Multiple-Entry Testing of Android Applications by Constructing Activity Launching Contexts

Jiwei Yan
Tech. Center of Softw. Eng.
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China
yanjw@ios.ac.cn

Hao Liu
Dept. of Informatics
Beijing University of Tech., China
Beijing, China

Linjie Pan
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

Jun Yan ∗
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

Jian Zhang ∗
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

Bin Liang
School of Information
Renmin University of China
Beijing, China

ABSTRACT
Existing GUI testing approaches of Android apps usually test apps from a single entry. In this way, the marginal activities far away from the default entry are difficult to be covered. The marginal activities may fail to be launched due to requiring a great number of activity transitions or involving complex user operations, leading to uneven coverage on activity components. Besides, the test space of GUI programs is infinite, it is difficult to test activities under complete launching contexts using single-entry testing approaches.

In this paper, we address these issues by constructing activity launching contexts and proposing a multiple-entry testing framework. We perform an inter-procedural, flow-, context- and path-sensitive analysis to build activity launching models and generate complete launching contexts. By activity exposing and static analysis, we could launch activities directly under various contexts without performing long event sequence on GUI. Besides, to achieve an in-depth exploration, we design an adaptive exploration framework which supports the multiple-entry exploration and dynamically assigns weights to entries in each turn.

Our approach is implemented in a tool called Fax, with an activity launching strategy FaxExp and an exploration strategy FaxEx. The experiments on 20 real-world apps show that FaxExp can cover 96.4% and successfully launch 60.6% activities, based on which FaxEx further achieves a relatively 19.7% improvement on method coverage compared with the most popular tool Monkey. Our tool also behaves well in revealing hidden bugs. Fax can trigger over seven hundred unique crashes, including 180 Errors and 539 Warnings, which is significantly higher than those of other tools. Among the 46 bugs reported to developers on Github, 33 have been fixed up to now.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

KEYWORDS
Android app, Static Analysis, ICC, Multiple-Entry Testing

ACM Reference Format:

1 INTRODUCTION
Mobile applications, especially Android apps, have witnessed an explosive growth in recent years. Meanwhile, the defects in Android applications aroused the attention of researchers [54]. Due to the event-based characteristic of Android apps, the test cases are in the form of GUI events. For GUI testing, a great number of automatic exploration approaches have been proposed, including random [24, 29, 60], model-based [28, 57, 58] and systematic ones [4, 22], aiming to cover more components or transitions. Despite using different exploration strategies, these approaches usually start their exploration from the default entry point, i.e., MainActivity of the target app. In this paper, we refer them as Single-Entry Testing (SET) ones.

In SET approaches, some obstacles, e.g., complex gesture or logical operations, make some activities unreachable. Besides, each activity has an implied exploration distance from the single entry point, which is unequal and leads to uneven coverage on activity components. One recent work makes use of the state-of-the-art tool Monkey [29] to test a widely used app WeChat, and has a similar observation: Monkey allocates a lopsided distribution of exploration time on each activity [60]. Furthermore, because of the
infinite test space of GUI programs, it is difficult to launch activities under complete contexts using SET. To address these issues and achieve in-depth GUI testing on mobile apps, we take each activity as a separate entry and propose the Multiple-Entry Testing (MET) approach.

Fig. 1 shows the exploration paths in SET and MET. Under SET, the activity Account can not be launched if its domain node, e.g., Detail1, failed to be visited. In addition, Account has different behaviors depending on its Activity Launching Contexts (ALCs), which is generated by previous event operations or received from outside (e.g., another app). Using SET, it is difficult to cover all ALCs and measure the test adequacy involving activity launching. If we adopt MET, Account can obtain a fair launching chance as the default entry and can be tested completely under multiple ALCs.

**Figure 1: Single- and Multiple-Entry Testing**

Usually, the user can successfully launch the default entry of an app by simply clicking the icon on the launcher of the phone, or sending an external command with the name of the target activity. However, a great number of activities require specific data items during launching and decide the execution paths according to the type and value of these data items. That is to say, without proper ALCs, the MET process may be incomplete and ineffective. In Android, several context-related information may influence the launching result, including the inter-component communication (ICC) message, device configuration, activity stack, and global data, among which the ICC message is the most important one. To generate ICC messages properly, we should precisely model the name, type, data structure as well as the value constraints of each attribute.

**Challenges.** The first challenge lies in the modeling of the complex attributes of ICC messages. Existing works [26, 45, 56, 59] extract the attribute declarations by XML file analysis. However, according to our research, the attribute declaration of activity is usually inconsistent with its usage. Therefore, it is necessary to model the ICC information by extracting the constraints of attribute usage on different execution paths. According to the APIs provided by Android Reference [20], Activity can receive two types of data items: Basic Attribute and Extra Parameter, in which the Extra Parameter can be further separated into Prime, Object and Bundle Extra Parameter according to the value it contains. For a basic attribute, we concentrate on the candidate values that can influence the branches; for a primary extra parameter, we should extract both the extra type and its key for input generation; an object extra parameter requires an object data item, which means a corresponding object must be instantiated first as test data; and for a bundle extra parameter, which is a nestable key-value map, we should reconstruct its original data structure. To generate proper ALCs as much as possible, all of these characteristics should be considered.

The second challenge is the weight assignment among entries in MET. Different from SET approaches, which always start testing from the default entry, the MET requires us to make decisions on the exploration weight of each entry. On one hand, some activities fail to be launched by the constructed ALCs due to unexpected app crashes. Thus the exploration should make more efforts to cover these activities for testing. On the other hand, the activities that can lead to more activity transitions should have higher exploration weight. For example, the default entry is usually more important than the leaf activities which never jump out. The static Activity Transition Graph (ATG) can be built for help, however, it may be imprecise. During testing, the activity launching result changes dynamically and new transitions can be detected. Thus, the weight assignment should be adaptive during the MET process.

