Launch-Mode-Aware Context-Sensitive Activity Transition Analysis for Android Apps

Anonymous Author(s)

ABSTRACT

A fundamental static analysis in Android is to model activity transitions in the presence of nondeterministic event-driven callbacks. The analysis paves the way for various clients, such as information leakage detection and GUI testing. Existing static Android analyses conservatively model the activity transitions by constructing a "fat" harness main method consisting of the object allocation for every component and its life-cycle callbacks, against which pointer analysis and subsequent client analyses are performed. However, separating activity transition modeling from a subsequent pointer analysis leads to context-insensitive transitions even when a context-sensitive pointer analysis is used, resulting in a large number of infeasible transition paths.

This paper presents Chime, a context-sensitive activity transition analysis that efficiently and precisely resolves activity transitions together with object-sensitive pointer analysis. The key idea is to use activity transition sequences as contexts to distinguish an activity in different transition paths. By considering different launch-modes that affect the life-cycles of an activity, Chime further boosts the precision of activity transition modeling for analyzing real-world Android apps. Our experimental evaluation shows that our context-sensitive activity transition analysis is more precise than its context-insensitive counterpart in terms of removing infeasible transitions and facilitating GUI testing.

1 INTRODUCTION

Activity, as a major type of Android components, lies at the heart of 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.

A fundamental static analysis in Android is to model activity transitions in the presence of event-driven callbacks. It serves as a cornerstone for a wide variety of clients, such as security vulnerability detection [5, 6, 10, 11, 15, 17, 25, 31, 34], malware detection and mitigation [7, 8, 25], GUI testing [2–4, 18, 21], and GUI model generation [32, 33].

The core data structure of static activity transition modeling is *activity transition graph* (ATG) [2, 4, 21], which approximates all possible activity transitions in an Android app. In an ATG, each node represents an activity instance, and an edge between two nodes denotes a transition.

Activities interact with each other through inter-component communication (ICC) using an intent object specifying the target activity. An activity instance can be launched in one of the four launch modes (i.e., `standard`, `singleTask`, `singleTop` and `singleInstance`), which is either configured in `AndroidManifest.xml` or specified in the intent object. These

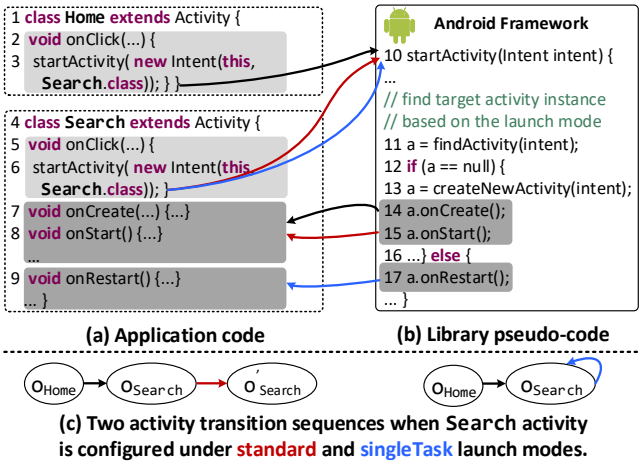


Figure 1: Three activity transitions, represented by ⟶, ⟶ and ⟶, via Android callbacks: `Home` launches `Search`, `Search` launches itself with `standard` and `singleTask` modes.

modes affect the transitions and life-cycles of an activity. For example, the Android framework always creates a new activity instance if the activity is launched in the `standard` mode, while `singleTask` allows only one instance for each transition path at runtime.

Figure 1 gives an example to demonstrate activity transitions in the presence of Android ICC callbacks with the `Search` activity being configured as `standard` and `singleTask` (Figure 1(c)). We assume a navigation scenario where a user clicks a button on the current `Home` activity to launch a `Search` activity via the ICC call `startActivity()` at line 3 in Figure 1(a). Next, the user clicks another button on the launched `Search` activity at line 6 to either open a new `Search` activity (configured with `standard` mode) or reuse the existing one for his/her search (with `singleTask` mode).

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 configured launch mode. A new instance is created (line 13) if the activity is launched the first time, e.g., $o_{Home} \longrightarrow o_{Search}$, or launched in `standard` mode, e.g., transition $o_{Search} \longrightarrow o'_{Search}$, where o_{Search} and o'_{Search} are two different instances. Finally, the framework launches `Search` via the callbacks (lines 14-15) to the life-cycle methods (lines 7-8). If the activity is configured with a special mode, e.g., `singleTask`, The Android framework retrieves the existing instance to restart the activity (line 17) as illustrated by transition o_{Search} ⟲ .

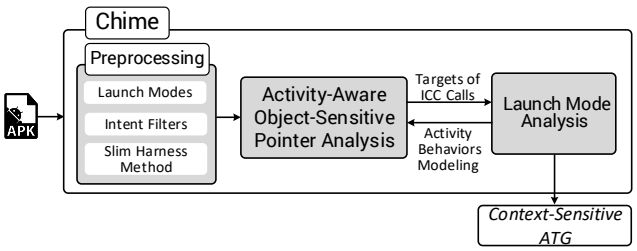


Figure 2: An overview of Chime.

Developing precise static analysis 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, i.e., activities are implicitly launched by the Android framework through nondeterministic user and system events, which significantly complicate static activity transition analysis in the presence of a substantial number of callbacks. Activity transitions through inter-component communication (ICC) use intent objects for specifying target activities. It requires a precise pointer analysis to compute the values of intents. However, the heap allocation sites of an activity object are not visible in the app code, instead, they are distributed via deep call chains in the Android framework. Performing a precise context-sensitive pointer analysis over the entire Android framework (millions of lines of code) is unrealistic [10].

To trade precision for efficiency, existing approaches [1, 10, 15, 31] simplify the activity transitions by modeling every activity object in a context-insensitive manner, i.e., constructing a "fat" harness main method consisting of the object creation for every activity and its life-cycle callbacks (e.g., `onCreate()`), against which an objective-sensitive pointer analysis [10] or a call-site-sensitive pointer analysis [31] is performed. However, separating activity transition modeling from a subsequent pointer analysis this way leads to context-insensitive transitions even when a context-sensitive pointer analysis is used [10, 31], resulting in a large number of infeasible transition paths in a conservative ATG, i.e., two activity instances along different transition paths are merged at the joint points of ATG.

Moreover, an activity's launch mode, which is often ignored by the existing static analyses, affects activity transitions as illustrated in Figure 1, since an activity can be launched for different purposes in different ways, e.g., `standard` and `singleTask` modes.

To address the above challenges and the limitations of the existing static analyses, this paper presents Chime, a launch-mode-aware context-sensitive activity transition analysis for Android apps by precisely resolving the activity transitions together with object-sensitive pointer analysis. The key idea is to use activity transition sequences as the contexts for every activity object to capture the precise transition sequences in a context-sensitive manner, with the launch mode of every activity being considered. Figure 2 illustrates our framework, named Chime, consisting of three components. The activity-aware object-sensitive pointer analysis and launch mode analysis are mutually dependent and resolved until a fixed point is reached.

The preprocessing of Chime extracts metadata, such as launch modes and intent filters of an Android app. Unlike previous approaches [1, 10, 15, 31] that model an activity in a context-insensitive manner by placing the allocations of every activity in the harness main method, Chime generates a slim harness method consisting of only the entry activities, including the ones registered via intent flters or exported to the system. All the other activities are gradually introduced into ATG during the on-the-fly object-sensitive pointer analysis.

