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
Univ. of Chinese Academy of Sciences
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, since 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 and detect more bugs, we design an adaptive exploration framework which supports the multiple-entry exploration and dynamically assigns a weight to each entry.

Our approach is implemented in a tool called *Fax*, with an activity launching strategy Fax_{la} and an exploration strategy Fax_{ex}. The experiments on 20 real-world apps show that Fax_{la} can cover 96.4% and successfully launch 60.6% activities, based on which Fax_{ex} further achieves a relatively 19.4% 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. The number of real bugs that can be triggered without any modification of the app is 180, 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.

*Corresponding Authors.
Email: yanjun@ios.ac.cn, zj@ios.ac.cn

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380347

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

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

ACM Reference Format:
Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-Entry Testing of Android Applications by Constructing Activity Launching Contexts. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380347

1 INTRODUCTION

Mobile applications, especially Android apps, have witnessed an explosive growth in recent years. For GUI testing, a great number of automatic exploration approaches have been proposed, including random [22, 44, 61], model-based [48, 58, 59] and systematic ones [32, 42], 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 these approaches as Single-Entry Testing (**SET**) ones.

In SET approaches, some obstacles, e.g., complex gesture or logical operations, make parts of 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 [22] to test a widely used app *WeChat*, and has a similar observation: Monkey allocates a lopsided distribution of exploration time on each activity [61]. Besides, because of 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` cannot be launched if its domain node, e.g., `Detail`, failed to be visited. In addition, `Account` has different behaviors depending on its Activity Launching Contexts (**ALC**), which is generated by previous event operations or received from outside (e.g., other 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.

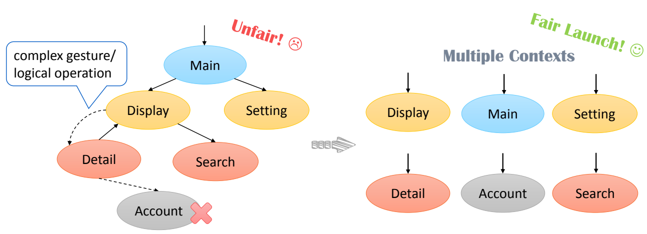

Figure 1: Single- and Multiple-Entry Testing

Usually, user can successfully launch the default entry of an app by simply clicking the icon on the launcher of phone, or sending an external command with the correct activity name. 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. For an ICC message, its launching model should contain 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 [46, 52, 57, 60] extract the attribute declarations by XML file analysis. However, according to our comparison, the attribute declaration of activity usually be 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 [13], the activity receiving data items can be categorized into two types: *Basic Attribute* and *Extra Parameter*, in which the *Extra Parameter* can be further separated 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 in the ALC; and for a bundle extra parameter, which is a nestable key-value map, we should reconstruct its original data structure. We need to consider all of these characteristics to construct proper ALCs as much as possible.

The second challenge is with respect to the weight assignment among entries in MET. Different from SET approaches, which always starts testing from the default entry, the MET requires us to make decisions on the exploration weight calculation of each entry. On the one hand, part of activities failed to be launched by the constructed ALCs. 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. A 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 required ALC-related attributes in each activity. Based on the ALM, we generate ALCs, each of which 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 lie in three-folds:

- **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_{la} and the exploration strategy Fax_{ex}.
- **Tool Implementation.** We implement our approach in the tool *Fax*, which can launch marginal activities easily and aims to test activities under various ALCs. The experimental results indicate that Fax has strong bug detection ability and reaches high code coverage.

2 BACKGROUND

This section provides the 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 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 [17]. 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 sequence.

2.2 Activity Launching Contexts

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 mobile phone, e.g, wifi, GPS status, which depends on the setting of phone.
- **Activity Stack.** The activity stack stores the history activities visited before reaching 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 much program paths as possible using these sub-contexts. For the ICC message, we can model their usage with the help of static analysis to achieve backward ALC generation. For device configuration, we scan the corresponding APIs and analyze the related configuration items of an app. For stack context, an activity can be tested under empty stack with Fax_{la} and under non-empty stack with Fax_{ex}. The global data is difficult to be controlled, e.g., get the number of file on the sdcard. However, if the global data can be modified during exploration, its value can help to trigger more program paths in the multiple-round testing. For example, if the Setting Activity is explored in multiple rounds, the operations on it will create different contexts for other activities.