**Our Approach.** In this paper, we propose the MET approach to replace the traditional SET approaches. We first analyze the ICC receiving process and perform an inter-procedural, flow-, context- and path-sensitive analysis to construct the Activity Launching Models (ALM), which describes the constraints of required attributes and parameters in each activity. Each ALC in the ALM corresponds to a unique intent-resolving related execution path. Then we calculate the weights of ALCs according to both the activity launching status and the reachability information between activities. The exploration is designed for multiple-round testing, which first launches each ALC and assigns the initial weight. Then it increases the number of total events in each round and reassigns the testing weight dynamically. Finally, we can achieve an in-depth exploration by ALC generation and adaptive weight assignment.

**Contributions.** The contributions of this work are three-fold:

- **Context Construction.** We perform static analysis to build ALMs for activities and generate proper ALCs for them.
- **Exploration Framework.** We design a general adaptive framework for multiple-entry testing, which supports two strategies: the activity launching strategy Fax\_\text{ex} and the exploration strategy Fax\_\text{ex}
- **Tool Implementation.** We implement our approach in the tool Fax [13], which is publicly available on GitHub. The experimental results indicate that Fax has a strong bug detection ability and reaches high code coverage.

## 2 BACKGROUND

This section provides an introduction of the fundamental building blocks in Android apps.

### 2.1 Android Activity

Activity, which provides a graphical user interface to users, is the most frequently used component in the Android system. The user performs UI operations on activities and triggers activity transitions to complete their daily tasks. If a user triggers an activity transition, the caller activity will send an ICC message according to the Intent mechanism [20]. Then the current activity will be stopped and the new one will be launched, which is managed by the Android system. Each ICC message contains a specific invocation target as well as a series of data items. According to whether the target...
activities can be launched by external apps, they can be separated into internal activity (IA) and exported activity (EA). The IAs can only be visited through a set of user operations that start from the EAs, while the EAs can be taken as hidden entries of the app and be launched directly. To visit one activity directly, we can modify it as an EA to support one step invocation by declaring the attribute android:exported=true or setting intent-filters in the manifest file. After exported, even internal activities can be launched directly without performing complex user event sequences.

2.2 Activity Launching Context

According to the activity launching process, four types of sub-context may have influences on the launching result.

- **ICC Message.** An ICC message is an Intent object carried with a set of data items, which depends on the caller activity.
- **Device Configuration.** The device configuration denotes the user-configurable status of the mobile phone, e.g., wifi, GPS status, which depends on the setting of the phone.
- **Activity Stack.** The activity stack stores the history activities visited before reaching the current one, which depends on the exploration trace.
- **Global Data.** The global data can be modified anywhere, which depends on the previous user operations.

To achieve effective testing of Android apps, we try to create ALCs that can trigger as many program paths as possible using these sub-contexts. For the ICC message, we model their usage with the help of static analysis to achieve backward ALC generalization without performing complex user event sequences.

2.3 ICC Message and Intent Receiving

ICC message is one of the most important sub-context, which has a complex composition and is necessary for the activity launching. Table 2 lists the required attributes in ICC invocation and gives their characteristics, in which columns Loc_d and Loc_u denote the declaration and the usage locations of these attributes, respectively. In this table, we classify ICC attributes into two types: **Basic Attribute** and **Extra Parameter.** As we can see, the Basic Attribute including action, category, data and type, can be declared both in the intent-filters in the manifest file as well as used in the Java files. However, there are mismatches between the attribute declaration and its usage. For example, the activity MessageList in popular app k9Mail [42], which has over 4000 stars on Github, requests three actions in Java code, in which only one of them is declared in the manifest. This activity also declares two values in the intent-filter in the manifest file, but one is invalid and not used in the Java code.

<table>
<thead>
<tr>
<th>Type</th>
<th>SubType</th>
<th>Loc_d</th>
<th>Loc_u</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Category</td>
<td>Data</td>
<td>Type</td>
<td>Action</td>
</tr>
<tr>
<td>Data</td>
<td>Type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extra</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bundle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Besides **Basic Attribute**, ICC messages also accept **Extra Parameters**. Each extra parameter is a key-value pair (k, v), which can be separated into three sub-types: Primary Extra Parameter, Object Extra Parameter and Bundle Extra Parameter, according to the type of the value v. Different from the basic attribute, the extra parameters are not declared anywhere, but only used in code. The caller activity can attach an intent with different types of extra data items via a series of overloaded APIs. Android system provides a number of APIs for the receiver activity to get the transferred data according to the given key. According to Android API document [8, 20], the value of extra parameter can be any type of the Java primitive data type, e.g., Int, Boolean, or other types like String, Array and ArrayAdapter, etc. For example, the API getIntExtra(String city) is used to get an integer value according to the key city. The value of an extra parameter can also be object type (Serializable and Parcelable) or bundle type (Bundle), in which the object type denotes an object implementing a specific interface, and a bundle type is a set of key-value pairs that stores a group of sub-items in types of primary, object or nested bundle extra parameter.
3 MOTIVATING EXAMPLE

In order to show the process of how the ICC messages are received by activities as well as how the ICC attributes/parameters are used in the code, we take ExampleActivity as our motivating example, which is shown in Fig. 2.