Chime performs a context-sensitive activity transition analysis to remove infeasible activity transition paths by distinguishing activity objects using light-weight contexts in the form of activity transition sequences, resulting in a precise context-sensitive ATG. For a target activity resolved at an ICC call site, e.g., *a.startActitivity(intent)* by our pointer analysis, Chime analyzes the corresponding `AndroidManifest.xml` to determine the launch modes of the target activity. For an instance a of activity class A that launches another activity B, a new context-sensitive object b of class B is created and captured in the ATG to model the transition from a to b if B is in `standard` mode. Otherwise, Chime retrieves the existing activity instances $\mathbb{B}$ of B for each transition path reaching a in the context-sensitive ATG if B is configured with a special launch mode, e.g., `singleTask`, then Chime connects a to each instance in $\mathbb{B}$ in its context-sensitive ATG.

This paper makes following contributions:

- We present Chime, a context-sensitive activity transition analysis for Android apps by precisely resolving the activity transitions together with object-sensitive pointer analysis.
- We introduce (for the first time) a launch mode analysis, which is successfully incorporated into the activity transition analysis that distinguishes activity instances context-sensitively.
- We have implemented Chime in the Soot framework [30] in 6K LOC of Java code. We conduct extensive experiments on 42 large real-world Android apps from Google Play. Our results show that Chime is effective in constructing context-sensitive ATGs to remove infeasible transitions and helping GUI testing by comparing Chime against its context-insensitive couterpart.

2 MOTIVATION

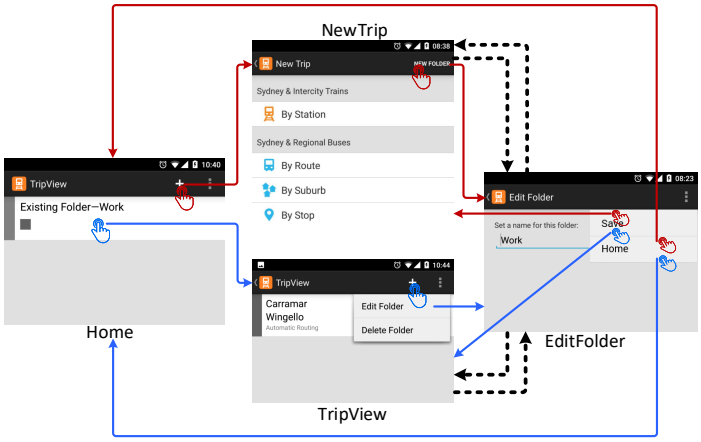
Figure3(a) shows a code snippet (lines1-26) of a real Android app *TripView* (a public transportation trip plan tool) to demonstrate how Chime precisely models the objects of the four activities (i.e., `Home`, `NewTrip`, `TripView`, `EditFolder`) in a context-sensitive manner to identify infeasible activity transition paths. Lines 27–29 in Figure3(a) gives the `AndroidManifest.xml` file that exports only `Home` activity to the system under `singleTask` launch mode. The other three activities are not exported and they are configured under `standard` launch mode.

2.1 Feasible and Infeasible Navigations

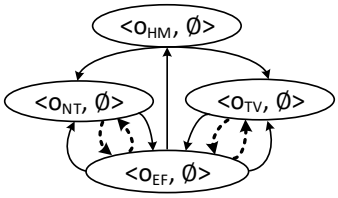
Figure 3(b) demonstrates two activity navigations highlighted in red and blue arrows, respectively. For the first navigation (red arrows), `Home` launches `NewTrip` via the ICC call at line 4 in Figure 3(a) by clicking the "+" on the right corner of `Home`. Then `NewTrip` launches `EditFoler` to create a new trip folder via the ICC call `startActivityForResult()` at line 12 by clicking the "New

```
1  class Home extends Activity {
2   void onClick(...) {
3    Intent newTrip = new Intent(this, NewTrip.class);
4    startActivity(newTrip); }
5   void onItemSelected(...) {
6    Intent tripView = new Intent(this, TripView.class);
7    startActivity(tripView); }
8   void onNewIntent(..., Intent i) {...}
9  class NewTrip extends Activity {
10   void onClick(...) {
11   Intent intent = new Intent(this, EditFolder.class);
12   startActivityForResult(intent, ...); }
13  void onActivityResult(..., Intent i) {...} }
14 class TripView extends Activity {
15  void onItemSelected() {
16   Intent intent = new Intent(this, EditFolder.class);
17   startActivityForResult(intent, ...); }
18  void onActivityResult(..., Intent i) {...} }
19 class EditFolder extends Activity {
20  void onClick(...) {
21   Intent ret = new Intent();
22   ret.putExtra("folder_name", name);
23   setResult(..., ret); }
24  void onClick(...) {
25   Intent home = new Intent(this, Home.class);
26   startActivity(home); } }
27 <manifest>
28 <activity android:name="Home" android:exported="true"
29   android:launchMode="singleTask" /> </manifest>
```

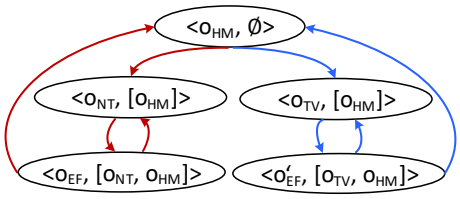
(a) App code snippet from the app TripView.



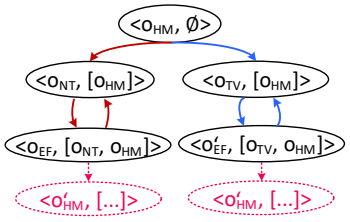
(b) Activity transitions and GUI events of the code.



(c) Context-insensitive ATG.



(d) Launch-mode-aware context-sensitive ATG.



(e) Launch-mode-unaware context-sensitive ATG.

Figure 3: A motivating example. → and → denote two activity transition paths. ⇢ denotes infeasible transition paths introduced by context-insensitive modeling. ⬭ and → denote spurious ATG nodes and edges introduced by launch-mode-unaware analysis. We use o_{HM}, o_{NT}, o_{TV} and o_{EF} to denote the objects of `Home`, `NewTrip`, `TripView` and `EditFolder` respectively.

Folder" button on the right corner of `NewTrip`, which launches the target activity and receives an intent object from the target activity as its result.

`EditFoler` transits back to `NewTrip` once the "Save" is clicked on the menu, so that the `setResult()` API is invoked at line 23, returning an intent object as the results to the previous `NewTrip` instance that launches `EditFoler` through the Android callback `onActivityResult()` at line 13. If "Home" on the menu is clicked, `EditFoler` transits back to `Home` via the ICC call at line 26.

For the second navigation (blue arrows), `Home` launches `TripView` to look up an existing trip by clicking the existing folder via the code at line 7. `TripView` launches `EditFolder` to edit the name of an existing folder by clicking the "Edit Folder" button via the ICC call at line 17. Then `EditFolder` transits back to `TripView` once the "Save" button is clicked, or returns to `Home` if "Home" is clicked.

The above two navigation scenarios are realizable. However, a context-insensitive static transition analysis, which does not distinguish activity instances under different contexts, produces two infeasible transition sequences: `NewTrip` ⇢ `EditFolder` ⇢ `TripView` and `TripView` ⇢ `EditFolder` ⇢ `NewTrip`, as highlighted in dotted arrows in Figure 3(b).

