Effectively Finding ICC-related Bugs in Android Apps via Reinforcement Learning

Hui Guo
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
guohui@mail.dhu.edu.cn

Ting Su*
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
tsu@sei.ecnu.edu.cn

Xiaoqiang Liu*
School of Computer Science and Technology, Donghua University
Shanghai, China
liuxq@dhu.edu.cn

Siyi Gu
School of Computer Science and Technology, Donghua University
Shanghai, China
2222710@mail.dhu.edu.cn

Jingling Sun
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
jingling.sun910@gmail.com

***Abstract*—Inter-component communication (ICC) is a key mechanism in Android. It utilizes *intents* to achieve the communications between different components in the apps. Thus, the successful execution of ICCs (named *ICC calls*) is fundamental to the app operations. However, existing testing tools for Android seldom explicitly consider these ICC calls, which may fail to find those ICC-related bugs. To this end, we propose a novel ICC-guided exploration strategy to effectively find the ICC-related bugs. Our idea is that, we can (1) build an ICC call graph from the app under test, and (2) use this graph to guide the exploration toward exercising the ICC calls. To achieve this idea, we design this ICC-guided exploration strategy based on Q-learning, a classic reinforcement learning algorithm. Specifically, the reward function explicitly considers the number of explored intents, the number of promising-to-explore intents and the exploration order of explored intents to improve testing effectiveness. Moreover, to build a more complete ICC call graph, we design a graph enhancement exploration strategy also based on Q-learning to complement the call graph construction via static analysis. We have implemented our idea as an automated testing tool ICCDROID. The evaluation on 28 real-word Android apps shows that ICCDROID can effectively find the most number of ICC-related bugs within the same testing time, compared to existing testing tools — the bugs found by ICCDROID are 1.7∼2.7 times more than the others. So far, ICCDROID has found 13 previously unknown ICC-related bugs, all of which have been confirmed by the app developers and five have already been fixed.**

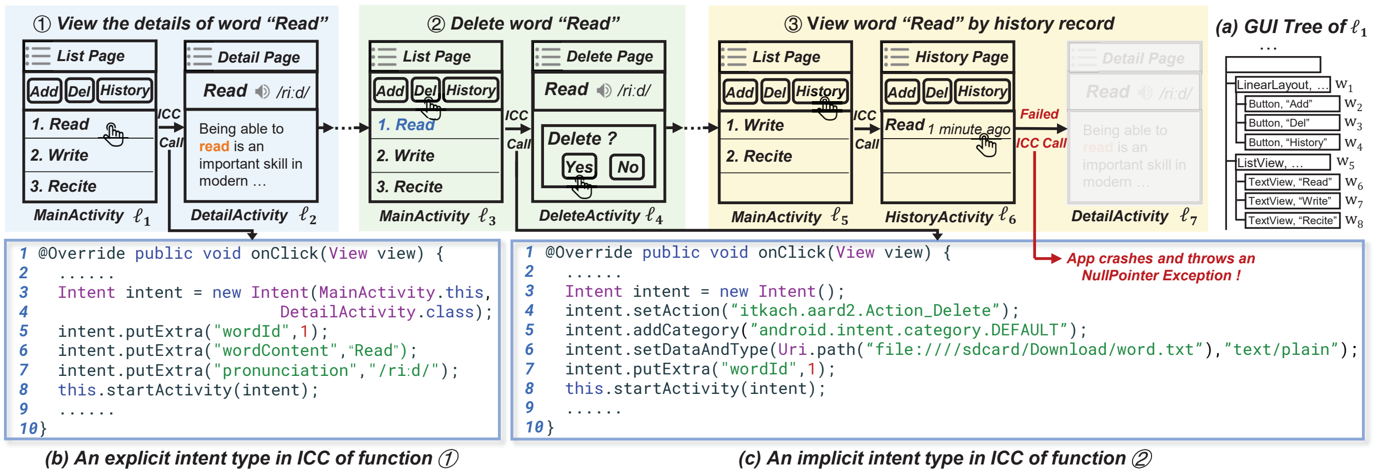
I. INTRODUCTION

Android apps are composed of four major components: *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. These components communicate with each other through the inter-component communication (ICC) mechanism. This ICC mechanism is implemented via an *intent* object. Leveraging the intent, the components can collaborate to achieve the app functionalities. Thus, the successful execution of ICC serves as the foundation for app operations. A failed ICC may result in the apps crash, leading to poor user experience [1], [2].

An illustrative example. Fig. 1 shows a real bug in *AARD2*. *AARD2* is an app for managing one's dictionary, which is released on Google Play. However, viewing word *Read* by history record has a serious error (shown in function ③ in Fig. 1). Specifically, a user first clicks a word *Read* to view the details of this word (see function ① in Fig. 1). Then the user performs some UI events (including navigation events) to make app jump from *DetailActivity* to *MainActivity* (see $\ell_2 \sim \ell_3$ in Fig. 1). Subsequently, the user deletes the visited word *Read* in *MainActivity* (see function ② in Fig. 1). As we expected, this word *Read* is successfully deleted in *MainActivity* (see ℓ_5 in Fig. 1). However, in this case, if the user views this word again through the history record, a *NullPointerException* is thrown when the app attempts to launch the *DetailActivity* of word *Read* (see function ③ in Fig. 1).

In this example, the event sequence of this bug consists of three sequentially executed functions (①→②→③). For each function, there exists an activity switching. Taking function ① as an example, after clicking word *Read*, the corresponding event handler (`onClick`) in Fig. 1.(b) creates an *intent* object and puts necessary messages (including target component and detailed word information) in this object (lines 3-7). With the help of `startActivity` method provided by Android system, the intent object implements the communication from *MainActivity* to *DetailActivity* (line 8). Therefore, the *DetailActivity* will be launched after the app executes `onClick` method. The process from intent creation to target component launching is called an ICC call. Unfortunately, the ICC call in function ③ fails as the deletion of word *Read* removes some crucial information (e.g., word pronunciation), which is necessary for the intent creation in function ③. In this paper, we name such

*Ting Su and Xiaoqiang Liu are the corresponding authors.



```
@Override public void onClick(View view) {
    ......
    Intent intent = new Intent(MainActivity.this,
                               DetailActivity.class);
    intent.putExtra("wordId",1);
    intent.putExtra("wordContent","Read");
    intent.putExtra("pronunciation","/ri:d/");
    this.startActivity(intent);
    ......
}
```

(b) An explicit intent type in ICC of function ①

```
@Override public void onClick(View view) {
    ......
    Intent intent = new Intent();
    intent.setAction("itkach.aard2.Action_Delete");
    intent.addCategory("android.intent.category.DEFAULT");
    intent.setDataAndType(Uri.path("file:////sdcard/Download/word.txt"),"text/plain");
    intent.putExtra("wordId",1);
    this.startActivity(intent);
    ......
}
```

(c) An implicit intent type in ICC of function ②

Fig. 1: A real ICC-related bug in AARD2 app

errors that occur during the ICC call as *ICC-related bugs*, which are induced by improper implementation of ICC calls.

However, detecting such ICC-related bugs is non-trivial due to two major reasons. First, the combination of different ICC calls could be complex and diverse. For example, triggering the bug in Fig. 1 requires a specific execution order of three ICC calls. The order ①→②→③ can crash the app, but if the user views a word record (function ③) before deleting this word (function ②), the app will perform successfully. Second, the ICC calls as well as the connections between different ICC calls need to be triggered by UI events. For example, in function ①, the ICC call is triggered by clicking the word *Read* and the connection between ① and ② (the transition from *DetailActivity* to *MainActivity*) is triggered by some other UI events (e.g., navigation).

