{33} be the intent object resolved at line 33, so that `"Home"` $\in fpt(o_{33}, \_, \mathtt{tgtClassName})$ ([I-Ex]). Two relations will be established at line 34, $(\langle o^{\mathsf{EF}}_{16}, [o^{\mathsf{NT}}_5, o^{\mathsf{HM}}_{\mathtt{main}}] \rangle, \langle o_{33}, \_ \rangle, \mathtt{Task}, 34) \rightsquigarrow$ `Home` and $(\langle o^{\mathsf{EF}}_{23}, [o^{\mathsf{TV}}_{10}, o^{\mathsf{HM}}_{\mathtt{main}}] \rangle, \langle o_{33}, \_ \rangle, \mathtt{Task}, 34) \rightsquigarrow$ `Home`. As `Home` uses `singleTask` launch mode, the final ATG is shown in Figure 3(d).

4 EVALUATION

The objective of our evaluation is to demonstrate that our context-sensitive activity transition analysis is more precise than its context-insensitive counterpart in modeling activity transitions, guiding GUI testing, and improving the pointer analysis precision.

4.1 Implementation

Chime is built on top of the Spark pointer analysis framework in DroidSafe [9]. We leverage the Android Device Implementation (ADI) provided by DroidSafe [9], a comprehensive modeling of the Android framework and Java library, to improve the precision and soundness of our analysis. DroidSafe is used to decompile an app, generate a slim harness `main()`, and obtain app matadata from its manifest file. To resolve reflection, we use Ripple [34] but may consider a hybrid analysis [18] in future work.

4.2 Experimental Setup and Methodology

We have selected a total of 42 Android apps in the top-chart free apps from Google Play downloaded on 30 November 2016. Our $k-$object-sensitive pointer analysis is configured to be 2obj+h (with two elements for a method context and one for a heap context), which is a widely used configuration for achieving the best tradeoff between precision and scalability [12, 17, 28–30].

Our experiments are conducted on a Xeon E5-1660 3.2GHz machine with 256GB RAM running Ubuntu 16.04 LTS. The analysis time of every app is the average of three runs.

Chime models activity transitions context-sensitively and distinguishes activity launch modes. To demonstrate the benefits of our approach, we compare Chime with CiChime, a simplified version of Chime (realized with [Atg-Ai]), which models activity transitions context-insensitively, and consequently, incapable of handling or distinguishing activity launch modes adequately.

Our evaluation answers the following research questions (RQs):

- **RQ1.** Is Chime able to distinguish an activity along different transition paths compared with CiChime?
- **RQ2.** Is Chime capable of analyzing launch modes effectively?
- **RQ3.** is Chime useful for facilitating GUI testing?
- **RQ4.** is Chime able in improving the pointer analysis precision?

4.3 Results and Analysis

4.3.1 ***RQ1. Activity Transition.*** Table 1 compares Chime and CiChime in terms of the ATGs constructed for the 42 apps evaluated. For each app, the context-sensitive ATG from Chime is usually larger than the context-insensitive ATG from CiChime (Columns "ATG Node" and "ATG Edge"). To see why Chime can distinguish different activity transition sequences leading to an activity better than CiChime, we rely on the maximal/average/minimal number of the class types of the predecessor activities of an activity in an ATG (Columns "Max/Avg/Min Typ"). The "Max/Avg/Min Typ" for Chime is always no larger than that for CiChime, showing that Chime can distinguish different activities that would otherwise be merged under CiChime. Figure 9 plots the same data measured by these three metrics. Finally, Chime is more costly in maintaining context-sensitivity. The average pointer analysis time spent per app is 183.9 seconds under CiChime but 472.6 seconds under Chime.

4.3.2 ***RQ2. Launch Modes.*** Launch modes for activities can significantly affect how activities are created and their transitions. We first examine the widespread use of special launch modes, `singleTask`, `singleTop` and `singleInstance`, other than just `standard` in real-world Android apps. We then present and discuss

Table 1: Comparing CHIME and CICHIME in terms of ATGs constructed. For each app, Column "ATG Node" ("ATG Edge") gives the number of nodes (edges) in an ATG, Column "Max/Avg/Min Typ" represents the maximal/average/minimal number of the class types of the predecessor activities of an activity, and Column "PTA Time" gives the pointer analysis time in seconds.