2.2 Context-Sensitive Activity Transition Analysis

Previous ICC analyses [10, 22–24, 31] model the four activities context-insensitively. For each of the four activities, a single abstract object is created and parameterized with an empty context ∅ in the harness method. Figure 3(c) illustrates the context-insensitive ATG which consists of four context-insensitive objects, i.e., $\langle o_{HM}, \emptyset\rangle$, $\langle o_{NT}, \emptyset\rangle$, $\langle o_{TV}, \emptyset\rangle$ and $\langle o_{EF}, \emptyset\rangle$. The context-insensitive activity modeling merges `EditFolder` instances created from different transition paths into one abstract object, causing two infeasible transition paths to be generated, as illustrated.

Chime places only the exported-to-system activity `Home` (which serves the entry activity of the app) in the harness method, so that an object o_{HM} of class `Home` is parameterized with the empty context ∅ in the ATG.

Our activity-aware pointer analysis is performed on the harnessed app to resolve activity-related ICC calls. We query the points-to values of the intent variables at the ICC calls at line 4

and 7 to determine the target activities. Once the target activity classes NewTrip (line 4) and TripView (line 7) are resolved, Chime creates two objects o_{NT} and o_{TV} as the target activity objects. Then, Chime creates the two transition edges in ATG from o_{HM} to o_{NT} and o_{TV}, respectively, where o_{NT} and o_{TV} are parameterized with contexts in the form of the object sequences reaching them in the context-sensitive ATG, i.e.,$[o_{HM}]$ as illustrated in Figure 3(d).

There are two transition paths reaching activity EditFolder, so it is cloned to have two objects $\langle o_{EF}, [o_{NT}, o_{HM}]\rangle$ and $\langle o'_{EF}, [o_{TV}, o_{HM}]\rangle$ parameterized under two different transition sequences as shown in Figure 3(d). Therefore, the infeasible transitions, e.g., NewTrip --➔ EditFolder --➔ TripView are eliminated.

At line 26 in Figure3(a), the instance $\langle o_{EF}, [o_{NT}, o_{HM}]\rangle$ of EditFolder launches activity Home. Since Home is configured with singleTask mode, Chime retrieves the existing instance $\langle o_{HM}, \emptyset\rangle$ along the transition path reaching $\langle o_{EF}, [o_{NT}, o_{HM}]\rangle$, then an edge from $\langle o_{EF}, [o_{NT}, o_{HM}]\rangle$ to $\langle o_{HM}, \emptyset\rangle$ is created in the ATG in Figure 3(d). Similarly, the context-sensitive transition from $\langle o'_{EF}, [o_{TV}, o_{HM}]\rangle$ to $\langle o_{HM}, \emptyset\rangle$ is only created in the ATG. However, a launch-mode-unaware analysis will create two new spurious nodes as illustrated by assuming that Home is always configured with standard mode, as illustrated in Figure 3(e).

3 APPROACH

First, we describe the preliminary knowledge of Android intents and ICC mechanisms (Section 3.1) and the notations used to describe our analysis (Section 3.2). Next, we discuss how Chime preprocesses an Android app to enable subsequent analyses (Section 3.3). Then, we introduce our activity-aware object-sensitive pointer analysis (Section 3.4). Finally, we present context-insensitive (Section 3.5) and context-sensitive activity modeling (Section 3.6).

3.1 Android Intents and ICC Calls

The intent at an ICC call, e.g., startActivity(intent) determines the target ICC component, where intent is either an *explicit intent* specifying the class name of the target component, e.g., line 3 in Figure 1, or an *implicit intent*, which does not indicate a particular component, instead, it implicitly provides the intent's action, category and data fields information, so that the Android framework can look for the target components which register intent filters (in the AndroidManifest.xml) that match the information of the implicit intent.

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

3.2 Notations

Figure 4 gives the notations used in our analysis. The first part lists the analysis domains used. Abstract objects are labeled with their allocation sites. For object-sensitivity, a context is a sequence of abstract objects.

variable	$intent, cpnt, a \in \mathbb{V}$
abstract object	$o_i, o_j, o_k \in \mathbb{O}$
class type	$\mathtt{t} \in \mathbb{T}$
context	$c \in \mathbb{C} = \emptyset \cup \mathbb{O} \cup \mathbb{O}^2 \cup ...$
launch mode	$\{\mathtt{Std}, \mathtt{Top}, \mathtt{Task}, \mathtt{Inst}\} = \mathbb{LM}$
pt :	$\mathbb{V} \times \mathbb{C} \rightarrow \mathcal{P}(\mathbb{O} \times \mathbb{C})$
fpt :	$\mathbb{O} \times \mathbb{C} \times \mathbb{F} \rightarrow \mathcal{P}(\mathbb{O} \times \mathbb{C})$
newActObj :	$\mathbb{T} \rightarrow \mathbb{O}$
actCtxSelector :	$\mathbb{O} \times \mathbb{C} \rightarrow \mathbb{C}$
heapSelector :	$\mathbb{O} \times \mathbb{C} \rightarrow \mathbb{C}$
mtdCtxSelector :	$\mathbb{O} \times \mathbb{C} \rightarrow \mathbb{C}$
ATG	$G = (V, E)$
ATG Node	$\langle o_i, c\rangle \in V \subseteq o_{root} \cup \mathbb{O} \times \mathbb{C}$
ATG Edge	$(\langle o_i, c\rangle \hookrightarrow \langle o_j, c'\rangle, k) \in E \subseteq V \times V \times \mathbb{N}$

Figure 4: Notations.

A few mapping functions are used in our inference rules. Function $pt(p, c)$ obtains the points-to set of a pointer p under context c. Function $fpt(o_i, c, \mathtt{f})$ represents the points-to set of the field f of an abstract object o_i under context c. *newActObj*(t) creates an activity abstract object with type t. *actCtxSelector* generates the activity transition sequences as the heap context of the activity abstract object. Following [19, 20], we use *mtdCtxSelector* and *heapSelector* to generate the method contexts for method calls and heap contexts for non-activity abstract objects, respectively. The bottom part defines ATGs used by Chime. An ATG is a multi-edge graph. A node $\langle o_i, c\rangle$ is a context-sensitive activity abstract object. An edge $(\langle o_i, c\rangle \hookrightarrow \langle o_j, c'\rangle, k)$ between two nodes represents an activity transition resolved by our ICC analysis, where k is a unique ID of a call site that launches an activity, e.g., startActivity().

3.3 Preprocessing

During the preprocessing, an app is decompiled to extract code and metadata, such as the launch mode of each activity and intent filters registered by each activity in the AndroidManifest.xml file. We generate a slim app harness method consisting of only the entry activities, including the ones registered via intent filters or exported to the system. All the other activities are gradually introduced to the ATG by our activity-aware pointer analysis.

In addition to activity, we conservatively handle the other three types of Android components, i.e., services, broadcast receivers, content providers by placing a single object for each of those components in the harness method. Unlike activity component, services and broadcast receivers perform long-running background tasks interacting with the foreground activities. Thus, there are no navigation transitions for them. Content providers supports abstraction for the data maintained by the app and share the data across apps. As there are no transitions between two providers, there is no need to model them in the ATG.

3.4 Activity-Aware Object-Sensitive Pointer Analysis

Once the app preprocessing is finished, CHIME resolves the activity transitions together with object-sensitive pointer analysis. Figure 5 gives the rules for our activity-aware object-sensitive pointer analysis. Follow [27], we use $contexts(m)$ to denote all the contexts reaching the method m from the harness method. In [P-NEW], o_i uniquely identifies the abstract object created as an instance of `t` at the allocation site i. In [P-ASSIGN], 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 [P-CALL] (for non-ICC calls), 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'.