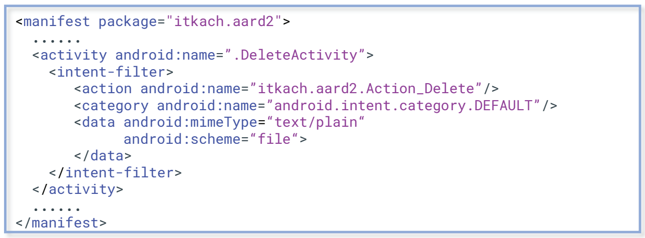
Limitations of existing techniques. Many automated GUI testing techniques have been proposed to detect bugs in Android apps. However, existing techniques have two major limitations in the detecting ICC-related bugs. First, most existing testing techniques [3]–[7] are generic and they are not specifically designed to find ICC-related bugs. Moreover, these techniques do not explicitly consider the execution orders of different ICC calls. Therefore, it is difficult for them to find such ICC-related bugs like in Fig. 1. Second, some techniques [8]–[15] design different *intent fuzzers* to mutate intents and then these intents are directly sent to apps to trigger ICC-related bugs. However, they are difficult in finding realistic ICC-related bugs because simply mutating intents may not make sense in real-world scenarios. Take the intent in Fig. 1.(b) as example, *intent fuzzers* can generate an intent whose *extras* field value is *null* or random key-value pairs (e.g., the key pair ("wordId", -100)). It is impossible for users to generate such intent through UI events.

Our approach. To this end, we propose a novel ICC-guided exploration strategy to effectively find the ICC-related bugs. Our idea is that, we can (1) build an ICC call graph from the app under test, and (2) use this graph to guide the exploration toward exercising ICC calls. Inspired by reinforcement learning-based testing techniques [16]–[20], we introduce Q-learning and design an ICC-related reward function to effectively guide the exploration to test these ICC calls in the apps. This reward function considers the number of explored intents, the number of promising-to-explore intents and the exploration order of explored intents during the exploration. It can encourage the Q-learning agent to generate meaningful test cases that contain complex and diverse ICC calls. Additionally, the introduction of Q-learning enables this approach to test apps like a normal user, ensuring that the exploration can intelligently connect different ICC calls. As a result, it is more promising to effectively reveal realistic ICC-related bugs.

Specifically, the ICC-guided exploration strategy depends on an ICC call graph of the app under test. We can use static analysis to generate the ICC call graph [21]–[25]. However, the generated graph may be incomplete due to the common weaknesses of static analysis [26], [27]. For example, some communication information (e.g., the values of intent fields) of target components are determined at app runtime, but the static analysis is difficult to capture such dynamic information. To this end, we propose a hybrid analysis technique combining static and dynamic analysis to build a more complete ICC call graph. First, according to the ICC mapping rules provided by Android development documentation [28], we conduct a static analysis on the app code to identify the target component of each intent object. In this way, a static ICC call graph can be built. Next, we design a graph enhancement exploration strategy also based on Q-learning to complement static analysis. It explores the UI events in the app to find additional ICC calls, which have not been included in the static ICC call graph.

Evaluation and results. We implemented our approach as an automated testing tool ICCDROID. The evaluation on 28 real-world Android apps shows that ICCDROID can find the most number of ICC-related bugs compared to existing tools within the same testing time. Specifically, the bugs found by ICCDROID are 1.7∼2.7 times more than the others. We have reported these ICC-related bugs revealed by ICCDROID to the app developers. To date, 13 reported bugs have been confirmed, and five of which have already been fixed.

Contributions. In this paper, we have made the following contributions:



```
<manifest package="itkach.aard2">
  ......
  <activity android:name=".DeleteActivity">
    <intent-filter>
       <action android:name="itkach.aard2.Action_Delete"/>
       <category android:name="android.intent.category.DEFAULT"/>
       <data android:mimeType="text/plain"
             android:scheme="file">
       </data>
    </intent-filter>
  </activity>
  ......
</manifest>
```

Fig. 2: A simplified Manifest.xml in AARD2 app

- We propose an ICC-guided exploration strategy based on reinforcement learning to effectively find the ICC-related bugs in Android apps.
- We combine static and dynamic analysis to generate a more complete ICC call graph, which improves the effectiveness of the ICC-guided exploration.
- We have implemented our approach as an automated testing tool ICCDROID[1]. The evaluation on 28 real-world apps shows that ICCDROID is much more effective and efficient than existing tools in finding ICC-related bugs.

II. BACKGROUND

A. Inter-component communication (ICC) mechanism

An Android app is composed of multiple components declared in *Manifest.xml*, these components are divided into four types: *Activity*, *Service*, *Content Provider* and *Broadcast Receiver*. The *List Page* (see ℓ_1 in Fig. 1) is an activity called *MainActivity*. The users can interact with the widgets on *MainActivity* (e.g., Button and TextView) to finish user-specific functions. For example, clicking word *Read* can trigger the UI event handler (see Fig 1.(b)) to achieve function ① (view the detail of word *Read*).

The communication mechanism between components can be implemented by *intent*. For example, The activity switching from *MainActivity* to *DetailActivity* (see function ① in Fig. 1) is implemented by the intent object in Fig. 1.(b). According to the ICC mapping rules provided by Android development documentation [28], [29], the target component in ICC is determined by two forms of intent: explicit intent and implicit intent. Explicit intent explicitly specifies the class of target component by setting the value of *component* field. The intent in Fig. 1.(b) is an explicit intent as it specifies the exact target component *DetailActivity* (line 3). Implicit intent does not specify the component class and the determination of target component depends on five intent fields (*action*, *category*, *data*, *type* and *extras*). Thus, the system needs to match the value of these fields with the *intent-filter* tag of each component declared in *Manifest.xml*.

For example, the intent in Fig. 1.(c) is an implicit intent and declares three conditions (the *action* name is `itkach.aard2.Action_Delete`, the *category* name is `android.intent.category.DEFAULT` and the data *type* that can be handled) for the target component (lines 4-6).

[1]https://github.com/androidAppGuard/IccDroid

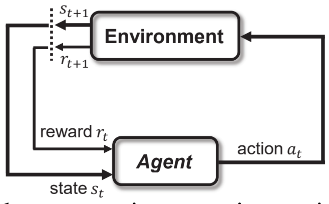


Fig. 3: Each agent-environment interaction in MDP

Then the system uses these values match each declared intent-filter (shown as Fig. 2). Finally, the *DeleteActivity* component matches successfully and is identified as the target component.

B. Q-learning

Q-learning is a well-known reinforcement learning algorithm. The reinforcement learning problem can be represented as a standard mathematical framework MDP (Markov Decision Process). This MDP can be defined as 4-tuple $\langle S, A, P, R\rangle$, where S is a set of all possible states, A is a set of all possible actions, P is the set of state transition probability and R represents the reward function. Fig. 3 shows the agent-environment interaction at each of the discrete time steps of a sequence. For each interaction t, the agent first observes the environment to obtain a state $s_t \in S$ and an immediate reward $r_t \in R$. The s_t and r_t have well-defined probability distributions dependent only on preceding state and action ($s_{t+1} \sim P(s_t, a_t)$ and $r_{t+1} \sim R(s_t, a_t)$). Then the agent selects an action $a_t \in A$ base on s_t and r_t to execute. This action can cause an environment transition from state s_t to state $s_{t+1} \in S$ and yield the next reward $r_{t+1} \in R$. After that, the next interaction starts and the process will continue until limit time runs out.

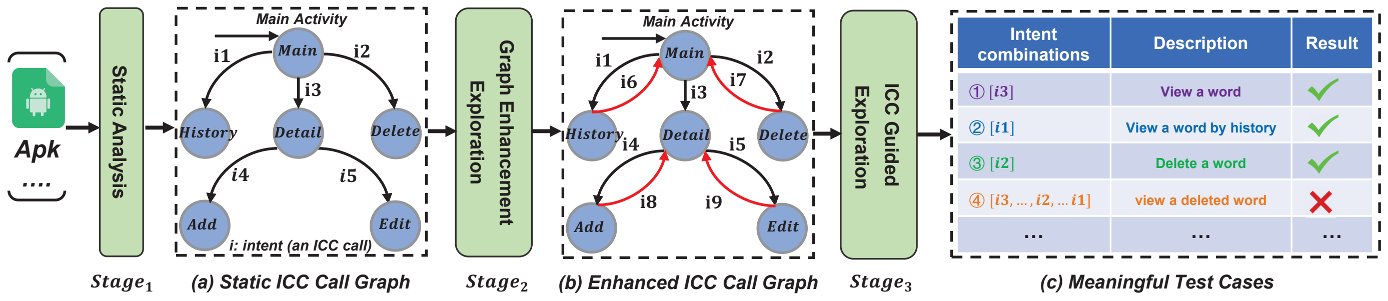
In Q-learning, the Q-table saves the value of each state-action pair (e.g., the Q-value $Q(s, a)$ represents the value of action a in state s) and is updated by equation (1). In this equation, α is the learning rate (range from 0 to 1) and $\underset{a}{max}\, Q(s_{t+1}, a)$ denotes the maximum future reward that can be achieved in state s_{t+1}. γ is the discount factor for future reward and usually between 0.8 and 0.99 in [17]–[19].