When an activity is launched, the Android system will call its lifecycle methods. In this example, we start from its lifecycle method onCreate(). First, the activity gets an Intent object through the API getIntent() and creates an Intent variable intent to store the information of the received ICC message. Then, the value of each attribute carried in the ICC message will be obtained through several APIs, e.g., using API getAction() to get the value of basic attribute action, using API getIntentExtra(key) to get the value of primary extra parameter with a specific key, etc. After that, the attribute value receiving variables will be used for branch picking, log recording or other purposes. When the received value of an attribute is used in a branch picking condition through comparison, there should be candidate values of this attribute that can influence the program’s execution path. For example, the statement if (mAction.equals("ACTION_VIEW")) contains an attribute receiving variable mAction, a comparing operation equals and a candidate value "ACTION_VIEW" of the basic attribute action.

However, not all of the candidate values can be obtained directly. On one hand, the attribute receiving variable may transfer its value to other variables and form new condition constraints. Like in line 6, the attribute receiving variable mAction1 is not used in branch picking conditions. As shown in the following lines, its value is transferred to variables action2 and action3. The new constraints are getAction().substring(1, 4).equals ("act") and getAction().charAt(2)=='C'. On the other hand, the string-type candidate values may not be obtained directly. It could be manipulated by a set of String-related APIs, come from the formal parameter of the current method or be the return value of another method. In line 13, for instance, the candidate value "sta" is decided by the return value of method getPrefix(), i.e., it depends on both the value of the formal parameters of method getPrefix() and the semantic of API getString(). In addition to string analysis, the received ICC message may have specific structures. As shown in lines 14-16, each extra parameter may belong to different data types, and may have specific structure (b1-bundle, s1-string, b1.b2-bundle, b1.b2.f3-fl2class), which should be precisely reconstructed in order to generate dummy caller activities with correct ICC message as test cases.

4 FRAMEWORK OVERVIEW

In Fig. 3, we give the framework overview of Fax, which takes an apk file as input and generates a group of test cases. After test execution, it outputs a set of corresponding reports. First, we instrument the original apk to expose IA into EA, which only modifies the manifest file and does not bring extra overhead in dynamic testing. Then, we perform static analysis to get the Activity Launching Model (ALM) that describes the attribute usage information as well as the Activity Transition Graph (ATG) that shows the relationship of activities. We use the ALM to generate ALCs of activities and perform test case execution on Android devices. Because the generated test scripts only launch activities under various contexts without GUI exploration, we call this strategy as FaxIA. Besides, we have another strategy FaxEX, which first filters ALCs using the launching results of FaxIA. Then, it takes the activity relationships in ATG and the set of activities failing to be launched, to guide the weight assignment among ALCs. Fax supports multiple-round testing. During the exploration, it collects the execution traces for the weight calculation in the next round. Besides, the user can adopt any exploration strategy according to their requirement. The random strategy is adopted in the current version of our implementation.

5 CONTEXT CONSTRUCTION FOR TESTING

This section introduces the ALC construction process, which is based on the inter-procedural, flow-, context- and path-sensitive static analysis techniques.

5.1 String Constraint Extraction

For basic attributes, we aim to find the constraints of their candidates precisely, which are combined by the manipulation of the data receiving variable (ReceiveVar), the comparison operation (CompareOp) and the pre-defined candidate values (CandidateVar). And for the extra parameters (ExtraPara), we try to get the correct key and type of value. For both kinds, our method is mainly

```java
public class ExampleActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // ICC Message
        Intent intent = getIntent(); // get intent
        String action1 = intent.getAction(); // get action
        String action2 = intent.getStringExtra("s1"); // get extra para
        char action3 = intent.getStringExtra("f2").charAt(2); // get extra para

        if(action2.equalsIgnoreCase("act") && action3 == 'C') {
            doSomething(); // unsat path
        } else if(action1.startsWith(getPrefix("startWith", 3))) {
            Bundle b1 = intent.getBundleExtra("b1");
            String s1 = intent.getStringExtra("s1");
            Float f3 = b1.getFloat("f3");
            if(f3 != null) {
                doSomething();
            } else {
                doSomething();
            }
        }
        // ICC Message
        Intent intent = getIntent(); // get intent
        String action1 = intent.getAction(); // get action
        String action2 = intent.getStringExtra("s1"); // get extra para
        char action3 = intent.getStringExtra("f2").charAt(2); // get extra para

        if(action2.equalsIgnoreCase("act") && action3 == 'C') {
            doSomething(); // unsat path
        } else if(action1.startsWith(getPrefix("startWith", 3))) {
            Bundle b1 = intent.getBundleExtra("b1");
            String s1 = intent.getStringExtra("s1");
            Float f3 = b1.getFloat("f3");
            if(f3 != null) {
                doSomething();
            } else {
                doSomething();
            }
        }
    }
}
```