App ID	App Name	CICHIME						CHIME					
		ATG Node	ATG Edge	Max Typ	Avg Typ	Min Typ	PTA Time	ATG Node	ATG Edge	Max Typ	Avg Typ	Min Typ	PTA Time
1	Instagram	12	171	10	3.8	1	238	288	6440	2	1.2	1	844
2	GoogleLite	4	12	2	1.7	1	16	11	39	1	1	1	25
3	Raider	6	7	2	1.2	1	19	17	21	1	1	1	29
4	Speed	11	19	6	1.6	1	34	13	20	6	1.6	1	54
5	FlashLight	5	11	4	3	2	68	13	25	1.5	1.3	1	108
6	Booster	6	5	3	2	1	30	6	5	3	2	1	45
7	Seek	4	8	4	2	1	19	11	15	1	1	1	29
8	ANZ	7	187	7	5	1	23	50	1182	1	1	1	41
9	Antivirus	26	244	20	8.2	1	508	369	5101	17	2.3	1	1800
10	Solitaire	2	6	1	1	1	66	26	126	1	1	1	148
11	Clean	8	7	3	1.7	1	34	9	7	3	1.7	1	53
12	PhoneClean	16	29	9	2	1	44	28	61	4.2	1.3	1	80
13	Pool	2	19	2	1.5	1	18	3	28	1	1	1	26
14	PayPal	15	221	15	4	1	19	90	1943	1	1	1	77
15	ClashRoyale	2	5	1	1	1	15	2	5	1	1	1	21
16	MobileStrike	5	13	3	2	1	18	15	45	1	1	1	27
17	Cleaner	7	6	3	1.7	1	30	7	6	3	1.7	1	48
18	Domino	3	11	2	1.3	1	19	16	47	1.8	1.3	1	31
19	Mail	21	710	21	9.8	2	2960	364	5216	2.6	1	1	6252
20	VLC	11	71	6	2.9	1	17	168	5357	2.8	1.4	1	105
21	PowerClean	21	45	8	3.1	1	81	80	77	1	1	1	301
22	Opal	10	38	9	2.8	1	26	46	113	1	1	1	50
23	RollingSky	9	242	9	3.8	1	288	37	998	1	1	1	439
24	StarWars	4	9	2	1.7	1	24	8	11	1.5	1.2	1	35
25	ClashofClan	2	5	1	1	1	15	2	5	1	1	1	21
26	TripView	12	14	2	1.1	1	11	28	41	1	1	1	19
27	FireFox	11	57	7	2.8	2	71	124	272	2	1.1	1	190
28	PowerCleaner	27	19	4	1.3	1	44	77	110	1	1	1	119
29	TalkingAngela	3	10	1	1	1	38	5	10	1	1	1	62
30	SpeedTest	3	3	2	1.5	1	83	6	6	1	1	1	136
31	TalkingTom	3	10	1	1	1	42	5	10	1	1	1	67
32	Slots	4	41	2	1.3	1	12	7	103	1	1	1	17
33	HealthEngine	8	149	8	5.6	1	73	113	7713	1	1	1	210
34	Tiles	18	675	18	5.4	1	824	391	23589	2	1.1	1	2901
35	HillClimb	4	119	4	2.3	1	84	8	236	1	1	1	126
36	HungryJacks	11	34	10	3.4	1	45	52	152	1.6	1.1	1	80
37	MachineZone	5	13	3	2	1	17	15	45	1	1	1	27
38	DeathWorm	3	5	3	2.5	2	31	6	9	1	1	1	47
39	Wallet	18	186	12	4.9	1	166	426	11260	10.1	2.3	1	867
40	Legends	4	9	3	1.5	1	12	41	165	1	1	1	28
41	Catch	40	232	21	3.1	1	1312	216	953	18	1.8	1	2398
42	K9	20	78	8	3.2	1	231	277	1775	2	1	1	1868