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \underset{a}{max}\, Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (1)$$

In equation (1), the value of subsequent state-action pairs $Q(s_{t+1}, a)$ can be propagated to the value of previous key-value pairs $Q(s_t, a)$. If the agent sufficiently explores the environment, the value of each state-action pair in Q-table will converge to its true value, as proven rigorously by [16]. Therefore, the Q-table can be precise to reflect the true value of each action in environment. In this case, if the agent chooses the action with the highest value at each interaction, the goal of maximizing cumulative reward can be achieved.

III. APPROACH

Fig. 4 describes the workflow of ICCDROID. ICCDROID takes an apk as input and outputs meaningful test cases that contain diverse intent combinations to reveal ICC-related bugs. The workflow contains three stages: one static analysis and two dynamic explorations based on Q-learning (graph enhancement exploration and ICC-guided exploration).



Intent combinations	Description	Result
① [$i3$]	View a word	✓
② [$i1$]	View a word by history	✓
③ [$i2$]	Delete a word	✓
④ [$i3, \ldots, i2, \ldots i1$]	view a deleted word	✗
...	...	...

Fig. 4: The workflow of IccDroid

Algorithm 1: Static analysis process

```
Function GenerateIccCallGraph(apk)
  $G \leftarrow \varnothing$                       // Initialize an ICC call graph
  $components \leftarrow getAllComponents(apk.manifestXml)$
  foreach $sendComponent \in components$ do
    foreach $method \in component.dalvikBytecode$ do
      if $containIntent(method.params)$ then
        $intent \leftarrow getIntent(method.params)$
        if $intent.type = explicit$ then          // explicit intent
          $targetComponent \leftarrow intent.component$
          $G \leftarrow G \cup \{sendComponent, intent, targetComponent\}$
        else                                      // implicit intent
          $fieldValues \leftarrow resolveIntentField(intent)$
          foreach $component \in components$ do
            if $match(fieldValues, component.intentFilter)$ then
              $targetComponent \leftarrow component$
              $G \leftarrow$
                $G \cup \{sendComponent, intent, targetComponent\}$
  return $G$
```

A. Static Analysis

The event sequence of such ICC-related bugs in Fig. 1 usually contains multiple ICC calls. To obtain these ICC calls, we conduct a static analysis on the *Manifest.xml* and DALVIK bytecode of the app by the ICC mechanism in Section II-A. Algorithm 1 shows the detailed analysis process.

Given an apk, we first extract all activity components (including the content of all *intent-filters*) from apk's *Manifest.xml* (line 3). For each component, all the methods in DALVIK bytecode are parsed to identify whether there is an ICC call (lines 4-16). If the method contains an intent parameter, we will further analyze the target component of this intent (lines 6-7). If it is an explicit intent, the target component class can be obtained from the *component* field and the edge from $sendComponent$ to $targetComponent$ will be added to the ICC call graph G (lines 8-10). Where each node is a component and each edge is a unique intent (ICC call). Otherwise it is an implicit intent, we refer to Android API document [28] to resolve the value of five intent fields that specify the target component (line 12). Then we will use these value to match the intent-filter of each component. If matching successfully, this component is the target component and the edge will also be added to G (lines 13-16). The stage continues until each method in each component is analyzed.

Example. The implicit intent in Fig. 1.(c) has set three fields for the target component. These value of intent fields can be resolved by the specified API methods (e.g., `setAction`, `addCategory` and `setDataAndType`). We further match these value with each intent-filter in *Manifest.xml* of Fig. 2 and identify the target component is *DeleteActivity*. Thus, the ICC call from *MainActivity* to *DeleteActivity* in function ② of Fig. 1 is identified and is denoted as *i2* (intent2) in Fig. 4.(a) (the simplified static ICC call graph for *AARD2*).

B. Dynamic Exploration based on Q-learning

In the two dynamic explorations, the goal of graph enhancement exploration is to generate a more complete ICC call graph and the goal of ICC-guided exploration is to effectively detect ICC-related bugs. The goals of these two explorations are clear and reinforcement learning is suitable for goal-driven scenarios. Thus, we mathematically model Android testing problem as an MDP and introduce Q-learning algorithm to achieve these goals. The definition of MDP 4-tuple $<S, A, P, R>$ takes the following formulation.

S: **States**. The goal of defining state is to distinguish different GUI pages where the layout of each GUI page is composed of a series of widgets. We utilize a variant of C-Lv4 GUI comparison criterion *GUICC* [30] to abstract one layout ℓ as the corresponding state s. Let ℓ be represented as a n-tuple $(w_1, w_2, \ldots, w_n)$ and w_i is the i_{th} widget in ℓ. We define state as s=$\cup_{w \in \ell}\{w.t\}$ where $w.t$ is the type of widget w, such as Button, TextView and TextEditor. This comparison criterion abstracts the structure characteristics by aggregating all the widget types in ℓ, which can effectively differentiate the overall layout of GUI pages.

For example, Fig. 1.(a) is the simplified GUI layout of ℓ_1. Because w_1, w_2, w_5 and w_6 can denote the all widget types in ℓ_1, the abstracted state is s=$\{w_1.t$,$w_2.t$,$w_5.t$,$w_6.t\}$. The layout ℓ_1 has one more TextView widget w_6 than ℓ_5 but the type of w_6 is same as w_7 and w_8, therefore ℓ_1 and ℓ_5 are abstracted as one state. The layout ℓ_1 and ℓ_2 are different states as the pronunciation picture (horn) widget only exists in ℓ_2.

A: **Actions**. We define the executable events as actions in MDP. In order to obtain these events, IccDroid dumps the layout hierarchy and analyzes the widget attributes (e.g., clickable, scrollable) to obtain executable events. Each executable event is saved as the corresponding state-action pair (s, a) in Q-table where a is an executable action in state s.

$$selectAction(s) = \begin{cases} \max\limits_{a} Q(s,a) & 1-\epsilon \\ random\ a\ action & \epsilon \end{cases} \quad (2)$$

We utilize the ϵ-greedy policy described in equation (2) to select next action. Under the probability of 1-ϵ, the agent selects an action with the highest value for exploiting previous exploration experience. In the remaining ϵ probability, the agent selects a random action (including system-level actions such as screen rotation and phone call) to explore the app. According to previous experiences [17]–[19], we set ϵ to 0.2 to balance *exploration* and *exploitation*.

R: **Reward**. The role of reward is to yield a numeric value for guiding agent toward a valuable exploration, hence it is crucial to design an effective reward function for triggering more bugs. We design two reward functions against the different exploration objectives during the two dynamic explorations.

P: **Transition Probability Function**. After performing action a_t, the transition probability $P(s_{t+1}|s_t, a_t)$ from state s_t to state s_{t+1} is decided by the app. Therefore, P is determined after the app development finishes.

Based on this MDP definition, we design two reward functions for the graph enhancement exploration and the ICC-guided exploration.

(1) Graph Enhancement Exploration

The ICC call graph generated by static analysis may be incomplete due to the imprecision of intent resolution. To complement the graph, we design an UI reward, based on the execution frequency of actions and the ICC calls at runtime, as the reward function of graph enhancement exploration.