The other four rules in Figure 5 are used for activity-aware object-sensitive pointer analysis. In [I-EX], an explicit intent is resolved by querying the points-to set of the field `mClass` of the object o_j pointed by $cpnt$, whose type is `ComponentName`. In [I-IM], an implicit intent is resolved to find the name of the target components via $actionToCpnt(o_j)$, where $pt(action)$ is the points-to set of $action$ and $actionToCpnt(o_j)$ returns a set of activity names that register an intent filter using the action o_j. The name of the target component of an intent object is stored to its field `Intent.target` introduced by CHIME. Then the points-to facts are propagated to the subsequent ICC call sites.

In [I-ACT], ICC calls that launch activities are intercepted and a fact $(\langle o_i, c'\rangle, \langle o_j, c''\rangle, \mathtt{lm}, p) \rightsquigarrow \mathtt{t}$ is generated, which means that the current activity object o_i under the context c' launches activity class `t` with the launch mode `lm` at the ICC call site p with intent object o_j under the context c''. We use the function $toType(o_k)$ to map the name of a class o_k to its corresponding class type. $toLM(\mathtt{t})$ obtains the launch mode of an activity class of type `t`. Note that we use `startActivity()` to represent all the 11 activity-related ICC APIs, such as `startActivities()` and `startActivityForResult()`.

[I-RET] delivers the results, an intent object, back to the the activity object that launches the current activity. An activity can be launched via the call `startActivityForResult()`. In addition to launching an activity and delivering an intent object, it also requires the target activity instance to return the results by an intent object via the callback `setResult(Intent)`, whose parameter is the result intent. The callback `onActivityResult(Intent)` is invoked on the activity object o_k by the Android system to receive the new intent object o_j. In [I-RET], the activity transitions from the launcher activity object o_k to current one o_i are captured by the ATG, so that the result intent object o_j can be delivered to o_k. We use a method `modelSetResult()` to represent the fact that it invokes callback `onActivityResult()` on the launcher activity object o_k to deliver the intent object o_j.

3.5 Context-Insensitive Activity Transition Analysis

Figure 6 gives the rule [ACT-CI] that describes the context-insensitive activity transition modeling, where a newly introduced

$$\frac{i : x = \mathtt{new\ t()}\ \ c \in contexts(m)}{\langle o_i, heapSelector(o_i, c)\rangle \in pt(x, c)}\ [\text{P-NEW}]$$

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

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

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

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

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

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

$$\frac{\begin{array}{c} p :\ a.startActivity(intent)\ \ c \in contexts(m) \\ \langle o_i, c'\rangle \in pt(a, c)\ \ \langle o_j, c''\rangle \in pt(intent, c) \\ \langle o_k, \_\rangle \in fpt(o_j, c'', \mathtt{target}) \end{array}}{\begin{array}{c} \mathtt{t} = toType(o_k)\ \ \mathtt{lm} = toLM(\mathtt{t}) \\ (\langle o_i, c'\rangle, \langle o_j, c''\rangle, \mathtt{lm}, p) \rightsquigarrow \mathtt{t} \end{array}}\ [\text{I-ACT}]$$

$$\frac{\begin{array}{c} a.setResult(intent)\ \ c \in contexts(m) \\ \langle o_i, c'\rangle \in pt(a, c)\ \ \langle o_j, x\rangle \in pt(intent, c) \\ (\langle o_k, c''\rangle \hookrightarrow \langle o_i, c'\rangle, \_) \in E\ \ o_k \neq o_{root} \end{array}}{\mathtt{modelSetResult}(\langle o_k, c''\rangle, \langle o_j, x\rangle)}\ [\text{I-RET}]$$

Figure 5: Rules for activity-aware object-sensitive pointer analysis.

$$\frac{(\langle o_i, \emptyset\rangle, \langle o_p, x\rangle, k, j) \rightsquigarrow \mathtt{t}}{\begin{array}{c} o_k = newActObj(\mathtt{t})\ \ hc = \emptyset\ \ \mathtt{modelActivity}(\langle o_k, hc\rangle) \\ \langle o_p, x\rangle \in fpt(o_k, hc, \mathtt{mIntent}) \\ \mathtt{modelNewIntent}(\langle o_k, hc\rangle, \langle o_p, x\rangle) \\ \langle o_k, hc\rangle \in V\ \ (o_i \hookrightarrow \langle o_k, hc\rangle, j) \in E \end{array}}\ [\text{ACT-CI}]$$

Figure 6: A rule for context-insensitive activity transition analysis.

target activity abstract object is created with $\emptyset$ as its context. We use the method $\mathtt{modelActivity}(\langle o_k, \emptyset\rangle)$ to model the life-cycle callbacks (e.g., `onCreate()`) and event handling of the target activity object o_k (e.g., `onClick()`). Moreover, the intent object o_p is added to the points-to set of the field `mIntent` of target activity object o_k. The field `mIntent` can be accessed via invoking the Android API `getIntent()` on the object o_k. We use the method

`modelNewIntent()`, which contains the callback `onNewIntent()` to deliver the intent object o_p to target activity object o_k, to model the relaunch of o_k. Finally, an edge from the current activity object o_i to the target one o_k is added to ATG.

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

In this section, we discuss ATG construction and launch mode analysis.

ATG is constructed on-the-fly with pointer and ICC analysis. It captures the activity transition relations of the whole app. A transition edge is inserted once a new transition is found.

The rules in Figure 7 establish some basic relations for an ATG. In [Atg-Root], a dummy root node, o_{root}, is added to the ATG to serve as its entry. In [Atg-Entry], any entry activity object, whose heap context is $\emptyset$, is added as a node to the ATG, and the node is connected with o_{root}. [Atg-Refl] and [Atg-Trans] introduce reflexive and transitive relations of transition edges. Note that the reflexive and transitive edges are used in the later launch mode analysis, but are not actually added to the ATG.

$$\frac{}{o_{root} \in V}\ [\text{Atg-Root}] \qquad \frac{\langle o_i, c\rangle \in V}{\langle o_i, c\rangle \sim \langle o_i, c\rangle}\ [\text{Atg-Refl}]$$

$$\frac{(\langle o_i, \emptyset\rangle, \_, \_, j) \rightsquigarrow \_}{\langle o_i, \emptyset\rangle \in V \ \ (o_{root} \hookrightarrow \langle o_i, \emptyset\rangle, j) \in E}\ [\text{Atg-Entry}]$$

$$\frac{(\langle o_i, x\rangle \hookrightarrow \langle o_j, y\rangle, \_) \in E \ \ (\langle o_j, y\rangle \hookrightarrow \langle o_k, z\rangle, \_) \in E}{\langle o_i, x\rangle \Rightarrow \langle o_k, z\rangle}\ [\text{Atg-Trans}]$$

Figure 7: Rules for activity transition graph (ATG).

The execution of an app is managed by one or more tasks. Every task has a *back stack*, which is used to manage activity navigation (adding the newly created activity to the stack) and backtrack (popping off the finished activity off the stack). A launch mode specifies how a new instance of an activity is associated with the current task [9]. Developers specify launch modes to keep smooth and consistent user experiences or to achieve the design requirements such as singleton patterns. For example, the launch mode of `Home` is specified as `singleTask` at line 29 in Figure 3(a), which means that only one instance of `Home` exists at any time in the back stack.

With the help of a context-sensitive ATG, launch modes can be effectively handled. The ATG keeps track of activity transitions for every transition path, which mimics the back stack of the Android activity manager. There are four types of launch mode: `standard`, `singleTop`, `singleTask` and `singleInstance`. Figure 8 gives the rules to analyze four launch modes of activities, i.e., `standard` ([L-Std]), `singleTop` ([L-TNew] and [L-TPrev]), `singleTask` ([L-TKNew] and [L-TKPrev]), and `singleInstance` ([L-Inst]).