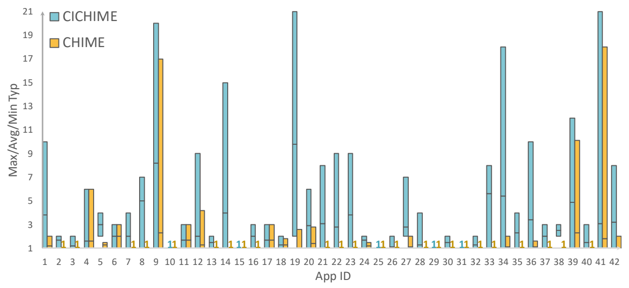


Figure 9: Comparing CICHIME and CHIME graphically in terms of "Max/Avg/Min Typ" from Table 1. For each bar, its top edge represents "Max Typ", its bottom edge represents "Min Typ" and the dash in the bar represents "Avg Typ". In the case of a "1", "Max Typ" = "Avg Typ" = "Min Typ" = 1.

the results of our launch mode analysis. Finally, we use a real-world Android app to demonstrate the effectiveness of our activity transition analysis in handling special launch modes.

Figure 10 present our results on the activity classes analyzed by CHIME in all the 42 apps, with default and special launch modes. Note that for each activity class, only one launch mode is counted (statically). We can see that 30% of all activity classes are configured with special launch modes. Developers often use special launch modes for some activities in real-world Android apps in order to provide a smooth and consistent user experience. Such special launch modes are also used to achieve certain design requirements such as the singleton pattern. Therefore, it is important to develop a launch-mode-aware analysis to understand activity transitions.

Let us examine why CHIME can but CICHIME cannot handle four types of activity launch modes, The key difference between a special launch mode and the default mode (i.e., `standard`) is that the former may cause an existing activity to be resumed but the latter will always cause a new activity to be created.

Without context-sensitivity, CICHIME represents all instances of an activity class with one single abstract object. As a result, different launch modes cannot be distinguished. In a context-insensitive ATG constructed, its edges are unaware of the launch modes used.

With context-sensitivity, CHIME can distinguish different launch modes as it can distinguish different instances of an activity class under different activity transition sequences. In a context-sensitive ATG constructed, its edges are essentially annotated with the launch modes used. To see this, let us distinguish two types of edges in an ATG: (1) `StdEdge` corresponding to `standard` and (2) `SplEdge` corresponding to `singleTask`, `singleTop`, or `singleInstance`.

Table 2 gives our results for individual apps. For each app, while many activity classes are configured with the default launch mode, many others use special launch modes (Columns "Std" and "Spl"). Note that these two columns contain the raw data used for plotting Figure 10. For each app, its ATG consists of not only edges marked by `StdEdge` but also many other edges marked by `SplEdge` (Columns "StdEdge" and "SplEdge"). For some apps, such as `Seek`, an activity class configured with a special launch mode may not have any incoming edge of type `SplEdge` since only one instance is ever created. On the other hand, for *Catch*, 14 activity classes with special launch modes result in 456 edges of type `SplEdge`.

For all the 42 apps combined, there are 275 and 138 activity classes configured for the default and special launch modes, respectively (Figure 10). In the 42 ATGs constructed, there are 59275 and 14067 edges marked with `StdEdge` and `SplEdge`, respectively.

Finally, let us conduct a case study, by considering the code snippet from `Catch` in Figure 11. The launch mode for `MainHome` (`MH`) is `singleTask` but the launch modes for `SearchActivity` (`SA`) and `ShoppingCartView` (`SC`) are irrelevant here. `MH` launches `SA` at ① and `SA` launches (`SC`) at ②. Afterwards, `SC` launches `MH` again at ③. CHIME generates a launch-mode-aware ATG to capture the activity transitions, by considering `singleTask` for `MH` at ③ ([L-TKPREV] in Section 3.6), so that the existing `MH` is resumed. Thus, the activity transitions $\begin{tikzcd} o^{MH}_{main} \arrow[r] & o^{SA}_{9} \arrow[r] & o^{SC}_{14} \arrow[ll, bend left] \end{tikzcd}$ are precisely captured.