UI reward function. The UI reward r_{ui} as defined as equation (3) consists of two parts: UI action reward and ICC call reward. The UI action reward $\frac{1}{|s_t, a_t|}$ denotes the reciprocal of execution frequency of action a_t in s_t. It signifies that the actions performed less have higher value. The ICC call reward is a sparse reward, which depends on whether the current UI action does trigger a new intent. If it does, there is a new ICC call and the reward value is 1; otherwise, it is 0.

$$r_{ui} = \begin{cases} \frac{1}{|s_t, a_t|} + 1 & \text{if new ICC call occurs} \\ \frac{1}{|s_t, a_t|} + 0 & \text{otherwise} \end{cases} \tag{3}$$

Based on this UI reward function, we utilize Q-learning algorithm to achieve a graph enhancement exploration. The detailed process is described in the function `graphEnhancementExploration` of Algorithm 2. For each interaction, IccDroid first abstracts structure characteristics of GUI page as state s_t and finds out all the executable actions $actions_t$ (line 5). The Q-table will record the value of each action or assign an default initial value 500 if the action executes for the first time (lines 6-8). Then IccDroid employs equation (2) to select an action and calculates the UI reward by equation (3) after executing this action (lines 9-11).

It is worth emphasizing that the graph enhancement exploration benefits from the propagation of Q-value (line 12), which enables agent to select actions with reward value regardless of the current state. If an executed action gains higher UI reward r_{ui}, its updated Q-value $Q(s_t, a_t)$ will increase and the action will be more possible to be executed in next explorations. Otherwise, the agent will explore other valuable actions. Since the UI reward function considers the action execution frequency, the actions performed less have higher reward and are prioritized. Thus, the agent sought to

Algorithm 2: Exploration Strategies

```
1  Function graphEnhancementExploration(G, app)
2      $Q \leftarrow \varnothing$ // initialize Q-table
3      $M \leftarrow \varnothing$ // save visited intent
4      while $\neg isTimeOver()$ do
5          $s_t, actions_t \leftarrow abstractStructureInfo()$
6          foreach $action_t \in actions_t$ do
7              if $action_t \notin Q$ then
8                  $setQvalue(Q, s_t, action_t, Q_{init})$
9          $a_t \leftarrow selectAction(Q, s_t)$
10         $s_{t+1} \leftarrow executeAction(app, a_t)$
11         $r_{ui} \leftarrow getUIReward(a_t, app)$
12         $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{ui} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
13         $intent \leftarrow getCurrentIntent(app)$ // use Java reflection
14         if $intent \neq null \wedge intent \notin M$ then // new ICC behavior
15             $M \leftarrow M \cup intent$
16             if $intent \notin G$ then // complement ICC call graph
17                 $G \leftarrow G \cup$
                       $\{intent.sendComponent, intent, intent.targetComponent\}$
18     return G
19 Function IccGuidedExploration(G, app)
20     $Q \leftarrow \varnothing$ // initialize Q-table
21     $seq \leftarrow$ List()
22     while $\neg isTimeOver()$ do
23         $s_t, actions_t \leftarrow abstractStructureInfo()$
24         foreach $action_t \in actions_t$ do
25             if $action_t \notin Q$ then
26                 $setQvalue(Q, s_t, action_t, Q_{init})$
27         $a_t \leftarrow selectAction(Q, s_t)$
28         $s_{t+1} \leftarrow executeAction(app, a_t)$
29         $r_{intent} \leftarrow getIntentReward(seq, G)$
30         $Q(s_t, a_t)$
               $\leftarrow Q(s_t, a_t) + \alpha(r_{intent} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
31         $seq.append(s_t)$
32         if $seq.length \geq c$ then
33             $restart(app)$
34             $seq.removeAll()$
```

systematically perform each UI action to find new dynamic ICC calls at runtime.

The sparse ICC call reward can encourage agent to find new ICC calls, which is used to complement the static ICC call graph. We first get the current runtime intent object in app (line 13). If the intent visits for the first time, it represents that the agent has explored a new ICC call (lines 14-15). Additionally, if this ICC call is not in the static ICC call graph, the graph will be updated (lines 16-17).

Overall, the graph enhancement exploration not only achieves a systematic testing for each UI event, but also generates a more complete ICC call graph that used for the next ICC-guided exploration.

(2) ICC-guided exploration

In order to reveal the ICC-related bugs like in Fig. 1, besides the single ICC calls, the combinations of different ICC calls also need to be considered. To this end, we design an intent reward for the ICC-guided exploration. This intent reward function considers the number of explored intents, the number of promising-to-explore intents (may be explored in subsequent actions) and the exploration order of explored intents to explore different intent combinations.

Intent reward function. We leverage the exploring state sequence seq and the ICC call graph G from graph enhancement exploration to calculate intent reward r. As shown in equation (4), r_{intent} consists of three parts: the number of explored intents, the number of promising-to-explore intents and the exploration order of explored intents. Where $f(seq, G)$ represents the number of intents currently explored by testing tool in G. It can be calculated through calculating the number of intents owned by both seq and the ICC call graph G. $g(seq, G)$ denotes how many intents that seq can continue to explore in G. We view the $g(seq, G)$ as the number of promising-to-explore intents. Both of $f(seq, G)$ and $g(seq, G)$ will be normalized by dividing by the maximum value (the number of all intents in G $|G|$). The last function $h(seq)$ is to identify whether the exploration order of explored intents in seq has been executed in prior episodes. If this exploration order is executed for the first time, $h(seq)$ is 1; otherwise 0.

$$r_{intent} = \frac{f(seq, G)}{|G|} + \frac{g(seq, G)}{|G|} + h(seq) \tag{4}$$

Example. Suppose G is the generated ICC call graph in Fig. 4.(b) and the exploring state sequence seq is [$s_1(Main)$,$s_2(Detail)$]. Where s_1 and s_2 is abstracted from *MainActivity* and *DetailActivity*, respectively. Because both seq and G contains the intent i_3 from *Main* to *Detail*, the number of explored intents $f(seq, G)$ is 1. The testing tool can continue to explore *Add* and *Edit* from s_2 through i_4 and i_5 of G, so the number of promising-to-explore intents $g(seq, G)$ is 2. Further, the number of all intents in G ($|G|$) is 9, then $\frac{f(seq,G)}{|G|}$ is 1/9 and $\frac{g(seq,G)}{|G|}$ is 2/9. Because this intent order $[i_3]$ is the first executed, the $h(seq)$ is 1. Thus, the total reward r_{intent} is 12/9.

In this exploration, we once again use Q-learning algorithm but take intent reward as the reward function to focus the execution of ICC calls, aiming to explore complex and diverse ICC combinations. The function `IccGuidedExploration` in Algorithm 2 shows the process of ICC-guided exploration. The whole process is similar to graph enhancement exploration. For each interaction, ICCDROID also first abstracts state (line 23), assigns initial value for each new action (lines 24-26), selects one action to execute (lines 27-28), then calculates the intent reward r_{intent} and updates the Q-table (lines 30-31). One difference is that ICCDROID needs to save the exploring state sequence seq. If the length of seq exceeds the specified test case length c (the default is 15), ICCDROID will restart the app and remove all states in seq to calculate r_{intent} for the next test case (lines 32-34). Through the definition of r_{intent}, if one test case contains more explored intents, can explore more promising-to-explore intents and this intent order has never been explored, r_{intent} will be higher, then these events in this test case can have a higher value through the propagation of Q-value, so they will be prioritized.

Thus, the intent reward can encourage the agent to generate long test cases that contain multiple intents and various execution order of intents. Then it is more promising to find ICC-related bugs.

IV. EVALUATION

In our experimental evaluation, we aim to answer the following three research questions:

- **RQ1** How effective is ICCDROID in finding ICC-related bugs, compared to existing testing tools?
- **RQ2** How efficient is ICCDROID in finding ICC-related bugs, compared to existing testing tools?
- **RQ3** Can the graph enhancement exploration strategy obtain a more complete ICC call graph? How effective is the ICC call graph in improving the testing effectiveness of the ICC-guided exploration strategy in bug finding?