**Figure 2: Motivating Example**

<table>
<thead>
<tr>
<th>Set</th>
<th>Z3</th>
<th>Ret Value</th>
<th>Used In</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>T</td>
<td>String</td>
<td>ReceiveVar</td>
<td>append, concat, toString, substring, charAt</td>
</tr>
<tr>
<td>S2</td>
<td>F</td>
<td>String</td>
<td>CandidateVar</td>
<td>trim, equalsIgnoreCase, toUpperCase, toLowerCase</td>
</tr>
<tr>
<td>S3</td>
<td>T</td>
<td>Boolean</td>
<td>CompareOp</td>
<td>equals, startsWith, endsWith</td>
</tr>
</tbody>
</table>
Algorithm 1 demonstrates the process of intent receiving analysis. It shows how the constraint set is constructed when finding an intent-receiving statement. It accepts a method \( m \) as input, outputs a set of extra key-value pairs and constraints of attributes. In line 2, we calculate use-define chains \( \text{udc} \) of each statement in \( m \), which can be obtained by some mature analysis framework (e.g., Soot [46]). Then we locate the input data obtaining instructions (IDOIs) according to Android APIs, including \( \text{getAction()} \), \( \text{getCategories()} \), \( \text{getData()} \), \( \text{getDataString()} \) and \( \text{getType()} \) [20]. For each instruction \( \text{ins} \) in IDOIs, function \( \text{get_attr_vars} \) returns a set of variables which store the received input data. Note that the value of an attribute receiving variable can be transferred to other variables through several value-related APIs, we track the tainted variables through \( \text{udc} \) and add them into the attribute receiving variable set \( \text{vars} \). In line 6, we get a group of statements \( S_{\text{co}} \) which use the variable \( \text{var} \) for comparison in a condition (see S3 in Table 3). For each \( \text{ins}_{\text{co}} \), the input data will be compared with another string, which is a candidate string. In line 9, we get the variable \( \text{canVar} \) which stores the candidate string and try to get its value. If \( \text{canVar} \) is defined as a constant string, we can obtain the string directly in line 11. Otherwise, we recursively analyze its value which may be modified by the string operation (see S1 and S2 in Table 3). If the candidate string is obtained from the return value of other methods or from the formal parameter sent by the caller of the current method, we look into the invoked methods with current invoking context, i.e., parameters of method invocation, or query the simulated method call stack to build the correct candidate. In line 15, we use the collected information to update the set \( \text{constraints} \). The extra parameters are extracted in line 19 by function \( \text{extract_extras}() \), which will be introduced in the following subsection.

**Algorithm 1** intent_receiving_analysis

| Input: | method \( m \) |
| Output: | attribute constraint set \( \text{constraints} \), extra set \( \text{extras} \) |
| 1: \( \text{constraints} = \emptyset \), \( \text{extras} = \emptyset \) |
| 2: \( \text{udc} = \text{ge_use_def_chains}(m) \) |
| 3: for each \( \text{ins} \) in all IDOIs of \( m \) do |
| 4: \( \text{vars} = \text{get_attr_vars}(\text{ins}, \text{udc}) \) |
| 5: for \( \text{var} \) in \( \text{vars} \) do |
| 6: \( S_{\text{co}} = \text{get_compare_ins}(\text{var}, \text{udc}) \) |
| 7: for \( \text{ins}_{\text{co}} \) in \( S_{\text{co}} \) do |
| 8: \( \text{co} = \text{get_compare_operation}(\text{ins}_{\text{co}}) \) |
| 9: \( \text{canVar} = \text{get_candidate_var}(\text{ins}_{\text{co}}) \) |
| 10: if \( \text{canVar} \) can be obtained directly then |
| 11: \( \text{canValue} = \text{get_constant_value}([\text{canVar}]) \) |
| 12: else |
| 13: \( \text{canValue} = \text{get_value_of_var}(\text{canVar}, \text{udc}) \) |
| 14: end if |
| 15: \( \text{constraints} = \text{constraints} \cup \{ (\text{ins}, \text{var}, \text{co}, \text{canValue}) \} \) |
| 16: end for |
| 17: end for |
| 18: end for |
| 19: \( \text{extras} = \text{extract_extras}(m, \text{udc}) \) |
| 20: return \( \text{constraints}, \text{extras} \) |

In the example in section 3, we can get variables \( \text{vars} = \{(l5-intent), (l6-action), (l7-action.substring(1,4)), (l8-action.charAt(2))\} \), and constraints \( \text{constraints} = \{(l6, action.substring(1,4)), \text{equals}, \text{"act"}, (l10, action.charAt(2)), \text{equals}, \text{"C"}, (l13, action, startsWith, \text{"sta"})\} \).

### 5.2 Extra Parameter Analysis

The \( \text{extras} \) in Algorithm 1 contains a set of extra parameters, each of which is a key-value pair. An activity makes use of several specific system APIs of the Intent class to retrieve the data with a
specific type by a user-defined key, e.g., getExtra("s1"). Our goal is to locate the invocations of these APIs and obtain keys and types of data items for each activity. To locate the extra parameter obtaining instructions (EPOIs), we collect an API list, including 29 APIs of Intent class, 24 APIs of Bundle and 4 APIs of Parcelable, that can be used to get extra parameters with different types, according to the API Reference [8, 20]. With this list, it is easy to find out the EPOIs and determine the types of extra parameters. Note that, these extra parameters are usually used for providing values, thus their data are arbitrary. After extracting their type and key, we generate a set of data according to their types.

Algorithm 3 gives the process of how to extract extra parameters. For example, in Java file, we may have a statement getStringExtra ("s1"). In the intermediate representation format Jimple/Simple transformed by Soot, the key "s1" will be stored in an additional variable, e.g., we may have x = "s1", getsStringExtra(x). In line 3, the function get_used_var() will return the variable that stores the key value, i.e., get variable x. To get the key of extra parameters, we then invoke the function get_value_of_var() in line 4 to get the value of the given variable, i.e., get value "s1". Function get_type() in line 5 returns the type of an extra data item, which can be a basic Java type (like String type for getStringExtra()) or an encapsulated Android-specific type (like Bundle for getBundleExtra()). The algorithm of function get_type() is shown in Algorithm 3. For basic types, they can be extracted from the name of APIs. The special situation is the getBundleExtra()-like instructions, which obtain a data item with Bundle type that may consist of multiple extra parameters, i.e., key and type pairs. Thus it needs to recurce to extract these nested key and type pairs. For this case, the method get_type() first calculates the set of instructions ps that var is propagated to. With this set, we recursively get the key-type pairs attached to the Bundle type and return a bundle object as the type. At last, the tuple (key, type) will be collected and returned.