Table 1: Composition of ICC Message

Type	SubType	Loc_d	Loc_u	Type
Basic	Action	Xml	Java	String
	Category	Xml	Java	Set ⟨String⟩
	Data	Xml	Java	String
	Type	Xml	Java	String
Extra	Primary	–	Java	$\langle k, v \rangle$ pair, k is in String type, v is a sub-item in type of Java Primitive data types, String, Array, ArrayList etc. [13]
	Object	–	Java	$\langle k, v \rangle$ pair, k is in String type, v is a Serializable/Parcelable object or a set of objects. [13]
	Bundle	–	Java	set of $\langle k, v \rangle$, each of which can be a primary extra, an object extra or a nest bundle extra. [6]

2.3 ICC Message and Intent Receiving

ICC message is one of the most important sub-context, which has complex composition and is necessary for the activity launching. Table 1 lists the required attributes in ICC invocation and gives their characteristics, in which columns Loc_d and Loc_u present the declaration and the usage locations of these attributes. 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 be 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* [20], which has over 4000 stars on Github, requests three actions in Java code, in which only one value is declared in the manifest. It declares two values in the manifest file, but one is invalid and not used in the Java code as well.

We further perform the consistency detection between the declaration and usage of 1200 apps collected from F-Droid [9]. The results are listed in Table 2, including the statistic result of the number of declared (Dec) and used (Use) attribute values, as well as how many of them are consistent (Con). Note that the value of data is a regular expression but not a specific string, so we do not give the consistency of data attribute. As we can see, there are huge inconsistencies. The key reason is that the intent-filters are designed for implicit invocation. Developers can declare multiple attribute values in intent-filters to characterize one activity but do not use them in the source code. The attribute they actually used may not be related to implicit invocation and do not be declared. Therefore, only collecting the declared values of ICC-related attributes in the manifest is not sufficient for activity modeling.

Table 2: Consistency of Basic Attributes of 1200 Apps Collected from F-Droid

	Declaration	Usage	Consistent
Action	2670	777	445
Category	2430	0	0
Data	2772	361	–
Type	727	81	36

Besides *Basic Attribute*, ICC messages also accept *Extra Parameters*. Each extra parameter is a key-value pair $\langle k, v \rangle$, we further separate it 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 [6, 13], 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 `ArrayList`, etc. For example, the API `getIntExra(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 object 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 intent receiving and activity modeling, 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, this activity gets the network connection information by invoking API `getSystemService`, which belongs to device configuration sub-context and is modeled as a configurable attribute. If the network is connected, the activity will get an intent object through the API `getIntent()` and create an Intent variable 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., `getAction()`, `getIntExtra()`, 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 as a condition of branch picking through comparison, there should be a candidate value of this attribute that can influence the program's execution. For example, for the statement `if(mAction.equals("ACTION_VIEW"))`, we can get 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 line 5-10 show, the new constraints are `getAction().substring(1,4).equals ("act")` and `getAction().charAt(2)=='C'`. On the other hand, the candidate values may be manipulated through a set of String-related APIs before value comparison. In line 13, for instance,

```
1  public class ExampleActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle savedInstanceState) {
4      // ICC Message
5      Intent intent = getIntent(); //get intent
6      String action1 = intent.getAction(); //get action
7      String action2 = action1.substring(1,4);
8      char action3 = action1.charAt(2);
9      if(action2.equals("act")){
10       if(action3 == 'C'){
11         doSomething(); //unsat path
12       }
13     }else if(action1.startsWith(getPrefix("startWith", 3))){
14       Bundle b1 = intent.getBundleExtra("b1");
15       String s1 = intent.getStringExtra("s1");
16       Float f3 = b1.getBundle("b2").getFloat("f3");
17       if(f3!=null) {
18         doSomething();
19       }
20     }
21   }
22   private String getPrefix(String str, int i) {
23     String newStr = str.substring(0, i);
24     return newStr; //String operation
25   }
26 }
```

Figure 2: Motivating Example

the candidate value `"sta"` is related to both the value that comes from formal parameters and the semantics of API `getsubstring()`. In addition to string analysis, the data structure of the received ICC message should be reconstructed in some cases. As shown in line 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-float`), which brings difficulties to both the ICC message generation and test case (in app-form) generation.