A. Evaluation Setup

Tool Implementation. As a fully automated testing tool, ICCDROID reuses and extends the following tools: SOOT framework [31] for resolving static intents in Dalvik bytecode (similar to ICCBOT [21]), UIAUTOMATOR2 [32] for dumping GUI hierarchy files of GUI pages, Android Debug Bridge (ADB) [33] for sending UI events and LOGCAT [34] for recording ICC calls and monitoring runtime exceptions.

To identify the intents at runtime, ICCDROID first utilizes *Java* reflection to load the single-instanced *ActivityThread* class. Then the current activity object *mActivities* can be obtained by assessing the member variable of *ActivityThread*. Finally, the intent object and these intent fields can be directly fetched from *mActivities*.

App Subjects In this evaluation, we aim to select representative target apps by following the three steps. First, ANDROTEST [35] and THEMIS [36] are the two standard Android testing benchmarks used in several work [17], [37], [38]. Thus, we collected all the 78 open-source apps in these two benchmarks. Second, we excluded 21 apps that have not been maintained within the recent two years to ensure timely feedback from app developers. Third, to focus on real-world and non-trivial apps, we excluded 19 apps with fewer than 1000+ installations on *Google Play* and fewer than 100 stars on *Github*, and 10 apps which only have fewer than 2 activity components. Finally, a total of 28 apps are used as the app subjects in our evaluation. Table I shows these apps sorted by executable lines of code (ELOC) where the column **AC** denotes the total number of activity components in these apps.

Execution Environment. All the 28 open-source apps ran on 2GB RAM Android emulators (Marshmallow version, Android 7.0) deployed on a 64-bit Ubuntu physical machine with Intel(R) Core(TM) i5-3470 CPU and 16GB RAM.

Bug and Coverage Metrics. According to Android development documentation [28], the communication between components depends on the specified API methods (e.g., `startActivity()`, `startActivityForResult()` and `startActivityIfNeeded()`) to launch an ICC call. We call these specified API methods as *ICC methods* in this paper. Further, we follow the definition in STOAT [39], where an app bug is identified if the exception lines in stack traces contain the keyword of app's package name. For an app bug, if the call chain in the stack traces contains these ICC methods, it denotes this bug occurs during an ICC call and we identify such bugs as ICC-related bugs. Additionally, if one ICC-related bug is identified, we will also record the bug-triggering time to measure the efficiency of testing tools.

TABLE I: Testing results on the 28 open-source apps

Target App						%ICC Method Coverage					#ICC-related Bug					#Nodes		#Edges	
App Name	**#Install**	**#Star**	**ELOC**	**AC**	**ME**	Mo	Qt	Fa	Ma	Ic	Mo	Qt	Fa	Ma	Ic	**SA**	**HA**	**SA**	**HA**
PdfViewer	$10K^+$	341	1108	5	16	31.2	6.2	68.8	31.2	62.5	0	0	0	0	0	3	4	14	17
Privacyfriendlynotes	$10K^+$	76	3330	11	19	57.9	68.4	84.2	63.2	73.7	0	1	1	1	1	11	11	32	38
Swiftp	$1K^+$	678	5248	4	6	100	83.3	66.7	83.3	66.7	1	0	1	1	1	3	3	22	24
Dokuwiki	$10K^+$	51	5344	3	11	45.5	81.8	81.8	81.8	81.8	1	0	4	2	1	3	3	10	17
CatimaLoyalty	$10K^+$	469	6131	10	13	30.8	38.5	61.5	53.8	30.8	2	0	1	1	2	9	10	22	28
Aard2	$10K^+$	349	6619	12	10	30.0	30.0	20.0	20.0	70.0	0	0	1	2	1	2	4	7	14
BudgetWatch	$5K^+$	79	7430	10	16	43.8	43.8	81.2	25.0	25.0	0	0	2	1	0	8	9	28	36
Tomdroid	$50K^+$	16	8076	8	26	46.2	27.0	30.8	53.8	61.5	1	0	0	1	1	2	7	11	13
AlarmClock	$10M^+$	385	9092	5	10	50.0	60.0	50.0	60.0	60.0	0	1	0	1	1	2	4	9	13
Trackworktime	$5K^+$	100	10689	13	24	33.3	50.0	37.5	4.2	62.5	0	1	0	1	2	13	13	43	46
FairEmail	$500K^+$	1.8K	18073	4	38	18.4	2.6	13.2	18.4	13.2	1	0	0	1	1	4	4	17	21
Vanilla	$500K^+$	1K	18604	13	21	19.1	28.6	33.3	38.1	47.6	0	0	1	1	2	9	9	35	42
Timber	$100K^+$	17	20238	6	20	0.0	0.0	30.0	5	5	0	0	0	0	0	4	6	25	26
Material0istic	$10K^+$	20K	21919	23	35	42.9	60.0	42.9	20.0	37.1	1	2	1	0	0	22	22	115	131
BetterBatteryStats	–	565	22680	12	17	52.9	76.5	17.6	64.7	47.1	2	3	2	2	3	9	9	14	21
AnyMemo	$100K^+$	144	23486	28	67	13.4	7.5	16.4	43.3	50.7	0	0	0	3	4	16	20	28	37
Wikipedia	$50M^+$	1.9K	29557	40	122	3.3	0.0	5.7	6.6	10.7	0	0	0	0	4	40	40	121	151
Feeder	$10K^+$	374	31358	4	13	38.5	38.5	38.5	53.8	38.5	0	0	0	0	0	0	2	0	4
Runnerup	$50K^+$	655	34714	17	28	0.0	3.6	0.0	21.4	7.1	0	0	0	1	0	13	13	19	26
AmazeFileManager	$1M^+$	4.5K	34790	5	32	3.1	3.1	6.2	3.1	9.4	0	0	0	0	3	4	5	15	25
APhotoManager	–	214	36606	11	16	37.5	12.5	12.5	25.0	37.5	3	0	1	0	3	10	10	62	82
BookCatalogue	$100K^+$	369	41638	35	78	51.3	65.4	51.3	39.7	65.4	2	3	2	0	2	17	22	60	123
AntennaPod	$500K^+$	4.9K	47555	10	66	7.6	1.5	22.7	13.6	13.6	2	2	3	3	1	7	7	59	69
SuntimesWidget	–	264	47947	25	67	19.4	14.9	0	4.5	26.9	0	0	0	1	1	24	24	142	157
AnkiAndroid	$5M^+$	6.4K	50707	21	37	18.9	2.7	18.9	56.8	48.6	1	1	4	1	3	17	17	32	45
MyExpenses	$1M^+$	551	63276	32	79	3.8	6.3	2.5	8.9	29.1	0	0	0	0	2	30	30	194	221
Gadgetbridge	–	596	79798	36	58	0.0	3.5	0.0	3.5	3.5	0	1	1	0	1	28	28	46	63
K9Mail	$5M^+$	2.3K	93455	30	54	0.0	3.7	0.0	3.7	16.7	0	1	0	0	3	23	24	53	72
Average/Sum	-	-	-	15	36	28.5	29.7	31.9	32.4	39.0	17	16	25	24	43	330	360	1239	1545

[1] Column **AC**, **ME**, **#Install** and **#Star** denote the number of activity components, the number of ICC methods, the number of installations on *Google Play* and the number of stars on *Github*, respectively.

[2] Column Mo, Qt, Fa, Ma and Ic denote Monkey, Q-testing, Fax, MATE and IccDroid, respectively. Column SA and HA represent static analysis and hybrid analysis, respectively.

Code coverage is one mainstream way to represent the test adequacy of app source code during testing. However, traditional code coverage (e.g., line code coverage) may not be able to intuitively reflect the test adequacy of ICC calls in the app under test. Therefore, we define *ICC method coverage* to measure how many ICC methods are covered during testing. Because this metric can intuitively reflect the coverage of ICC calls in apps. To calculate the achieved ICC method coverage during testing, we instrumented each ICC method in each target app. The column **ME** in Table I shows the total number of ICC methods in each app.