5.3 Activity Launching Model Construction

The Activity Launching Model (ALM) can be formally defined as a 5-tuple \( M = (entry, N_e, N_v, E, R) \), where

- \( entry \) is the root node of the ALM, which represents the execution entry of the activity.
- \( N_e \) is a set of basic attribute related nodes. Each node \( n \in N_e \) is a triple \((id, attr, constraint)\), where \( id \) is the identifier of the node, \( attr \) is an ICC-related basic attribute, and \( constraint \) describes the constraints about \( attr \).
- \( N_v \) is a set of extra parameter related nodes. Each node \( n \in N_v \) is a tuple \((id, para)\), where \( id \) is the identifier of the node, \( para \) contains a group of extra parameters in the form of \((key, type)\).
- \( E \) is a set of edges that link nodes on the same execution paths.
- \( R \) is a tuple \((path, res)\), where \( path \) contains a list of \( ids \) of nodes, \( n.id \in path, n \in N_v \cup N_e \), and \( res \) gives the solving results of the constraints. After generating test data for each attribute/parameter for one feasible path in \( R \), we can get an Activity Launching Context (ALC) as a test case.

Fig. 4 uses a tree to display the ALM of the motivating example, in which each path corresponds to a list of attributes or parameters in an ICC message and the leaf nodes display the constraint solving results. In this case, we get three feasible paths. One path is dropped out because the path condition is unsatisfiable.

![Figure 4: ALM of the Motivating Example](image-url)

In an ALM, each path can be transformed into a set of ALCs by giving assignments of attributes/parameters in the path. For the basic attributes, we generate assignments that satisfy the collected constraints in section 5.1. Considering that each received value from ICC should be checked by null-checkers before used, we mutate the default paths by adding null1 value as candidate value for each attribute/parameter without using an explicit null-checker. For the extra parameter, we design a set of abnormal values for extra parameters, e.g., null1 value, the boundary of corresponding extra type, extremely long string, etc.
5.4 Test Script Generation

The ALCs will be transformed into executable test cases. The adbf- form command is widely adopted owing to its simplicity and efficiency. Unfortunately, it has several limitations. First, the adbf-form invocation does not support permission requirement, i.e., the callee activity restricts its callers by requiring specific permission. Besides, the type of parameters that an adbf command can carry is limited. If a test case contains any Java object, such as Bundle or ArrayList object, it cannot be sent through adbf.

To deal with this problem, we design a dummy app to transit the launching command. First, Fax detects the required permissions of the app under test. Then, an empty Android project will be created with the required permissions. For each ALC, Fax creates an exported dummy activity who can be directly launched using adbf command. For example, "adb shell am start -n com.fax.test/dummy_id". In the onCreate() method of each dummy activity, we generate launching related code that sends an ICC message with corresponding ALC. For each parameter in an ICC message, we create objects dynamically. At first, the ATG information is constructed statically according to its type. For bundle type, we reconstruct the proper data structure. Clicking on the UI of the activity in the dummy app can also launch the activity to be tested. So that we can easily perform the test by starting the activity-form test cases in dummy apps instead of the activities in the app under test.

6 MULTIPLE-ENTRY EXPLORATION

Besides launching activities with proper ALCs, we also want to detect hidden bugs that can be triggered during the in-depth exploration through multiple entries. Therefore, we need to measure the importance of each entry and assign weights among them during testing. However, part of activities can not be successfully launched as exploration entries and the contribution of activities vary in the whole testing approach, e.g., a leaf activity which never jumps out is likely to need fewer testing events. Furthermore, both the launching result and the transition contribution are difficult to be measured out is likely to need fewer testing events. Furthermore, both the launching result and the transition contribution are difficult to be measured.

Algorithm 4 gives the process of the adaptive exploration, which starts with a coarse-grained ATG and adjusts the weights of ALCs dynamically. At first, the ATG information is constructed statically and the execution information is empty. We use function LA( lc ) to get the actually launched activity by executing ALC lc . In each round, we obtain the subview of each ALC. For crash-triggering ALCs, their weights are zero. For a crash-irrelevant ALC, we record the subview svlc as the sub-graph of the current ATG, which contains all the reachable activities starting from activity LA( lc ). Set SFlc is the set of activities that failed to be launched in svlc. For each ALC in the launching context set LCs, its weight will be recalculated in multiple rounds. The weight of lc in the ith round exploration can be calculated by formula (1):

$$
Weight(lc, i) = \frac{\theta}{Dis(LA(lc), a_j)} + \sum_{k=0}^{\gamma} N_m(a_k) + \gamma
$$

where $i > 0$, $lc \in LCs$, $0 < j \leq |SF_{lc}|$, $0 < k \leq |SV_{lc}|$. The function $Dis(LA(lc), a_j)$ evaluates the distance between the launching target activity LA(lc) and each element in SFlc. The ALC that can reach more unvisited activities or reach unvisited activities with fewer transitions will have a higher weight. We use function $N_m$ to count the number of methods contained in the activities in the subview $SV_{lc}$, which indicates the subview size of each launching target activity. The ALC whose subview reaches more methods will have a higher weight. In the ith round, we use parameter $\theta$ to balance the distance to unvisited activities as well as the contribution of the launching target. Parameter $\gamma$ is a basic constant weight, which is designed for non-leaf activities whose transitions are lost in the initial ATG. It guarantees the weights of all ALCs to be positive. After weight calculation, we use the weight ratio among all launching contexts in the set LCs to get the exploration priority by formula (2), where $1 \leq m \leq |LCs|$.