4 FRAMEWORK OVERVIEW

In Fig. 3, we give the framework overview of Fax, which takes an apk file as input and outputs a group of test 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 Fax_{la}. Besides, we have another strategy Fax_{ex}, which first filters the available ALCs using the launching results of Fax_{la}. Then, it takes the activity relationships in ATG and the set of activities failed to be launched, to guide the weight assignment among available 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, e.g., 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 is combined by the manipulation on the data receiving variable (**ReceiveVar**), the comparison operation (**CompareOp**) and the pre-defined constant values (**ConstantVal**). And for the extra parameters (**ExtraPara**), we try to get its correct key and type of value. For both kinds, our method is mainly

Table 3: Information of String-related APIs

Set	Z3	Ret Value	Used In	API
S1	T	String /Char	ReceiveVar CompareVar ExtraPara	append, concat, substring, charAt, toString
S2	F	String	ConstantVal ExtraPara	trim, toUpperCase toLowerCase, equalsIgnoreCases
S3	T	Boolean	CompareOp	==, !=, isEmpty, startsWith, endsWith, contains, equals

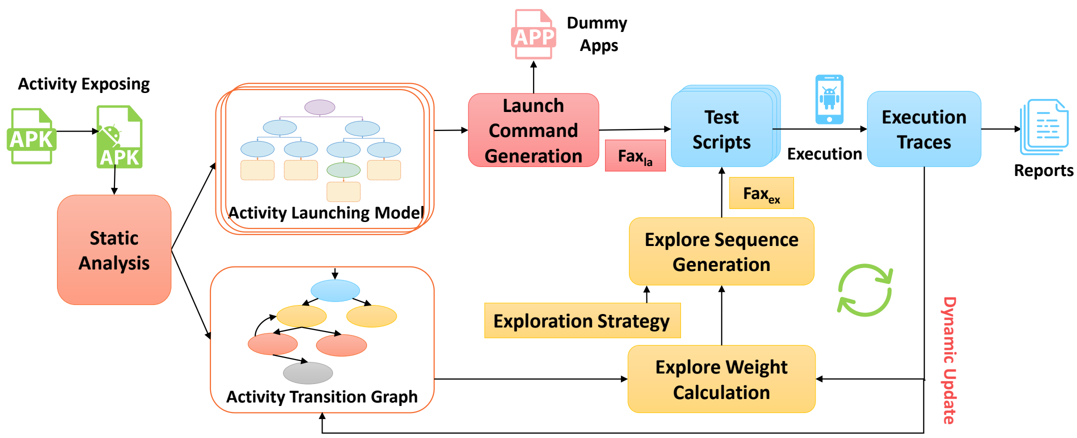

Figure 3: Overview of the Multiple-Entry Testing framework, which is implemented in the tool Fax

based on reaching definition technique [25], which is a commonly used data-flow analysis. It statically determines which definitions may reach a given point in the code, and can help us to construct use-define chains [29] to capture data propagation.

Both the candidate value of basic attribute and the key of the extra parameter are related to the type of `String`. In Java 8, there are totally 67 string related APIs [19], including *value-related APIs* that return a new string or char, *compare-related APIs* that return a boolean flag, and *info-related APIs*, such as `length()`, etc. In our work, we only concentrate on the first two types. In Table 3, we classify string-related APIs into three classes according to their return type and whether they are supported by string constraint solver *Z3* [37, 62]. In the third column, we show where these APIs are used. Note that operator "+" for `string` will be transferred to the invocation of append in the intermediate representation of Java.