Evaluation Setup of RQ1 and RQ2. Most existing automated testing tools [17], [39]–[43] for Android apps are generic. Although these testing tools mainly focus on UI events and do not explicitly consider the intents during testing, they also may find ICC-related bugs. Thus, we chose two representative tools (Monkey and Q-testing) for comparison. Additionally, some other testing tools [10], [11], [14], [15], [23], [24], [44], [45] leverage intents to test the ICC calls in the apps for finding ICC-related bugs. These testing tools are relevant to IccDroid. Thus, we also chose two representative tools (Fax and MATE) for comparison.

- Monkey and Q-testing: These two tools represent those typical generic testing tools. Monkey [40] is the state-of-practice testing tool, which sends pseudo-random events to the app under test. Q-testing [17], like IccDroid, is a testing tool based on reinforcement learning. It designs a curiosity-driven exploration strategy to select the event that is most likely to jump to a new page. These two testing tools are widely used as baselines for comparison in many work [20], [37], [39], [42], [46].
- Fax and MATE: These two tools represent those testing tools that explicitly consider the ICC calls during testing. Fax [14] first sends an intent to launch a specified activity and then injects random UI events (e.g., monkey events) to explore this activity. MATE [15] generates random tests with the combination of 90% UI events and 10% intents to discover more bugs. Fax and MATE consider the combination of UI events and intents and thus these two testing tools could detect ICC-related bugs. Other testing tools [10], [11], [23], [24], [44], [45] design different *intent fuzzers* to mutate intents. Then these intents are directly sent to the apps to detect failed ICC calls. However, these intents generated by *intent fuzzers* are unrealistic as the users are difficult to reach these scenarios through normal UI interactions. As a result, most bugs revealed by these testing tools cannot be reproduced [47], so we did not compare with them.

We allocated 2 hours for each evaluated tool in one run, and repeated 5 independent runs for each tool. Thus, the evaluation for 28 apps takes 5×2×5×28=1400 machine hours. According to the previous work [42], [43], we set 200 milliseconds event interval for Monkey. Q-testing is a closed-source testing tool, so we adopted the default configuration. For Fax and MATE, the 2-hour testing time includes the time required by their static analysis and dynamic exploration. Note that these two tools were setup with the assistance from the corresponding tool authors to ensure fair comparison. We allocated 1 hour for IccDroid to construct the ICC call graph (including the static analysis and graph enhancement exploration) and the remaining 1 hour for ICC-guided exploration.

Evaluation Setup of RQ3. IccDroid leverages the graph enhancement exploration to complement the static ICC call graph and uses the ICC-guided exploration to effectively find ICC-related bugs. Thus, we implemented two versions of IccDroid (IccDroid$^{\alpha}$ and IccDroid$^{\beta}$) as the baselines to evaluate the effectiveness of these strategies.

- IccDroid$^{\alpha}$: This baseline comparison aims to evaluate the effectiveness of the ICC-guided exploration in bug finding. Specifically, for the workflow in Fig. 4, IccDroid$^{\alpha}$ only runs $Stage_1$ (static analysis) and $Stage_2$ (graph enhancement exploration) but without $Stage_3$ (ICC-guided exploration). Within the 2-hour testing time, IccDroid$^{\alpha}$ runs $Stage_2$ until timeout after $Stage_1$ finishes. If IccDroid can find more ICC-related bugs than IccDroid$^{\alpha}$, we can conclude that the ICC-guided exploration is indeed useful.
- IccDroid$^{\beta}$: This baseline comparison aims to evaluate whether a more complete ICC call graph can improve the effectiveness of ICC-guided exploration in bug finding. Specifically, for the workflow in Fig. 4, IccDroid$^{\beta}$ runs runs $Stage_1$ (static analysis), $Stage_2$ (graph enhancement exploration *without updating the ICC call graph* like Fig. 4.(a)) and $Stage_3$ (ICC-guided exploration). Within the 2-hour testing time, IccDroid$^{\beta}$ runs $Stage_1$ and $Stage_2$ within the first one hour, and runs $Stage_3$ in the remaining one hour. If IccDroid can find more ICC-related bugs than IccDroid$^{\beta}$, we can conclude that a more complete call graph (with the help of graph enhancement exploration) is indeed useful.

We allocated the same 2-hour time for IccDroid, i.e., one hour for static analysis and the graph enhancement exploration and the remaining one hour for the ICC-guided exploration.

B. Experimental Results

Answer to RQ1. Table I shows the achieved ICC method coverage of five testing tools (Monkey, Q-testing, Fax, MATE and IccDroid) on the 28 apps. The gray cell highlights which tool achieves the highest coverage. The last row gives the overall average result. IccDroid achieved 39.0% ICC method coverage on average and exceeded Monkey by 10.5%, Q-testing by 9.3%, Fax by 7.1% and MATE by 6.6%. Additionally, IccDroid achieved the highest ICC method coverage on 15 apps, while Monkey, Q-testing, Fax and MATE win on 3, 6, 7 and 8 apps, respectively.

Performing an ICC method can launch an ICC call, so the number of the execution of ICC calls during testing is also important to achieve high ICC method coverage. During the 2-hour testing, Monkey, Q-testing, Fax, MATE and IccDroid averagely executed 9132, 9053, 9722, 9977 and 15337 ICC calls on 28 apps. For Monkey and Q-testing, they only focus on generating UI events but do not explicitly consider the ICC calls during testing. As a result, lots of UI events that can trigger ICC calls are missed and the number of executed ICC calls is only about 60% of IccDroid. Fax first sends an intent to launch a specified activity, and this process triggers one execution of ICC call. However, Fax only generates random monkey events to test the app after launching the activity. Thus, most of the testing time is also spent on UI events instead of intents. MATE ranks second in the number of the execution of ICC calls as its test cases are composed of 90% UI events and 10% intents. IccDroid designs an ICC call reward in graph enhancement exploration to encourage agent to explore unexecuted ICC calls. Additionally, the intent reward in the next ICC-guided exploration also drives IccDroid to explore the combinations of different ICC calls. Thus, IccDroid performed more ICC calls than other tools during testing.

Table I also shows the ICC-related bugs found by each tool. IccDroid discovered the most ICC-related bugs (43), which is 2.5 times more than Monkey (17), 2.7 times more than Q-testing (16), 1.7 times more than Fax (25) and 1.8 times more than MATE (24). Additionally, IccDroid detected the most number of ICC-related bugs in 17 apps, followed by MATE (9), Fax (7), Q-testing (6) and Monkey(5).

We also compared the ICC-related bugs revealed by the five testing tools. These tools found a total of 61 unique ICC-related bugs on 28 apps. Among these bugs, IccDroid achieved 70.4% detection rate of ICC-related bug, followed by Fax (40.9%), MATE (39.3%), Monkey (27.9%), and Q-testing (26.2%). For the number of unique ICC-related bugs, IccDroid detected the most unique ICC-related bugs (13) while Monkey, Q-testing, Fax and MATE only found 2, 2, 7 and 2. Fax found 7 unique ICC-related bugs as it sends intents to both internal activities and exposed intents while MATE only generates intents for exposed intents. As a result, Fax found many unique bugs in internal activities. However, in principle, Android system does not allow internal activities to be started by external apps. We have included these bugs even if they are difficult to trigger in real-world scenarios. IccDroid is the only tool that considers the execution order of ICC calls, so it can generate many long test cases that contain different ICC calls to discover deep ICC-related bugs and effectively find the most unique ICC-related bugs.

Most ICC-related bugs revealed by IccDroid have been reproduced and reported to developers. So far, 13 reported bugs have been confirmed. Among them, 5 have already been fixed. Table II lists the detailed information for each confirmed bug. Take a bug confirmed by developers in *AmazeFileManager* as example. This app throws a *NullPointerException* when pasting a deleted file. The shortest event sequence for triggering this bug requires 7 consecutive events. There are three different ICC calls ("copy a file", "delete a file" and "paste a file") in this buggy event sequence. The random exploration strategy of Monkey is hard to generate such long meaningful event sequence. Q-testing pays attention to the events that can discover new pages, but most events in this buggy event sequence do not change the page. As a result, Q-testing missed the bug. Even if Fax and MATE generated the intents to trigger these ICC calls during testing.