$$
Priority(lc, i) = \frac{Weight(lc, i)}{\sum_{m=1}^{Weight(lc, m, i)}}
$$

According to the exploration event number in each round, we can get the exploration event owned by each activity using formula (3).

$$
Event(lc, i) = Priority(lc, i) \times E_n(i)
$$

For example, consider the example in Fig. 1. If we have three Lcs (see Fig. 5) that can successfully launch activities: Main, Details and Setting, and the other activities failed to be launched. Suppose $\theta = 6$, $\gamma = 1$, $i = 1$ and each activity has one method, we can calculate their exploration weight as: $Weight(lc_1)=11+6=18$, $Weight(lc_2)=15+4+1=20$, $Weight(lc_3)=6+2+1=9$. $Priority(lc_1)=18/47$, $Priority(lc_2)=20/47$, $Priority(lc_3)=9/47$. If we have 470 exploration events in the first round, we have $Event(lc_1)=180$, $Event(lc_2)=200$, $Event(lc_3)=90$, rather than the assignment $Event(lc_1)=470$ in SET.
7 EVALUATION

We design a benchmark IntentBench [19] to evaluate the effectiveness of the context construction. It contains 43 activities and involves various features, e.g., branch, loop, override, inter-procedure, and intent-receiving characteristics. We show the self-checking result in Table 4, in which the first two columns give the category name and the number of activities (#A). The following columns give the results of ICC attribute identification and ALC generation. For attribute identification, we collect the number of attribute values used in each category (#Attr), the correctly extracted attributes by attribute identification, we collect the number of attribute values (#AV). For ALC generation, we check the correctness of ALCs by comparing them with all the ICC-related program paths.

Table 4: Effectiveness Checking on IntentBench

<table>
<thead>
<tr>
<th>Category</th>
<th>#A</th>
<th>ICC Attribute #TP</th>
<th>#FP</th>
<th>#FN</th>
<th>Launching Context #TP</th>
<th>#FP</th>
<th>#FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Attribute</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Extra Parameter</td>
<td>8</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Basic and Extra</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>String</td>
<td>7</td>
<td>29</td>
<td>0</td>
<td>2</td>
<td>34</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Null Checker</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Override</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lifecycle</td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>13</td>
<td>35</td>
<td>3</td>
<td>1</td>
<td>32</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Complete</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>44</td>
<td>153</td>
<td>4</td>
<td>3</td>
<td>127</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The loss of precision in the sensitivity category is due to several reasons: 1) there may be FPs when an attribute variable is compared with a field/static value, whose assignment may be wrongly obtained; 2) when the string value is operated by loop operations or obtained from unknown library functions, there will be FNs. Besides, the path-sensitive analysis for ALC generation may suffer from path-explosion, there will be FNs when the actual number of paths is beyond the threshold. In our experiments, we limit the number of paths to 100,000.

7.2 Effectiveness of Activity Launching

According to the previous works [10, 60], Monkey is one of the most popular and effective testing tools due to its effectiveness and simplicity. Although Monkey behaves well in GUI testing, we noticed that there is a model-based testing tool Ape [15] which aims to replace Monkey. And intentFuzzer [51], which sends intents with null value as well as serializable data, aims to trigger activity-launching related bugs specifically. Therefore, we compare with the baseline Monkey and the state-of-the-art GUI exploration tool Ape, as well as the fuzzing tool intentFuzzer, whose characteristics are listed in Table 5. In the following experiments, we set one hour as the testing upper limit time for all tools on each instance. Before testing, we log in apps manually according to their functional requirements.

Table 5: Characteristics of Tools

<table>
<thead>
<tr>
<th>Name</th>
<th>Target</th>
<th>Entry</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monkey</td>
<td>GUI Exploration</td>
<td>Single</td>
<td>Random</td>
</tr>
<tr>
<td>Ape</td>
<td>GUI Exploration</td>
<td>Single</td>
<td>Model-based</td>
</tr>
<tr>
<td>intentFuzzer</td>
<td>Intent Fuzzing</td>
<td>Multiple</td>
<td>None</td>
</tr>
<tr>
<td>Fax</td>
<td>Both</td>
<td>Multiple</td>
<td>Random</td>
</tr>
</tbody>
</table>

Fig. 6 gives the number of activities of each app in our benchmark as well as the coverage reached by all tools. There are 391 activities in our benchmark. After one hour of testing, Monkey covers 147 of them and Ape covers 208. As we can see, Monkey reaches high activity coverage when the number of activities in an app is small but becomes ineffective when an app has a large number of activities. Ape has a similar tendency as Monkey, but it usually reaches higher coverage than Monkey does. The tool intentFuzzer only covers 158 activities. The performance of intentFuzzer is not stable when testing all activities continuously. And Fax with strategy $\text{Fax}_{x}$ covers 377 (96.4%), which works well regardless of the size of the app. Note that, some activities crashed when launched, e.g., which...
than Monkey and Ape. The reason we inferred is that the switching of entry takes extra costs, and some ALCs bring fewer benefits than the default entry does while having a high weight.

7.4 Bug Detection Ability

During the app launching and exploration, we record the runtime log information and collect the triggered unique crashes. Totally, Fax detected 719 unique crashes, among which 655 are launching related bugs by triggering 1303 launching commands, and 64 are detected during the GUI exploration of apps. As a comparison, Monkey finds 8 crashes during exploration, Ape finds 12. IntentFuzzer finds 18 crashes by testing the original EAs in apps, and it finds 81 ones by testing all activities after exposing IAs into exported ones.