Standard. This is the default mode. The system always creates a new activity instance and then delivers the intent to the target activity object. A target activity abstract object is created and parameterized with the context generated by *actCtxSelector*(). We use the heap context of an activity abstract object to represent the reachable transition sequences. $\langle o_i, c\rangle$ represents the current

$$\frac{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Std}, j) \rightsquigarrow \texttt{t}}{\begin{array}{c} o_j = newActObj(\texttt{t}) \ \ hc = actCtxSelector(o_i, c) \\ \langle o_j, hc\rangle \in V \ \ \texttt{modelActivity}(\langle o_j, hc\rangle) \\ \langle o_p, x\rangle \in fpt(o_j, hc, \texttt{mIntent}) \\ (\langle o_i, c\rangle \hookrightarrow \langle o_j, hc\rangle, j) \in E \end{array}}\ [\text{L-Std}]$$

$$\frac{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Top}, j) \rightsquigarrow \texttt{t} \ \ getType(o_i) \neq \texttt{t}}{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{std}, j) \rightsquigarrow \texttt{t}}\ [\text{L-TNew}]$$

$$\frac{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Top}, j) \rightsquigarrow \texttt{t} \ \ getType(o_i) == \texttt{t}}{\begin{array}{c} (\langle o_i, c\rangle \hookrightarrow \langle o_i, c\rangle, j) \in E \\ \texttt{modelNewIntent}(\langle o_i, c\rangle, \langle o_p, x\rangle) \end{array}}\ [\text{L-TPrev}]$$

$$\frac{\begin{array}{c} (\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Task}, j) \rightsquigarrow \texttt{t} \\ \neg(\langle o_k, c'\rangle \hookrightarrow \langle o_i, c\rangle \wedge getType(o_k) == \texttt{t}) \end{array}}{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Std}, j) \rightsquigarrow \texttt{t}}\ [\text{L-TkNew}]$$

$$\frac{\begin{array}{c} (\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Task}, j) \rightsquigarrow \texttt{t} \\ \langle o_k, c'\rangle \hookrightarrow \langle o_i, c\rangle \ \ getType(o_k) == \texttt{t} \end{array}}{\begin{array}{c} (\langle o_i, c\rangle \hookrightarrow \langle o_k, c'\rangle, j) \in E \\ \texttt{modelNewIntent}(\langle o_k, c'\rangle, \langle o_p, x\rangle) \end{array}}\ [\text{L-TkPrev}]$$

$$\frac{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Inst}, j) \rightsquigarrow \texttt{t}}{(\langle o_i, c\rangle, \langle o_p, x\rangle, \texttt{Task}, j) \rightsquigarrow \texttt{t}}\ [\text{L-Inst}]$$

Figure 8: Rules for ATG based launch mode analysis.

activity object with a transition sequence $c = [o_{i-1}, ..., o_1]$ for full-object-sensitivity. $actCtxSelector(\langle o_i, c\rangle) = [o_i, ..., o_1]$ denotes concatenation operation that adds o_i to c.

For k-object-sensitive analysis with (k–1)-context-sensitive heap [12, 26], we keep the most recent $(k-1)$ elements in a transition sequence. Given $c = [o_{i-1}, ..., o_{i-k+1}]$, $actCtxSelector(\langle o_i, c\rangle) = [o_i, ..., o_{i-k+2}]$ adds o_i to the top of the context stack by removing tail element o_{i-k+1} to keep the size of the stack unchanged.

SingleTop. This mode reuses the activity instance on the top of the back stack if its type is the same as that of the target activity. In [L-TNew], we match the type of the current activity object with that of the target activity. If they are the same, delivering the new intent object is modeled by method `modelNewIntent()`. Otherwise, we resort to [L-Std] for handling it as the `standard` mode following [L-TPrev].

SingleTask. This mode is similar to `singleTop`, except that the activity instance is reused if there is an activity instance in the back stack whose type is the same as that of the target activity. Otherwise, it is resolved as in the `standard` mode. In [L-TkNew], we inspect whether there is a reachable activity with the same type as that of the target one via the transitive or reflexive edge established by [Atg-Trans] and [Atg-Refl]. If so, delivering the new intent object is modeled by method `modelNewIntent()`. Otherwise, it is handled as in the `standard` mode.

SingleInstance. This mode is similar to `singleTask`, except that the task holding the new activity object is always the only member of its containing task. This mode is represented in the same way as the `singleTask` mode in the ATG, so that we handle it as the `singleTask` mode following [L-Inst].

To the best of our knowledge, CHIME is the first static analysis that uses activity transition sequences as contexts for handling launch modes to obtain precise activity transitions.

3.7 Examples

Let us apply CHIME to the example in Figure 3 to resolve ICC calls and construct ATG. We use $\langle o_{HM}, \emptyset \rangle$ to represent the `Home` object that is allocated in the harness method. Suppose that the method context of the callback methods `onClick()` at line 2 and `onItemSelected()` at line 5 are c_1 and c_2, respectively, and intent objects allocated at lines 3 and 6 are o_3 an o_6. These intent objects can be resolved by applying [I-Ex], so that we obtain "`NewTrip`" $\in fpt(o_3, c_1, \texttt{target})$ and "`TripView`" $\in fpt(o_6, c_2, \texttt{target})$. By applying [I-Act] to lines 4 and 7, the ICC call relations $(\langle o_{HM}, \emptyset \rangle, \langle o_3, c_1 \rangle, \texttt{Std}, 4) \rightsquigarrow$ `NewTrip` and $(\langle o_{HM}, \emptyset \rangle, \langle o_6, c_2 \rangle, \texttt{Std}, 7) \rightsquigarrow$ `TripView` are established.

Then we can apply [L-Std] to create the target activity objects of the two ICC calls context-sensitively. Function *actCtxSelector* generates $[o_{HM}]$ for the both target activity objects o_{NT} and o_{TV} as their heap contexts. Two edges $(\langle o_{HM}, \emptyset \rangle \hookrightarrow \langle o_{NT}, [o_{HM}] \rangle, 4)$ and $(\langle o_{HM}, \emptyset \rangle \hookrightarrow \langle o_{TV}, [o_{HM}] \rangle, 7)$ are added to the ATG.

Similarly, we can resolve the ICC calls at lines 12 and 17. Function *actCtxSelector* generates heap context $[o_{NT}, o_{HM}]$ for the target activity object o_{EF} of the ICC call at line 12. For line 17, the heap context $[o_{TV}, o_{HM}]$ for the target activity object o'_{EF} is generated. Two edges $(\langle o_{NT}, [o_{HM}] \rangle \hookrightarrow \langle o_{EF}, [o_{NT}, o_{HM}] \rangle, 12)$ and $(\langle o_{TV}, [o_{HM}] \rangle \hookrightarrow \langle o'_{EF}, [o_{TV}, o_{HM}] \rangle, 17)$ are added to the ATG. Therefore, the two objects of the activity `EditFolder` allocated in two transition paths are distinguished by the context-sensitive activity modeling. The two infeasible paths in Figure 3(c) are eliminated.