Algorithm 1 shows how the constraint set is constructed when finding an intent-receiving statement. It accepts a method m, outputs a set of extra keys and constraints of attributes. In line 1, we calculate use-define chains udc of each statement in m, which can be obtained by some mature analysis framework (*e.g.*, Soot [26]). Then we locate the input data obtaining instructions (*IDOIs*) according to five Android APIs, including `getAction`, `getCategories`, `getData`, `getDataString` and `getType` [13]. For each instruction *ins* in IDOIs, function *get_key_vars* returns a set of variables which stores 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 *udc* and add them into the attribute receiving variable set *vars*. In line 5, we get a group of statements S_{co} which use the variable *var* for comparison in a condition (see S3 in Table 3). For each ins_{co}, the input data will be compared with another string, which is a constant string. In line 8, we get the variable *can_var* which stores the candidate string and try to get its value. If *can_var* is defined as a constant string, we can obtain the string directly. Otherwise, we recursively analyze its value which may be modified by the string operation (see S1 and S2 in table 3). If the constant candidate is obtained from the return value of other methods or from the formal parameter sending by the caller of 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 14, we use the collected information to update the set *constraints*. The extra parameters are extracted in line 17 and updated to set *extras*, which will be introduced in the following subsection.

Algorithm 1 intent_receiving_analysis

Input: Method m
Output: constraint set *constraints*, extra set *extras*

1: $constraints = \emptyset$, $extras = \emptyset$
2: udc = ge_use_def_chains(m)
3: **for** each ins in all IDOIs of m **do**
4: $vars$ = get_key_vars(ins,udc)
5: **for** var in $vars$ **do**
6: S_{co} = get_co_ins(var, udc)
7: **for** ins_{co} in S_{co} **do**
8: co = get_compare_operation(ins_{co})
9: can_var = get_used_var(ins_{co})
10: **if** can_var can be obtained directly **then**
11: $can\_value = can\_var.value$
12: **else**
13: can_value = get_can_value(can_var, udc)
14: **end if**
15: $constraints = constraints \cup \{\langle ins, var, co, can\_value\rangle\}$
16: **end for**
17: **end for**
18: $extras = extras \cup \{(m, udc)\}$
19: **end for**

In the motivation example, we can get $vars$ = {(l_{10}-intent), (l_{11}-action), (l_{12}-action.substring(1,4)), (l_{13}-action.charAt(2))}, and *constr–aints* = {(l_{14}, action.substring(1,4), equals, "act"), (l_{15}, action.charAt(2), ==, 'C'), (l_{18}, action, startsWith, "sta")}.

5.2 Extra Parameter Analysis

The extra parameter 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

Algorithm 2 extract_extras

Input: Method m, use define chain udc
Output: extra information set S_{ex}
1: $S_{ex} = \varnothing$
2: **for** each ins in all EPOIs of c **do**
3: var = get_used_var(ins)
4: key = get_value_of_var(var, udc)
5: $type$ = get_type(ins, var, udc)
6: update $\langle loc, type, key\rangle$ to S_{ex}
7: **end for**

Algorithm 3 get_type

Input: EPOI ins, variable var, chain udc
Output: type bt
if ins is in Basic Type **then**
  bt = get_basic_type(ins)
else if ins is in Bundle Type **then**
  ps = get_propagation_set(var, udc)
  for each $bins$ in ps **do**
    bi_var = get_used_var($bins$)
    bi_key = get_value_of_var(bi_var, udc)
    bi_type = get_type($bins$, bi_var, udc)
    add $\langle bi\_key, bi\_type\rangle$ to bt
  end for
end if

type by the a user-defined key, *e.g.*, `getStringExtra("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 [6, 13]. 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 value providing, thus their data are arbitrary. After extracting their type and key, we generate a set of data according to their types.