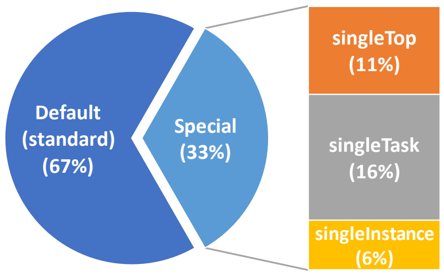
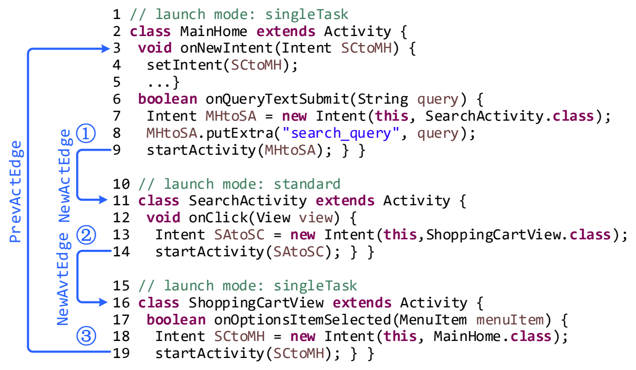


Figure 10: Percentage of the analyzed activity classes with the default and special launch modes in the 42 apps.

Table 2: Classification of ATG edges under the default and special modes. For each app, Columns "Std" and "Spl" give the number of different activity classes configured with the default and special launch modes, respectively. For each ATG, Columns "StdEdge" and "SplEdge" give the number of edges marked by `StdEdge` and `SplEdge`, respectively.

App ID	App Name	Launch Mode		ATG Edge		App ID	App Name	Launch Mode		ATG Edge	
		Std	Spl	StdEdge	SplEdge			Std	Spl	StdEdge	SplEdge
1	Instagram	9	3	5446	994	22	Opal	9	1	113	0
2	GoogleLite	4	0	39	0	23	RollingSky	8	1	998	0
3	Raider	4	2	20	1	24	StarWars	3	1	9	2
4	Speed	4	7	10	10	25	ClashofClan	1	1	2	3
5	FlashLight	3	2	23	2	26	TripView	12	0	41	0
6	Booster	2	4	2	3	27	FireFox	7	4	255	17
7	Seek	1	3	15	0	28	PowerCleaner	5	22	110	0
8	ANZ	7	0	1182	0	29	TalkingAngela	1	2	3	7
9	Antivirus	17	9	4632	469	30	SpeedTest	2	1	6	0
10	Solitaire	2	0	126	0	31	TalkingTom	1	2	3	7
11	Clean	2	6	4	3	32	Slots	2	2	102	1
12	PhoneClean	7	9	31	30	33	HealthEngine	8	0	7713	0
13	Pool	1	1	27	1	34	Tiles	15	3	20805	2784
14	PayPal	13	2	1883	60	35	HillClimb	3	1	236	0
15	ClashRoyale	1	1	2	3	36	HungryJacks	10	1	131	21
16	MobileStrike	5	0	45	0	37	MachineZone	5	0	45	0
17	Cleaner	2	5	3	3	38	DeathWorm	2	1	9	0
18	Domino	1	2	37	10	39	Wallet	12	6	2623	8637
19	Mail	17	4	5106	110	40	Legends	3	1	164	1
20	VLC	7	4	4957	400	41	Catch	26	14	497	456
21	PowerClean	13	8	76	1	42	K9	18	2	1744	31


```
// launch mode: singleTask
class MainHome extends Activity {
 void onNewIntent(Intent SCtoMH) {
  setIntent(SCtoMH);
  ...}
 boolean onQueryTextSubmit(String query) {
  Intent MHtoSA = new Intent(this, SearchActivity.class);
  MHtoSA.putExtra("search_query", query);
  startActivity(MHtoSA); } }

// launch mode: standard
class SearchActivity extends Activity {
 void onClick(View view) {
  Intent SAtoSC = new Intent(this,ShoppingCartView.class);
  startActivity(SAtoSC); } }

// launch mode: singleTask
class ShoppingCartView extends Activity {
 boolean onOptionsItemSelected(MenuItem menuItem) {
  Intent SCtoMH = new Intent(this, MainHome.class);
  startActivity(SCtoMH); } }
```

Figure 11: Activity transitions tracked by CHIME in `Catch`.

4.4 RQ3. GUI Testing

Existing static analyses, TRIMDROID [23], BRAHMASTRA [4] and A^{3}E [2], rely on context-insensitive ATGs to guide GUI testing. For this client, CHIME is expected to be more effective than CICHIME as context-sensitive ATGs are more beneficial than context-insensitive ATGs. For example, BRAHMASTRA et al. [4] instrument an app to track a transition path from one activity to another based on a context-insensitive ATG for the app in order to find security vulnerabilities at run time. Due to context-insensitivity, the user has to work out the transition paths between the two activities to be tested, with potentially some infeasible transition paths being instrumented redundantly (causing false positives). This problem can be alleviated if a context-sensitive ATG is used for the app, since the transition paths between two activities are available context-sensitively (with the number of infeasible transitions also being reduced).