Finally, we can resolve the ICC call at line 26, whose target activity is `Home` and configured with `singleTask` mode (line 29 in Figure 3(a)). The transitive relation $\langle o_{HM}, \emptyset \rangle \Rightarrow \langle o_{EF}, [o_{NT}, o_{HM}] \rangle$ generated by [Atg-Trans] indicates that there is an object of `Home` reachable to the current activity object o_{EF} via the transition paths maintained by the ATG. Therefore, [L-TkPrev] is applied to resolve the ICC call at line 26 by adding an edge $(\langle o_{EF}, [o_{NT}, o_{HM}] \rangle \hookrightarrow \langle o_{HM}, \emptyset \rangle, 26)$ to the ATG. Similarly, $(\langle o'_{EF}, [o_{TV}, o_{HM}] \rangle \hookrightarrow \langle o_{HM}, \emptyset \rangle, 26)$ is also added to the ATG. Therefore, the `singleTask` mode for the activity `Home` is effectively resolved without introducing spurious ATG nodes and edges.

4 EVALUATION

The aim of our evaluation is to demonstrate that CHIME is effective in constructing context-sensitive ATGs to remove infeasible transitions and facilitating GUI testing by comparing CHIME against its context-insensitive counterpart.

4.1 Implementation

CHIME is built on top of the SPARK pointer analysis framework in SOOT [13, 30], in about 6 KLOC of Java code. We leverage the Android Device Implementation (ADI) provided by DROIDSAFE [10], a comprehensive modeling of the Android framework and Java library, to improve the precision and soundness of our analysis. DROIDSAFE is used to decompile apps, generate harness methods and obtain app matadata from manifest files. We have resolved reflection using the approach presented in [16] to obtain a more sound call graph.

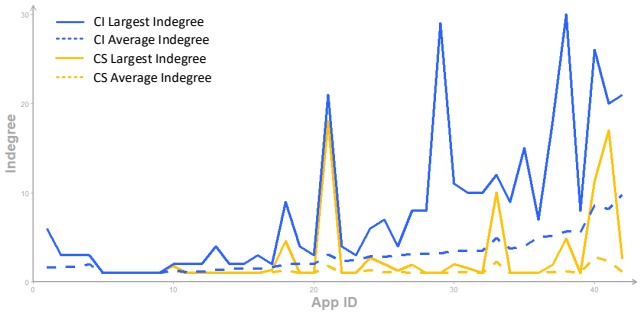


Figure 9: Comparison between CICHIME (blue line) and CHIME (yellow line). The largest (solid line) and average (dotted line) numbers of the incoming activity classes of the nodes on an ATG. The x-axis gives the 42 apps, and the y-axis represents the number of incoming activity classes.

4.2 Experiment Setup and Methodology

We select 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 (i.e., two elements for a method context and one element for a heap context), which is a widely used configuration for achieving the best tradeoff between precision and scalability [12, 26, 28, 29].

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 Android app is the average of three runs.

TRIMDROID [21], BRAHMASTRA [4] and A^{3}E [2] use static analysis to generate an app's activity transition graph to guide the GUI testing tasks. These approaches perform context-insensitive activity transition analysis that generates a context-insensitive ATG. In order to evaluate the effectiveness of context-sensitive activity transition analysis in helping GUI testing, we compare CHIME with CICHIME, which is a simplified version of CHIME, according to [Atg-Ai] in Section 3.5 by answering the following research questions:

- **RQ1.** Is CHIME able to distinguish an activity in different transition paths compared against CICHIME?
- **RQ2.** Is CHIME capable of analyzing launch modes effectively?
- **RQ3.** is CHIME useful for facilitating GUI testing?

4.2.1 ***RQ1: Context-Sensitive ATG Analysis.*** As shown in Table 1, we compare CHIME with CICHIME in terms of the number of incoming activity classes of the nodes in the ATGs generated by context-sensitive and context-insensitive activity transition analyses for each of the 42 apps evaluated (smaller is better). The solid blue and yellow lines show the largest incoming activity classes, and the dotted blue and yellow lines show the average incoming activity classes of all the activity nodes in an ATG.

CHIME is more precise than CICHIME as shown in Figure 9 by comparing the ATGs generated by CHIME and CICHIME. CHIME is able to analyze an activity under different contexts, so that an activity launched can be distinguished and analyzed separately.

Table 1: The statistics of the ATGs generated by CiChime and Chime. For each app, Columns "Max Ind" and "Avg Ind" represent the maximal and average number of incoming activity classes of nodes in an ATG. Column "PTA Time" gives the pointer analysis time in seconds.

App Name	CiChime					Chime					App Name	CiChime					Chime				
	ATG Edge	ATG Node	Max Ind	Avg Ind	PTA Time	ATG Edge	ATG Node	Max Ind	Avg Ind	PTA Time		ATG Edge	ATG Node	Max Ind	Avg Ind	PTA Time	ATG Edge	ATG Node	Max Ind	Avg Ind	PTA Time
Instagram	171	12	10	3.8	238	6440	288	2.0	1.2	844	Opal	38	10	9	2.8	26	113	46	1.0	1.0	50
GoogleLite	12	4	2	1.7	16	39	11	1.0	1.0	25	RollingSky	242	9	9	3.8	288	998	37	1.0	1.0	439
Raider	7	6	2	1.2	19	21	17	1.0	1.0	29	StarWars	9	4	2	1.7	24	11	8	1.5	1.2	35
Speed	19	11	6	1.6	34	20	13	6.0	1.6	54	ClashofClan	5	2	1	1.0	15	5	2	1.0	1.0	21
FlashLight	11	5	4	3.0	68	25	13	1.5	1.3	108	TripView	14	12	2	1.1	11	41	28	1.0	1.0	19
Booster	5	6	3	2.0	30	5	6	3.0	2.0	45	FireFox	57	11	7	2.8	71	272	124	2.0	1.1	190
Seek	8	4	4	2.0	19	15	11	1.0	1.0	29	Cleaner	19	20	4	1.3	44	110	77	1.0	1.0	119
ANZ	187	7	7	5.0	23	1182	50	1.0	1.0	41	TalkingAngela	10	3	1	1.0	38	10	5	1.0	1.0	62
Antivirus	244	26	20	8.2	508	5101	369	17.0	2.3	1800	SpeedTest	3	3	2	1.5	83	6	6	1.0	1.0	136
Solitaire	6	2	1	1.0	66	126	26	1.0	1.0	148	TalkingTom	10	3	1	1.0	42	10	5	1.0	1.0	67
Clean	7	8	3	1.7	34	7	9	3.0	1.7	53	Slots	41	4	2	1.3	12	103	7	1.0	1.0	17
PhoneClean	29	16	9	2.0	44	61	28	4.2	1.3	80	HealthEngine	149	8	8	5.6	73	7713	113	1.0	1.0	210
Pool	19	2	2	1.5	18	28	3	1.0	1.0	26	Tiles	675	18	18	5.4	824	23589	391	2.0	1.1	2901
PayPal	221	15	15	4.0	19	1943	90	1.0	1.0	77	HillClimb	119	4	4	2.3	84	236	8	1.0	1.0	126
ClashRoyale	5	2	1	1.0	15	5	2	1.0	1.0	21	HungryJacks	34	11	10	3.4	45	152	52	1.6	1.1	80
MobileStrike	13	5	3	2.0	18	45	15	1.0	1.0	27	MachineZone	13	5	3	2.0	17	45	15	1.0	1.0	27
Cleaner	6	7	3	1.7	30	6	7	3.0	1.7	48	DeathWorm	5	3	3	2.5	31	9	6	1.0	1.0	47
Domino	11	3	2	1.3	19	47	16	1.8	1.3	31	Wallet	186	18	12	4.9	166	11260	426	10.1	2.3	867
Mail	710	21	21	9.8	2960	5216	364	1	1.0	6252	Legends	9	4	3	1.5	12	165	41	1.0	1.0	28
VLC	71	11	6	2.9	17	5357	168	2.8	1.4	105	Catch	232	40	21	3.1	1312	953	216	18.0	1.8	2398
PowerClean	45	21	8	3.1	81	77	80	1.0	1.0	301	K9	78	20	8	3.2	231	1775	277	2.0	1.0	1868

Since Chime analyzes each activity under different contexts, the analysis time is longer than CiChime, with the longest analysis under 2 hours for the large real-world Android app, `Mail`.