We categorize the crashes detected by Fax into Errors and Warning according to their triggering entries, error for EA and warning for IA. The details of these crashes are listed in Table 7. All the 180 crashes triggered by EA launching and triggered by an exploration starting from EAs can be taken as real errors. In these cases, anyone can make the target app crash by sending malformed commands to EA. Besides, we find 539 crashes that can be categorized as warnings. These crashes are triggered on exported IAs and may not harm the usage of the app actually. A warning means the correctness of the crash point in the callee depends on the quality of the caller activity. However, developers suffer from the misexposure of activities [55], which means they may misexpose activities unanticipatedly and make these warnings become attackable. For example, a bug fixing by the developer of EteSync is to turn the EA AccountActivity into an IA, which means there is a misexposed activity. So, we take these crashes triggered on IAs as potential bugs and warn developers earlier.

### Table 7: Category of Crashes Detected by Fax.

<table>
<thead>
<tr>
<th>Crashes</th>
<th>Entry</th>
<th>Normal</th>
<th>Object</th>
<th>Faxex</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>101 EAs</td>
<td>49</td>
<td>109</td>
<td>22</td>
<td>180</td>
</tr>
<tr>
<td>Warning</td>
<td>290 IAs</td>
<td>107</td>
<td>390</td>
<td>42</td>
<td>539</td>
</tr>
<tr>
<td>Sum</td>
<td>391 Acts</td>
<td>156</td>
<td>499</td>
<td>64</td>
<td>719</td>
</tr>
</tbody>
</table>

The distribution of exception types is shown in Fig. 7. The ClassCastException is the most common one, which means the target class could not be loaded. The BadParcelableException happens when an activity receives an unexpected object value. If

![Figure 7: Detected Unique Crash Distribution](image-url)
By analyzing the composition of the 1303 crash-triggering commands by activity launching, we count the number of ICCs without any attribute or parameter (Basic) and with only basic attribute (Basic). As we can see in Table 9, 10.1% crashes are triggered with empty ICC, while 82.8% need specific basic attribute and up to 81.7% contain extra parameters. For ICCs that contain extra parameters, we count the number of ICCs with primary parameter only (Primary), with Bundle item (Bundle) as well as with Serializable or Parcelable object item (Object). About 67.4% commands contain objects, which means object-carrying commands can easily crash an app. In our experiment, the longest crash triggering test case requires six non-null attributes, including one basic attribute and five extra parameters.

### Table 9: Type of ICC Message that Trigger Crash

<table>
<thead>
<tr>
<th>Type</th>
<th>Null</th>
<th>Basic</th>
<th>Primary</th>
<th>Bundle</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>132</td>
<td>107</td>
<td>137</td>
<td>97</td>
<td>878</td>
</tr>
<tr>
<td>Ratio</td>
<td>10.1%</td>
<td>8.2%</td>
<td>10.5%</td>
<td>7.4%</td>
<td>67.4%</td>
</tr>
</tbody>
</table>

We pick 46 crashes that can be triggered by launching EA and commit issues about them on Github. The committed bugs are picked for three reasons: 1) we only select the crashes that are triggered by launching EA; 2) we only submit the normal-type crashes, which can be triggered by test cases that do not carry complex objects, to make the bug confirming easier; 3) we exclude the apps that did not update within one year. Among the 46 reported crashes, 33 have been confirmed and fixed. The results are shown in Table 8, in which the issues without the developer’s reply are dropped out. For apps AnkiDroid and PassAndroid, we open pull requests for bug fixing according to developers’ requirements, and the developers have confirmed these fixing. Other bugs are fixed by developers, in which two bugs in app AntennaPod were fixed in a recently released version before our report. The developer of project SuntimesWidget replied that the “intent resolving” was pretty much untested before and they decided to add test cases to avoid this problem.

### 8 THREATS TO VALIDITY

**Internal validity:** There are two internal threats in our approach: false positive of IA-related bugs and weight assignment.

The first threat relates to the false positive of the bug detection on IAs. If the exported activities are taken as the testing entries, all detected activities are real bugs that can be exploited by attackers, i.e., all identified Errors are true positives. In our approach, to detect more hidden bugs, we allow Fax to take internal activities as testing objects. The testing of internal activities is more likely to be unit testing. Without analyzing ICC flows and tracking all the constraints of the input data, Fax supposes that any input received by internal activities is reasonable, which may contain invalid values. For Warnings, we will conduct further analysis to automatically get the number of true positives, e.g., make a forward tracking of each received value to figure out the data sources and constraints.

Another threat relates to the accuracy of the weight assignment. For activity launching testing, more entries can exploit more possible bugs, but in exploration, the low-quality entry will decrease the total coverage. In tool Fax, we evaluate the importance of each entry based on the dynamically constructed ATG and use heuristic strategies to filter out the entries with lower importance. The exploration weight calculation depends on the accuracy of ATG and the dynamic execution traces. Generally, it is difficult to identify all the transitions statically and judge whether the transition is available or not by pure static analysis. We complement the static ATG by dynamic exploration, however, we still can not guarantee the fairness of testing. But we make efforts to recalculate the exploration weight among ALCs by multiple rounds and try to optimize the weight assigning process adaptively.

**External validity:** Threats to external validity relate to the generalizability of our experimental results. Our study is limited to the evaluated Android apps and our results may not generalize beyond the evaluated apps.

### 9 RELATED WORKS

In this section, we will briefly introduce representative works that are related to the GUI exploration, ICC analysis and intent fuzzing techniques in recent years.