TABLE II: Confirmed bugs by testing tools

App Name	App Category	Issue ID	Bug State	Cause
AmazeFileManager	Tool	1	Confirmed	NullPointerException
AntennaPod	Player	2	Confirmed	XmlPullParserException
AlarmClock	Tool	3	Fixed	ActivityNotFoundException
Aadr2	Education	4	Fixed	NullPointerException
AnyMemo	Education	5	Confirmed	ExpatParserException
BookCatalogue	Education	6	Confirmed	DeadObjectException
BookCatalogue	Education	7	Confirmed	NullPointerException
APhotoManager	Life	8	Confirmed	RuntimeException
Betterbatterystats	Tool	9	Confirmed	UnavailableException
CatimaLoyalty	Finance	10	Fixed	IllegalArgumentException
CatimaLoyalty	Finance	11	Fixed	ActivityNotFoundException
Swiftp	Tool	12	Fixed	ActivityNotFoundException
Simpletask	Tool	13	Confirmed	IllegalArgumentException

However, neither of them considers the execution order of intents, so the two tools both missed this bug. Fortunately, the enhanced ICC call graph generated by hybrid analysis contains these three ICC calls, and the intent reward designed for ICC-guided exploration considered the execution order of different intents. This bug was found by IccDroid when the ICC-guided exploration ran for only 15 minutes, which shows the effectiveness of our approach.

In summary, IccDroid *found the most number of ICC-related bugs compared to existing tools. Specifically, the bugs found by* IccDroid *are 1.7∼2.7 times more than the others.*

Answer to RQ2. Fig. 5 shows the number of revealed ICC-related bugs over the testing time. At the first 5 minutes, IccDroid, Fax and MATE conducted static analysis on the target apps, so the number of detected ICC-related bugs is 0. After generating the ICC call graph, IccDroid entered the graph enhancement exploration. The ICC call reward designed for this stage encourages the agent to explore new ICC calls to complement the static ICC call graph, so some ICC-related bugs were found. As a result, IccDroid took the first place at about 13 minutes. It indicates that IccDroid can find several ICC-related bugs within a short time. As the progress of testing continues, these testing tools are difficult to find new ICC-related bugs. Therefore, after about 37 minutes, the curves of these tools saturate.

After 1 hour, IccDroid entered the ICC-guided exploration. During this stage, the intent reward considers the number of explored intents, the order of explored intents and the number of promising-to-explore intents, which enables the agent to explore complex and diverse combinations of ICC calls. Thus, IccDroid can continuously discover new deep ICC-related bugs and keep top one during the subsequent testing. Overall, in term of detecting ICC-related bugs, IccDroid has exhibited high testing efficiency.

In summary, IccDroid *is more efficient than existing tools in finding ICC-related bugs within the same testing time.*

Answer to RQ3. The last four columns in Table I shows the number of nodes and edges in the ICC call graph generated by static analysis (IccDroid$^\alpha$) and hybrid analysis (IccDroid). We can see that the graphs generated by IccDroid has 30 more nodes and 306 more edges than the graphs generated by IccDroid$^\alpha$. This shows the graph enhancement exploration enhanced the static ICC call graph and the hybrid analysis approach generated a more complete ICC call graph. This phenomenon benefits from the ICC call reward in the graph enhancement exploration, which enables agent to dynamically explore new ICC calls to complement the static ICC call graph.

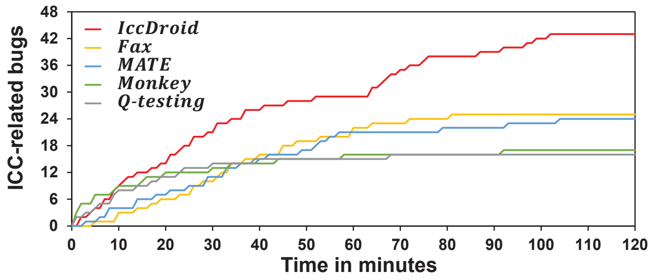


Fig. 5: Progressive ICC-related bugs revealed by testing tools

Fig. 6 shows the achieved ICC method coverage and the number of revealed ICC-related bugs over execution time for IccDroid, IccDroid$^\alpha$ and IccDroid$^\beta$. For the achieved ICC method coverage, IccDroid$^\alpha$ achieved 40.7% ICC method coverage and higher than IccDroid (39.0%) and IccDroid$^\beta$ (37.8%). We can see that IccDroid, IccDroid$^\alpha$ and IccDroid$^\beta$ achieved close code coverage in the previous 1 hour. After 1 hour, IccDroid$^\alpha$ continued the graph enhancement exploration while IccDroid and IccDroid$^\beta$ entered the ICC-guided exploration. After about 76 minutes, IccDroid$^\alpha$ exceeded the other two tools and kept top one during the subsequent testing. This phenomenon shows the graph enhancement exploration can indeed guide the tool to explore new ICC calls. Thus, the dynamic graph enhancement exploration can generate a more complete ICC call graph.

For the revealed ICC-related bugs, IccDroid found the most number of ICC-related bugs (43) while IccDroid$^\alpha$ and IccDroid$^\beta$ are 32 and 36. In the previous 1 hour, all the three tools entered the static analysis and the graph enhancement exploration, so the number of revealed ICC-related bugs were close. After 1 hour, IccDroid and IccDroid$^\beta$ entered the ICC-guided exploration while IccDroid$^\alpha$ continued the graph enhancement exploration. As a result, IccDroid and IccDroid$^\beta$ exceeded IccDroid$^\alpha$ after about 69 minutes. This phenomenon shows that the ICC-guided exploration can improve the ability of discovering ICC-related bugs. Additionally, the ICC-guided exploration of IccDroid used the enhanced ICC call graph while the ICC-guided exploration of IccDroid$^\beta$ continued to use the static ICC call graph. As a result, IccDroid found more 7 ICC-related bugs than IccDroid$^\beta$ in the ICC-guided exploration. It means the more complete ICC call graph can improve the ability of tools to found ICC-related bugs.

In summary, the ICC call graphs built by the hybrid analysis are more complete than those built by the static analysis. These more complete ICC call graphs help the ICC-guided exploration find more unique ICC-related bugs.

V. Discussion

Enumeration of ICC call combinations. Enumerating all the possible combinations of ICC calls via full permutation might be plausible to find more ICC-related bugs. However, we may face two challenges. First, the enumeration could be

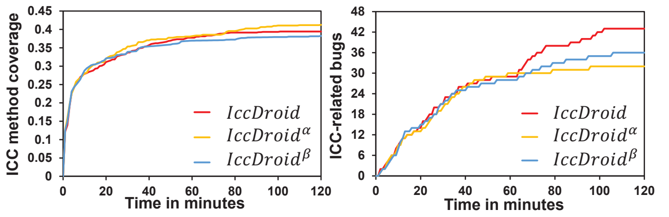


Fig. 6: The achieved ICC method coverage and the number of revealed ICC-related bugs over execution time

impractical as the number of combinations increase exponentially when the number of ICC calls is large. Second, simply enumerating the combinations of ICC calls may not be able to ensure the connections between different ICC calls. These connections usually require the execution of some additional UI events. As a result, most of the combined ICC calls could be invalid. Thus, we propose a ICC-guided exploration strategy to explore diverse combinations of ICC calls and introduce Q-learning to test the app by connecting different ICC calls.

Threats to validity. The main threat is the identification of ICC-related bugs. We identify an ICC-related bug by analyzing its call chain in stack traces. However, some other exceptions (e.g., *SecurityException*) may lead to similar call chains, thus incurring false positives. To reduce this threat, we reproduced all the found bugs with such exceptions, and manually analyzed the root causes to confirm whether they were true ICC-related bugs.