Cast study of imprecision in object-sensitive pointer analysis. Object-sensitive pointer analysis is not precise for analyzing static calls. The code snippet in Figure 10 taken from the app `Cleaner` illustrates a real scenario where object-sensitive pointer analysis may produce spurious activity transitions.

A static method `createIntent()` at line 7 creates an intent with the source activity object and target activity type at line 8, and it is invoked twice in activity b at lines 3 and 5, respectively, to create different intent objects and launch the corresponding activities.

The explicit intent in this example is resolved by querying the points-to set of `Intent.mComponent.mClass` (Section 3.4). The 2obj+h pointer analysis merges the intent objects created under two contexts, causing the intent analysis to resolve imprecisely the two activities `CpuUsageListActivity` and `JunkCleanActivity` at the two ICC calls at lines 4 and 6. The precision of our transition analysis can be improved if a more precise pointer analysis is used, such as hybrid context-sensitive analysis [12], which applies call-site-sensitivity to distinguish static calls at lines 3 and 5, so that the intent objects allocated at line 8 under two contexts can be distinguished for launching different activities context-sensitively.

4.2.2 ***RQ2: Launch Mode Analysis.*** Launch modes can significantly affect activity transitions. We firstly examine the prevalence of activity under special launch modes (other than the `standard` one) in real-world Android apps. We then present and discuss the results of our launch mode analysis. Finally, we use a real-world example to demonstrate the effectiveness of our activity transition analysis in handling special launch modes.

```
1 class b extends Activity {
2   void onClick(...) {
3   Intent intent = createIntent(this, CpuUsageListActivity.class);
4   startActivity(intent);
    ...
5   Intent intent = createIntent(this, JunkCleanActivity.class);
6   startActivity(intent); }

7 static Intent createIntent(Context src, Class tgt) {
8  Intent intent = new Intent(src, tgt);
9  return intent; } }
```

Figure 10: Code snippet from the app Cleaner. The ICC calls at lines 4 and 6 are imprecisely resolved by Chime, since the 2obj+h object-sensitive pointer analysis cannot distinguish the static calls at lines 3 and 5.

We collect the number of activities that are reachable by our analysis under special and standard launch modes for all the 42 apps, as shown in Figure 11. We can see that 30% of all the activities are configured with special launch modes. Developers prefer to use special launch modes for the activities in the real-world Android apps to keep smooth and consistent user experiences. The special launch modes are also used to achieve the design requirements such as singleton patterns. Therefore, it is important to develop launch-mode-aware analysis to understand the activity transitions in the Android apps.

Launch Mode Analysis Results. The key difference between the behaviors of launching an activity under the standard mode from a special mode is that the former always creates a new activity instance, whereas the latter may resume an existing activity that has already been created. In an ATG generated by Chime, we use two types of ATG edges, i.e., `NewActEdge` and `PrevActEdge` to

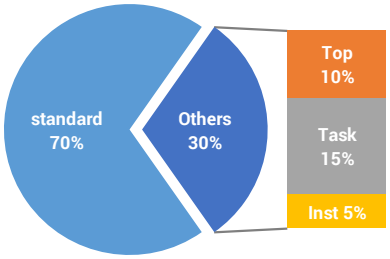


Figure 11: The percentage of activities with default and special launch modes in all the reachable activities of 42 apps.

distinguish these two types of transition cases, i.e., launching a target under the standard mode vs. under a special mode.

We measure the precision of Chime in terms of the number of PrevActEdges in an ATG generated by Chime for launching activities under the three special launch modes. Table 2 gives the number and percentage of activities under standard and special launch modes. We also give the number and percentage of NewActEdges and PrevActEdges in the ATGs of all the 42 apps.

Table 2: Statistics of our launch mode analysis.

	Launch Mode		ATG Edge	
	Standard	Special	NewActEdge	PrevActEdge
Number	358	157	59329	14067
Percentage	69.5%	30.5%	80.8%	19.2%

Table 2 shows that Chime's launch mode analysis is effective. In the ATGs of 42 apps, 30.5% of activities under special launch modes generate 19.2% PrevActEdges while 69.5% of the activities under standard mode generate 80.8% NewActEdges.

Table 3 gives a breakdown of the statistics in Table 2 for individual apps. Note that an activity configured with a special launch mode may not have an incoming edge of type PrevActEdge if it is only launched once, i.e., no more startActivity() exists for launching this existing activity again. For example, there is no PrevActEdge on the ATG of **Seek** (**in bold** in Table 3).

Case Study. We use a code snippet from real-world Android app Catch in Figure 12 to illustrate our launch mode analysis. MainHome (MH) activity, whose launch mode is singleTask, launches SearchActivity (SA) at ①, and later this activity launches ShoppingCartView (SC) at ②. Next, ShoppingCartView launches MainHome again at ③. Chime generates a launch-mode-aware ATG to capture the activity transitions, so that the special launch mode singleTask is resolved at ③ by [L-TkPrev] (Section 3.6). Finally, ③ resumes the existing activity instance MainHome. Thus, the activity transitions MH → SA → SC is precisely analyzed by Chime.

4.3 RQ3: Helping App GUI Testing

We show that the context-sensitive ATGs generated by Chime are useful for improving the efficiency of ATG-guided GUI testing analysis [2, 4, 21]. For example, Brahmastra [4] finds activity transition paths from the entry activities to a target activity by performing

Table 3: Breakdown data in Table 2 for each app. Column std and spl (special) give the number of activities with standard launch mode and other special launch modes, respectively.

App Name	Launch Mode		ATG Edge		App Name	Launch Mode		ATG Edge	
	Std	Spl	New	Prev		Std	Spl	New	Prev
Instagram	9	3	5446	994	Opal	9	1	113	0
GoogleLite	4	0	39	0	RollingSky	8	1	998	0
Raider	4	2	20	1	StarWars	3	1	9	2
Speed	4	7	10	10	ClashofClan	1	1	2	3
FlashLight	3	2	23	2	TripView	12	0	41	0
Booster	2	4	2	3	FireFox	7	4	255	17
Seek	1	3	15	0	Cleaner	5	22	110	0
ANZ	7	0	1182	0	TalkingAngela	1	2	3	7
Antivirus	17	9	4632	469	SpeedTest	2	1	6	0
Solitaire	2	0	126	0	TalkingTom	1	2	3	7
Clean	2	6	4	3	Slots	2	2	102	1
PhoneClean	7	9	31	30	HealthEngine	8	0	7713	0
Pool	1	1	27	1	Tiles	15	3	20805	2784
PayPal	13	2	1883	60	HillClimb	3	1	236	0
ClashRoyale	1	1	2	3	HungryJacks	10	1	131	21
MobileStrike	5	0	45	0	MachineZone	5	0	45	0
Cleaner	2	5	3	3	DeathWorm	2	1	9	0
Domino	1	2	37	10	Wallet	12	6	2623	8637
Mail	17	4	5160	110	Legends	3	1	164	1
VLC	7	4	4957	400	Catch	26	14	497	456
PowerClean	13	8	76	1	K9	18	2	1744	31

```
// launch mode singleTask
class MainHome extends Activity {
void onNewIntent(Intent intent) {
 setIntent(intent);
 ...}
 boolean onQueryTextSubmit(String query) {
  Intent intent = new Intent(this, SearchActivity.class);
① intent.putExtra("search_query", query);
 startActivity(intent); } }

 // launch mode standard
class SearchActivity extends Activity {
 void onClick(View view) {
② Intent intent = new Intent(this,ShoppingCartView.class);
 startActivity(intent); } }

 // launch mode singleTask
class ShoppingCartView extends Activity {
③ boolean onOptionsItemSelected(MenuItem menuItem) {
 startActivity(new Intent(this, MainHome.class)); }}
```

PrevActEdge NewActEdge NewAvtEdge

Figure 12: Activity transition under the special launch mode, singleTask, in the app Catch.

instrumentations based on the ATG of an app to dynamically check GUI bugs at runtime. In the context-insensitive ATGs used by Brahmastra, activity objects under different contexts are merged, giving rise to infeasible transition paths with redundant instrumentation. The ATGs generated by Chime enable Brahmastra to follow activity transitions context-sensitively, thereby avoiding infeasible transitions, so that the testing can be performed more efficiently.