To demonstrate that CHIME can be more effective than CICHIME in guiding GUI testing, Table 3 compares both in terms of the average number of activity classes (NAC) reaching an activity class in an ATG (with each cycle in an ATG counted only once). In a context-sensitive ATG, different nodes may represent different instances of a common class. In this case, the NAC for the class is the average of the NACs on the transition paths reaching all such nodes. For a given app, the smaller NAC is, the less effort is expected to be spent on GUI testing. In general, CHIME's NAC is always no larger than CICHIME's NAC. For 10 out of the 42 apps evaluated, both are indistinguishable. There are two different reasons behind (Table 1). In the case of `Booster`, `CashRoyale`, `Cleaner` and `ClashofClan`, CHIME and CICHIME produce the same ATG for each app. In the case of the other 6 apps, the ATGs built by CHIME and CICHIME for an app are different but their NACs happen to be the same.

Let us take a look at some specific activity classes. Consider `PostTwitterActivity` from `HungryJacks`. For CICHIME, it is represented by one abstract object in the ATG for `HungryJacks`. The number of activity classes reaching it is 11. For CHIME, this abstract object is split into 7 context-sensitive instances. The average number of activity classes reaching these instances remains to be 11. Consider `SignInActivity` from `Opal`. For CICHIME, it is also represented by one abstract object in the ATG for `Opal`. The number of activity classes reaching it is 9. For CHIME, this abstract object is now split into 24 context-sensitive instances. The average number of activity classes reaching these instances is 4.67.

4.5 RQ4. Pointer Analysis

Given an app, its ATG is context-sensitive under CHIME but context-insensitive under CICHIME. Thus, CHIME is more precise than CICHIME as the the callbacks of an activity are analyzed context-sensitively under CHIME but context-insensitively under CICHIME.

Table 4 compares CHIME and CICHIME in terms of the average size of the points-to sets of the intent variables accessed in the calls to `onCreate()` in all the activity classes in an app. For CHIME, the size of the points-to set of the intent variable accessed in a `onCreate()` call is the average of the sizes of the points-to sets under all its contexts analyzed. Recall how `intentEF` in `EditFolder` in Figure 4 is analyzed (Section 2.4), where `EditFolder` may be started from two different contexts, `NewTrip` and `TripView` (Figure 3(a)). As CICHIME is context-insensitive, `intentEF` in `onCreate()` will point to the two intent objects passed both contexts indiscriminately. However, CHIME can distinguish the two intents for `intentEF` as `onCreate()` is analyzed separately under these two contexts.

Table 3: Comparing Chime and CiChime in terms of the average number of activity classes reaching an activity class in an ATG (measuring the effort spent in GUI testing).

App ID	App Name	CiChime	Chime	App ID	App Name	CiChime	Chime
1	Instagram	7.25	5.12	22	Opal	4.10	3.15
2	GoogleLite	1.75	1.67	23	RollingSky	2.44	1.56
3	Raider	2.17	1.78	24	StarWars	2.50	1.88
4	Speed	2.55	2.36	25	ClashofClan	1.50	1.50
5	FlashLight	1.80	1.35	26	TripView	2.00	1.93
6	Booster	1.50	1.50	27	FireFox	3.45	2.42
7	Seek	2.25	2.00	28	PowerCleaner	4.33	3.21
8	ANZ	4.71	4.32	29	TalkingAngela	1.33	1.33
9	Antivirus	10.12	8.23	30	SpeedTest	1.67	1.50
10	Solitaire	1.50	1.50	31	TalkingTom	1.33	1.33
11	Clean	1.50	1.50	32	Slots	2.50	2.08
12	PhoneClean	4.56	4.56	33	HealthEngine	6.25	5.78
13	Pool	1.50	1.50	34	Tiles	8.39	7.12
14	PayPal	7.40	6.12	35	HillClimb	2.25	2.00
15	ClashRoyale	1.50	1.50	36	HungryJacks	5.09	4.14
16	MobileStrike	1.80	1.73	37	MachineZone	1.80	1.73
17	Cleaner	1.57	1.57	38	DeathWorm	2.00	1.89
18	Domino	2.33	1.95	39	Wallet	9.11	8.47
19	Mail	8.86	7.21	40	Legends	2.00	1.96
20	VLC	4.45	3.70	41	Catch	7.78	5.64
21	PowerClean	2.29	1.42	42	K9	8.00	6.18

Table 4: Comparing Chime and CiChime in terms of the average size of the points-to sets of the intent variables accessed in the calls to `onCreate()` in all activity classes.