VI. Related Work

Automated testing tools for Android apps. Most automated GUI testing tools [48]–[50] have been proposed to ensure the reliability and quality of Android apps, including random testing tools [40], [51], [52], model-based testing tools [30], [39], [42], [47], [53]–[55], search-based testing tools [41], [43], [56]–[58] and reinforcement learning-based testing tools [17]–[20], [59]. However, these testing tools focus on generating UI events to test apps but do not explicitly consider the ICC calls during testing. Therefore, it could be difficult for these tools to effectively find ICC-related bugs. In contrast, our tool IccDroid utilizes the hybrid analysis to generate an ICC call graph for the app under test and uses an ICC-guided exploration strategy to effectively find ICC-related bugs.

Recently, some GUI testing tools [17]–[20], [59]–[62] adopt reinforcement learning to do testing. Q-testing [17] designs a curiosity-driven exploration strategy to explore previously unvisited GUI pages. Vuong [19] utilizes the execution frequency of events as the reward function to systematically test each event in the app. However, these testing tools do not specifically design an ICC-related reward function to explore ICC-related bugs. Thus, they are limited in effectively finding ICC-related bugs. IccDroid designs a ICC-guided exploration based on an ICC call graph, aiming to comprehensively test these ICC calls in the apps.

Finding ICC-related bugs in Android apps. Some testing tools [2], [9]–[13], [45], [63] design different *intent fuzzers* to mutate intents. These intents are then directly sent to apps to trigger ICC-related bugs. For example, NullIntentFuzzer [63] generates intents with *null* value for all fields to detect failed ICC calls. IntentDroid [11] fuzzes different value for intent fields that used in apps to simulate various ICC calls, aiming to comprehensively find ICC-related bugs. ICCFuzer [13] first conducts a path-insensitive ICFG analysis to resolve the possible value of intents fields and these values are used to generate real intents in the actual processing of apps. However, these intents generated by different *intent fuzzers* are unrealistic as the users are difficult to reach these scenarios through normal UI interactions. We introduce reinforcement learning to enable IccDroid to test the app like a normal user, ensuring the agent can intelligently select realistic actions in the apps to discover realistic ICC-related bugs.

To our knowledge, few work [14], [15] leverages different combinations of intents and UI events to achieve higher coverage and find more bugs. Fax [14] first sends an intent to launch a specified activity and then injects random UI events (e.g., monkey events) to explore this activity. MATE [15] generates test cases with random combinations of 90% UI events and 10% intents to discover more bugs. However, Fax and MATE do not consider the execution order of intents during testing. The simple and random combinations of UI events and intents are difficult to find deep ICC-related bugs. IccDroid designs an intent reward for ICC-guided exploration to consider the number of explored intents, the number of promising-to-explore intents and the execution order of explored intents. This reward enables the agent to explore diverse combinations of ICC calls for effectively finding more deep ICC-related bugs.

VII. Conclusion

In this paper, we propose a novel approach to effectively find ICC-related bugs. Our approach designs a graph enhancement exploration strategy based on Q-learning to complement the ICC call graph generated by static analysis. Leveraging this graph, we further design an ICC-guided exploration strategy also based on Q-learning to improve the testing effectiveness in bug finding. We also implemented this approach as an automated testing tool IccDroid. The ICC-related bugs found by IccDroid on 28 target apps are 1.7∼2.7 times more than the others. We have reported these ICC-related bugs revealed by IccDroid to the app developers. So far, 13 reported bugs have been confirmed and five of which have already been fixed.

VIII. Acknowledgments

We thank the anonymous ISSRE reviewers for their valuable feedback and the valuable assistance from Jiwei Yan (University of Chinese Academy of Sciences) and Michael Auer (University of Passau) for runing their tools. This work was supported in part by NSFC Grant 62072178, National Key Research and Development Program (Grant 2022YFB3104002), "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, National Key Research and Development Program (Grant 2020AAA0107800), and the Shanghai Collaborative Innovation Center of Trusted Industry Internet Software.

References

[1] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[2] K. Choi, M. Ko, and B.-M. Chang, "A practical intent fuzzing tool for robustness of inter-component communication in android apps," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 12, no. 9, pp. 4248–4270, 2018.

[3] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.

[4] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[6] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A multi-agent model-based test generation system beijing bytedance network technology co., ltd." in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 93–96.

[7] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[8] X. Huang, A. Zhou, P. Jia, L. Liu, and L. Liu, "Fuzzing the android applications with http/https network data," *IEEE Access*, vol. 7, pp. 59 951–59 962, 2019.

[9] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014, pp. 1–5.

[10] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "Intentfuzzer: detecting capability leaks of android applications," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 531–536.

[11] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 118–128.

[12] J. Jenkins and H. Cai, "Dissecting android inter-component communications via interactive visual explorations," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 519–523.

[13] T. Wu and Y. Yang, "Crafting intents to detect icc vulnerabilities of android apps," in *2016 12th International Conference on Computational Intelligence and Security (CIS)*. IEEE, 2016, pp. 557–560.

[14] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 457–468.

[15] M. Auer, A. Stahlbauer, and G. Fraser, "Android fuzzing: Balancing user-inputs and intents," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 37–48.

[16] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[17] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[18] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.

[19] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.

[20] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *arXiv preprint arXiv:2101.02636*, 2021.

[21] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "Iccbot: fragment-aware and context-sensitive icc resolution for android applications," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 105–109.

[22] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 176–186.

[23] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[24] J. Jenkins and H. Cai, "Icc-inspect: Supporting runtime inspection of android inter-component communications," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 80–83.

[25] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 576–587.

[26] C. Izurieta, D. Rice, K. Kimball, and T. Valentien, "A position study to investigate technical debt associated with security weaknesses," in *Proceedings of the 2018 International Conference on technical debt*, 2018, pp. 138–142.

[27] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE software*, vol. 19, no. 1, pp. 42–51, 2002.

[28] Google, "Intent," https://developer.android.com/reference/android/content/Intent, 2023.

[29] ——, "Intents and intent filters," https://developer.android.com/guide/components/intents-filters, 2023.

[30] Y.-M. Baek and D.-H. Bae, "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*, 2016, pp. 238–249.

[31] Google, "Soot framework," https://github.com/soot-oss/soot, 2023.

[32] Github, "Ui automator2," https://developer.android.com/training/testing/other-components/ui-automator, 2023.

[33] Google, "Android debug bridge," https://developer.android.com/studio/command-line/adb, 2023.

[34] ——, "Logcat command-line tool," https://developer.android.com/studio/command-line/logcat, 2023.

[35] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[36] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 119–130.

[37] H. Guo, X. Liu, B. Li, L. Cai, Y. Hu, and J. Cao, "Sqdroid: A semantic-driven testing for android apps via q-learning," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 301–310.

[38] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "Understanding and finding system setting-related defects in android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 204–215.

[39] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[40] Google, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, 2022.

[41] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[42] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 469–480.

[43] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 481–492.

[44] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, "Security code smells in android icc," *Empirical software engineering*, vol. 24, no. 5, pp. 3046–3076, 2019.

[45] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013, pp. 68–74.

[46] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering*, 2020.

[47] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[48] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su, "An empirical study of functional bugs in android apps," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1319–1331.

[49] J. Sun, T. Su, K. Liu, C. Peng, Z. Zhang, G. Pu, T. Xie, and Z. Su, "Characterizing and finding system setting-related defects in android apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 04, pp. 2941–2963, 2023.

[50] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs." 2020.

[51] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[52] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[53] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2014.

[54] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[55] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, and A. Vernotte, "Recent advances in model-based testing," *Advances in computers*, vol. 101, pp. 53–120, 2016.

[56] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[57] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[58] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "Agrippin: a novel search based testing technique for android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 5–12.

[59] Y. Koroglu and A. Sen, "Reinforcement learning-driven test generation for android gui applications using formal specifications," *arXiv preprint arXiv:1911.05403*, 2019.

[60] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 105–115.

[61] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *2011 IEEE fourth international conference on software testing, verification and validation workshops*. IEEE, 2011, pp. 252–261.

[62] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.

[63] GitHub, "Nullintentfuzzer," https://github.com/MindMac/IntentFuzzer, 2023.