We collect the number of reachable activities from the entry ones in the harness methods to the target activities for the 42

apps as listed in Table 4 by using context-sensitive and context-insensitive ATGs. Columns 2 and 5 give the number of reachable activity objects from the entry activity objects to the target activity objects under all the contexts in the context-insensitive ATG for every app. Columns 3 and 6 give the average number of reachable activity objects from the entry activity objects to the target objects under every context-sensitive transition path.

For the most activities in Table 4, CHIME's ATGs are strictly more precise than the context-insensitive ones, since CHIME has successfully removed some spurious activities along some context-sensitive transition paths. Let us take a look at the two cases marked **in bold** in Table 4 to examine the effectiveness and the limitations of CHIME. For activity `SignInActivity`, GUI testing from the entry activities to `SignInActivity` needs to traverse 9 activity objects under every context in the context-insensitive ATG, while CHIME can distinguish the activity objects under different contexts, which allow testing tools to traverse only 4.67 activities on average per context. For a few activities, such as `PostTwitterActivity`, in Table 4, CHIME achieves the same precision as its context-insensitive counterpart. An average of 11 activities are reachable from the entry activities in both ATGs. This arises due to the nature of k−object-sensitive pointer analysis, which merges the activity objects under different contexts that cannot be distinguished by k-object-sensitivity, resulting in spurious transition relations.

Table 4: Effectiveness of CHIME in accelerating ATG-based GUI testing. Columns CI and CS show the number of reachable activities from the entry activities in the harness main to a specific activity as listed under context-insensitive and context-sensitive ATGs, respectively.

Activity Name	CI	CS	Activity Name	CI	CS
SignInActivity	9	4.67	**PostTwitterActivity**	11	11
WebFlowActivity	4	3	CBImpressionActivity	9	3.07
AddAccountActivity	3	2	AdActivity	2	1.5
FlashlightActivity	2	1.5	FxAccountStatusActivity	7	3.11
TJAdUnitActivity	4	2.25	RandomFbAdsActivity	6	3.75
BoostFinishActivity	3	2	CpuCoolResultActivity	8	8
BoostShortCutActivity	9	3.75	AddUserWhiteListActivity	7	7
GameBoostShortCutActivity	8	5.5	RubbishScanningActivity	6	3.75
AdyenResponse	2	1.8	AdActivity	2	1.67
HomeScreen	2	1.75	PrizeActivity	4	3.5
NetworkFailureActivity	3	2.29	AppLockSettingsActivity	4	4
InterstitialAdActivity	9	2.58	EditFolderActivity	3	2.2
ApkManagerActivity	2	2	EditTripActivity	2	2
CleanResultActivity	4	2.25	PrizeExpiredActivity	5	5
DeviceRootedActivity	10	4.83	AdColonyAdViewActivity	3	2.67

5 RELATED WORK

We limit our discussion to the most relevant work.

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 every component. Then a context-insensitive pointer analysis is performed to construct the call graph and inter-procedural control-flow graph, against which some intra-component taint analysis is performed. Li et al. [15] propose IccTA, a static taint analysis built on top of FLOWDROID, to detect inter-component privacy leaks in Android apps. It queries the ICC analysis results from IC3 [23] to instrument the targets at each ICC call site by using context- and flow-insensitive pointer analysis.

Gordon et al. [10] present DROIDSAFE, a static information leakage detection analysis for Android apps by applying object-sensitive pointer analysis to resolve intents at ICC calls. Wei et al. [31] introduce AMANDROID, a static inter-component data-flow analysis framework for various security vetting tasks. It uses call-site sensitive and flow-sensitive pointer analysis to construct the call graph for an app. Despite the fact that both techniques use context-sensitive pointer analysis to construct call graphs, the activities are still modeled in a context-insensitive manner.

CHIME distinguishes the target activities using their transition sequences using context-sensitive ATGs, which are constructed on-the-fly during object-sensitive pointer analysis with the launch modes of all activities being considered. The context-sensitive ATGs generated by CHIME can help improve the efficiency of various ATG-based GUI testing tools [2, 4, 21].

Pointer Analysis. Object-sensitive analysis proposed by Milanova et al. [19, 20] is known to be the best context-sensitive pointer analysis for object-oriented programming languages [12, 14, 26, 28, 29]. Type sensitivity and hybrid context-sensitivity [12, 26] are two variants based on object-sensitivity. The former approximates the object allocation sites in a calling context by the dynamic types of their allocated objects, which trade precision for better scalability and efficiency. The latter applies call-site-sensitivity to static calls and object-sensitivity to virtual calls. Recently, Tan et al. [28] improves the precision of k−object-sensitivity analysis by identifying redundant contexts using Object Allocation Graphs (OAGs), which are built based on a context-insensitive Andersen's analysis. However, all the existing context-sensitive pointer analyses are unable to reason about the Android-specific APIs, such as ICC calls, which contain a large number of callbacks for interacting with the Android framework (e.g., activity object allocations). CHIME is designed to be aware of the activity-related APIs by parameterizing every activity object with contexts in the form of transition sequences to produce a context-sensitive activity transition graph, which serves as foundation for various Android clients, e.g., information leakage detection and GUI testing.

6 CONCLUSION

This paper presents CHIME, a launch-mode-aware context-sensitive activity transition analysis for Android apps. CHIME precisely resolves the activity transitions together with object-sensitive pointer analysis by using the transition sequences in ATGs as the contexts to distinguish activity objects with the launch modes of all activities being considered. Our evaluation on 42 large real-world Android apps from Google Play shows that our context-sensitive activity transition analysis is more precise than its context-insensitive counterpart in terms of removing infeasible transitions and facilitating GUI testing. The experimental data can be downloaded from https://goo.gl/3G44FT.

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 *USENIX Security '14*. 1021–1036.

[5] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security '14*. 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. 2016. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. *arXiv preprint arXiv:1608.06254* (2016).

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

[10] 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 *NDSS '15*.

[11] 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

[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] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using S park. In *Proceedings of the 12th International Conference on Compiler Construction (CC '03)*. Springer, 153–169.

[14] 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.

[15] 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

[16] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *European Conference on Object-Oriented Programming*. Springer, 27–53.

[17] 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

[18] 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

[19] 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

[20] 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

[21] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE '16)*. 559–570. https://doi.org/10.1145/2884781.2884853

[22] 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

[23] 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

[24] 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 *USENIX Security '13*. 543–558.

[25] 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

[26] 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

[27] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 196–232.

[28] 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.

[29] 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

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[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] S. Yang, D. Yan, H. Wu, Y. Wang, and A. 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