**GUI Exploration** There are many kinds of GUI testing approaches for Android apps, including random, model-based, and systematic testing techniques. Wang et al. propose a description framework to demonstrate the key issues in automatic test-input generation [53]. In random testing, the test events will be generated randomly with less care of the current state of the app under test. Monkey is one of the most widely used black-box random testing tools. It is a simple and fully automatic tool that can generate a lot of test events within a short time. There are works based on
Monetary for detecting GUI bugs [17] and security bugs [25]. Several researches construct the models to guide their exploration process [2, 3, 9, 15, 16, 28, 57]. S. Yang et al. [57] provided a model called Window Transition Graph, with an accurate static callback analysis. Su et al. [48] proposed a model-based approach recently, which uses both dynamic and static analysis with a weighted UI exploration strategy. And they randomly inject system-level events, like sending null intent, to trigger more bugs. However, they extract events according to the declaration of tag intent-filter in the manifest. Systematic testing techniques [7, 22, 27] are applied in more complicated circumstances, e.g., automatically finding event sequences that reach a given target line in the application code. Our work focuses on GUI exploration, but we do not limit to one exploration strategy. We concentrate on starting the exploration from multiple entries and with various ALCs. During the exploration, every event picking strategy can be integrated.

**Intent Fuzzing.** Some researchers adopt fuzzing technique [49] to find out the poorly designed exported components, which also need to simulate the proper ALC. For example, tool Null Intent Fuzzer [18] sends intents with the only input data null. And tool DroidFuzzer [59] focuses on activities that process MIME data (e.g., “video/*”) passed via an URL. Besides, Maji et al. [26] presents the first empirical evaluation of the robustness of ICC in Android through fuzz testing methodology. However, when fuzzing explicit intents, they use straightforward strategies, such as “Semil_valid Action and Data”, “Blank Action or Data”, “Random Action or Data” as well as “Random Extras”, which may generate a large number of redundant test cases. In its experiment, around 9000 intents will be sent to test an activity, while we use less than ten test cases in our approach. To avoid the aimless exploration with invalid parameters, these works [26, 45, 56, 59] adopt the configuration-directed testing approach. They aim at the original exported components that have an XML-formed declaration in manifest, which is provided by the Android system for app configuration. However, there are severe mismatches between the attribute declaration and their actual usage according to our study. Some of the ICC parameters can only be obtained in code but not the manifest file. Another tool IntentFuzzer [45] is developed using some static analysis techniques with the goal of triggering bugs, which is similar to our activity modeling. However, they directly leverage FlowDroid, a static analysis tool designed for privacy leak detection, to extract the key-type pairs of extra parameters. So, they can not handle large-scale Android apps. Besides, their approach has the inherent weakness from fuzzing that the number of test cases is very large, while we avoid this problem by path sensitive attribute usage analysis.

**ICC Extraction.** Some works aim at extracting ICC information, for example, the research [31] proposed COAL language to model the ICC messages and apply the COAL solver to infer Android ICC values. In this work, they implemented a practical tool called IC3. Recently, some researchers [50] conduct researches based on it. However, IC3 does not provide the attribute usage information of ICC and it is unable to generate ALCs. Besides, it obtains basic attributes from manifest files, which is not accurate enough. In our approach, we adopt a light-weight intent analysis method in this paper to obtain the information needed.

**String Analysis.** As a widely used type in Android apps, the string is also widely studied by recent works. Rasthofer et al. [44] presents a framework for automatically generating an Android execution context to trigger malicious behaviors, in which string information should be inferred correctly. To accomplish this, they give several string value providers. The constant value provider they used gathers all the string constants as candidates for runtime values which compare against constants, which will increase the burden of testing, while the dynamically-computed values are not taken into account. In our work, to find out candidate values for ICC-related attributes precisely, we capture the data propagation to obtain the actually used constant candidates, and model the ICC related string APIs to calculate the dynamic operated ones.

**Symbolic Execution** Symbolic execution is a useful program analysis technique that can simultaneously explore multiple program paths with various execution contexts. However, the analysis suffers from path divergence without simulating the behavior of Android libraries. To verify Android apps precisely, Merwe et. al. extend JPF [21] to JPF-Android [52]. They model core libraries in the Android framework semi-manually and symbolically execute apps on Java Virtual Machine. Gao et al. [14] then proposed a dynamic symbolic execution engine for Android apps, which automatically synthesizes libraries without manual modeling. Our approach also adopts a symbolic-execution-like analysis and collects path constraints about ICC attribute variables. Concentrating on the ICC attributes modeling, we do not perform analysis on complete paths but drop the ICC attributes/parameters-irrelevant information.

## 10 Conclusion

In this paper, we aim to break the uneven activity coverage in the exploration of Android apps and try to test each activity in various launching contexts. We first investigate the launching process of activity component, then perform an inter-procedural, flow-, context- and path-sensitive analysis to build activity launching models and generate complete launching contexts. Besides, we proposed an adaptive exploration framework that reassigns events to multiple entries to enhance the exploration ability. The key challenges lie in how to handle various ICC attribute characteristics to construct proper contexts as well as how to calculate the exploration weight of each entry in each round. We implemented our approach in a tool called Fax, with an activity launching strategy Fax, and an exploration strategy Fax, where the experiments on real-world apps show that Fax behaves well both in the in-depth exploration and the context-aware activity launching testing. In the future, we will try to identify the trigger paths of IA-related crashes automatically to make the bug confirmation easier.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work is supported by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), and the National Natural Science Foundation of China (Grant No. 61672505).