App ID	App Name	CiChime	Chime	App ID	App Name	CiChime	Chime
1	Instagram	16.33	10.78	22	Opal	1.30	1.30
2	GoogleLite	3.00	2.58	23	RollingSky	6.44	5.79
3	Raider	5.33	3.80	24	StarWars	5.25	4.00
4	Speed	4.45	4.00	25	ClashofClan	2.00	1.60
5	FlashLight	5.60	5.00	26	TripView	4.08	3.92
6	Booster	4.17	3.83	27	FireFox	20.82	8.98
7	Seek	2.75	2.75	28	PowerCleaner	2.78	2.51
8	ANZ	4.14	3.86	29	TalkingAngela	8.67	7.92
9	Antivirus	11.42	7.76	30	SpeedTest	6.33	6.33
10	Solitaire	6.50	4.21	31	TalkingTom	9.33	8.58
11	Clean	4.75	4.13	32	Slots	4.75	4.38
12	PhoneClean	4.88	4.10	33	HealthEngine	4.50	3.87
13	Pool	2.00	2.00	34	Tiles	8.39	6.32
14	PayPal	8.00	7.59	35	HillClimb	3.75	3.75
15	ClashRoyale	2.00	1.60	36	HungryJacks	3.91	3.55
16	MobileStrike	4.00	3.60	37	MachineZone	4.00	3.60
17	Cleaner	4.14	3.86	38	DeathWorm	3.67	3.67
18	Domino	8.67	6.78	39	Wallet	13.78	5.37
19	Mail	17.19	11.00	40	Legends	6.00	4.00
20	VLC	5.91	5.08	41	Catch	8.60	5.94
21	PowerClean	5.43	4.19	42	K9	19.95	8.59

Given an app, the average size per points-to set under Chime is always no larger than that under CiChime. For `Booster`, `CashRoyale`, `Cleaner` and `ClashofClan`, Chime and CiChime build the same ATG in each case (Table 1). However, Chime is more precise than CiChime as Chime allows an activity's callbacks to be analyzed context-sensitively. For `Seek`, `Pool`, `Opal`, `SpeedTest`, `HillClimb` and `DeathWorm`, Chime and CiChime build different ATGs (Table 1) but are equally precise. Let us see why by considering `Seek` with four activity classes, (1) `MainActivity`, (2) `AuthActivity`, (3) `HardUpgradeActivity`, and (4) `com.adobe.mobile.ar`, where (1) starts (2) twice using a common intent, (1) starts (3) twice using a common intent, and (1) – (3) start (4) using a common intent. Distinguishing calling contexts for each `onCreate()` is not beneficial.

5 RELATED WORK

Pointer Analysis. Object-sensitive pointer analysis proposed by Milanova et al. [21, 22] is known to be the best context-sensitive pointer analysis for object-oriented programming languages [12, 13, 15–17, 28–30]. Type sensitivity [28] and hybrid context-sensitivity [12] are two variants based on object-sensitivity. The former approximates (roughly) the objects created at allocation sites by their dynamic types, trading precision for scalability and efficiency. The latter applies call-site-sensitivity to static calls and object-sensitivity to virtual calls. Recently, Tan et al. improve object-sensitivity pointer analysis to achieve better precision by removing redundant context elements [29] and better scalability in building call graphs by targeting type-dependent clients [30].

However, existing context-sensitive pointer analyses are insufficient in analyzing the Android-specific APIs, such as ICC calls, which trigger a large number of callbacks interacting with the Android framework (e.g., activity objects created therein). In contrast, Chime is aware of the activity-related APIs so that activity transitions are modeled context-sensitively, thereby improving the precision of the underlying object-sensitive pointer analysis.

Static Android App analysis. FlowDroid [1] is a popular open-source static information flow analysis, which constructs a harness method to model the behaviors of the Android components in an app. Then a context-insensitive pointer analysis is performed to construct its call graph and inter-procedural control-flow graph, against which some intra-component taint analysis is performed. IccTA [14] represents a static taint analysis built on top of FlowDroid to detect inter-component privacy leaks. It queries the ICC analysis IC3 [25] to instrument the target activities started at ICC calls by using a flow- and context-insensitive pointer analysis.

DroidSafe [9] detects information leaks by applying an object-sensitive pointer analysis to resolve the intents associate with ICC calls. Amandroid [31] is a static inter-component data-flow analysis framework for supporting security vetting tasks, by using a call-site-sensitive and flow-sensitive pointer analysis to construct the call graph for an app. Despite the fact that both frameworks use context-sensitive pointer analysis to construct call graphs, all the activities are still modeled context-insensitively.

Unlike these earlier approaches, Chime distinguishes different activities launched under different launch modes along different transition sequences by using context-sensitive ATGs, which are constructed on-the-fly during the object-sensitive pointer analysis for the entire app. Such context-sensitive ATGs can help improve the efficiency of GUI testing tools [2, 4, 23].

6 CONCLUSION

We present Chime, a launch-mode-aware context-sensitive activity transition analysis for Android apps. Chime models the activity transitions under different launch modes context-sensitively, together with an object-sensitive pointer analysis. Our evaluation shows that our context-sensitive activity transition analysis is more precise than its context-insensitive counterpart in modeling activity transitions, facilitating GUI testing, and improving the pointer analysis precision for real-world Android apps.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[2] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[3] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. ACM, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313

[4] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security '14)*. USENIX Association, San Diego, CA, 1021–1036.

[5] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *23rd USENIX Security Symposium (USENIX Security '14)*. USENIX Association, San Diego, CA, 1037–1052.

[6] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

[7] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, USA, 576–587. https://doi.org/10.1145/2635868.2635869

[8] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2017. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *The 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[9] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *The 2015 Network and Distributed System Security Symposium (NDSS '15)*.

[10] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

[11] Google Inc. 2017. Tasks and Back Stack. (2017). https://developer.android.com/guide/components/activities/tasks-and-back-stack.html

[12] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

[13] Ondrej Lhoták and Laurie Hendren. 2006. Context-sensitive points-to analysis: is it worth it?. In *Proceedings of the 15th International Conference on Compiler Construction (CC '06)*. Springer, 47–64.

[14] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 280–291. http://dl.acm.org/citation.cfm?id=2818754.2818791

[15] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing Reflection Resolution for Java. In *28th European Conference on Object-Oriented Programming (ECOOP '14)*. 27–53. https://doi.org/10.1007/978-3-662-44202-9_2

[16] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective Soundness-Guided Reflection Analysis. In *22nd International Static Analysis Symposium (SAS '15)*. 162–180. https://doi.org/10.1007/978-3-662-48288-9_10

[17] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (ECOOP '16)*. 15:1–15:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

[18] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. 2017. Reflection Analysis for Java: Uncovering More Reflective Targets Precisely. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE '17)*. 12–23. https://doi.org/10.1109/ISSRE.2017.36

[19] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/2382196.2382223

[20] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896

[21] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/566172.566174

[22] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[23] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 559–570. https://doi.org/10.1145/2884781.2884853

[24] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining Static Analysis with Probabilistic Models to Enable Market-scale Android Inter-component Analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 469–484. https://doi.org/10.1145/2837614.2837661

[25] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 77–88. http://dl.acm.org/citation.cfm?id=2818754.2818767

[26] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *22nd USENIX Security Symposium (USENIX Security '13)*. 543–558.

[27] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 2014. Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 56–65. https://doi.org/10.1145/2664243.2664275

[28] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

[29] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium (SAS '16)*. Springer, 489–510. https://link.springer.com/chapter/10.1007/978-3-662-53413-7_24

[30] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[31] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[32] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*, Vol. 1. 89–99. https://doi.org/10.1109/ICSE.2015.31

[33] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 658–668. https://doi.org/10.1109/ASE.2015.76

[34] Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. 2017. Ripple: Reflection Analysis for Android Apps in Incomplete Information Environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17)*. ACM, New York, NY, USA, 281–288. https://doi.org/10.1145/3029806.3029814