Algorithm 2 gives the detailed process of how to extract extra parameters. To get the key of extra parameters, we invoke the function *get_value_of_var* in line 4 which returns the value of the given variable. Function *get_type* in line 5 returns the type of an extra-date item, which can be a basic Java type or an encapsulated Android-specific types (like `Bundle`). For basic types, they can be extracted from the name of APIs, for instance, the type of `getStringExtra` is `String`. The special situation is the `getBundleExtra` instruction, which obtains a data item with `Bundle` type that may consist of multiple extra parameters, *i.e.*, key and type pairs. Thus it needs to recurse to extract these nested key and type pairs. For this occasion, 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 ⟨*loc*, *type*, *key*⟩ will be collected and returned.

5.3 Activity Launching Model Construction

To find out the effective combination of attribute values, we adopt path-sensitive analysis to get the attribute constraints in each path

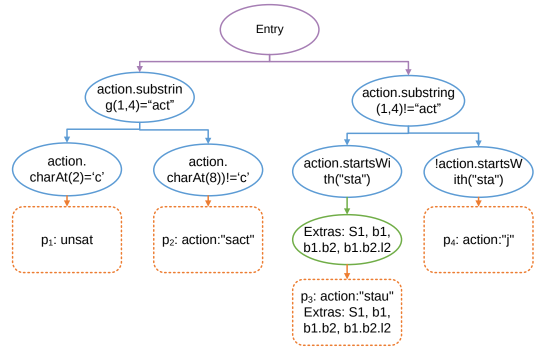

Figure 4: ALM of the Motivating Example

and build ALM for each activity. Fig. 4 gives the ALM of the motivating example, each path in which corresponding to an ICC message and the leaf nodes store the result of the constraint solving for each path. In this case, we get three feasible complete paths, i.e, three test cases. One path is dropped out because the path condition is unsatisfiable.

5.4 Test Script Generation

To generate activity launching test cases, we first construct ICC messages based on ALM. In an ALM, each path is a test case that involves a set of attribute assignments. We call them the default paths. By path-sensitive analysis, the number of default paths depends on the number of the judgment statement in code. Each received value from outside should be checked by null-checkers. For the default path without explicit null-checker, we automatically create new paths by adding null-checking branch nodes. Besides, for the basic attribute, their assignments are always used in branch statements, but for the extra parameter, their assignments vary. Therefore, we design a set of abnormal values for extra parameters, e.g., null value, the boundary of corresponding extra type, extremely long string, etc. Then we transform the ICC messages into executable test cases. Most existing works generate test cases in the form of adb command, which is provided by Android itself. The adb-form command is widely adopted owing to its simplicity and efficiency. Unfortunately, it does not suits all the occasions. Sometimes, activities will restrict their callers by requiring specific permission, that is not supported by adb invocation. Besides, some parameters in an invocation contain Java objects, such as `bundle` and `ArrayList` object, that is also cannot be carried in an adb command.

To deal with this problem, we design dummy app to transit launching command. First, Fax detect the required permissions of the app under test. Then, an empty Android project will be created with required permissions. For each ALC, Fax create an exported dummy activity who can be directly launched using adb 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 send an ICC message with corresponding ALC. For each parameter in an ICC message, we create objects according to its type, in which for bundle type we reconstruct the proper data structure. Clicking on the UI of the activity can also launch the activity to be tested. So that we can easily perform test by starting the activity-form test cases in dummy apps.

6 ADAPTIVE MULTIPLE-ENTRY EXPLORATION

As mentioned before, we test activities by generating proper ALCs. Beside launching activities with designed ALCs, we also want to detect hidden bugs that can be triggered during the in-depth exploration under these ALCs. Therefore, we need to measure the importance of each ALC and assign weights among them during testing. However, not all of the generated ALCs can effectively launch the target activity, i.e., some activities cannot be successfully launched as an exploration entry. And the contribution of activities vary in the whole testing approach, e.g., a leaf activity which never jump out is likely to need fewer testing events. Furthermore, both the launching status and the transition contribution are difficult to be obtained precisely by pure static analysis. In the strategy Fax_{ex}, we combine the static model as well as the dynamic execution results to perform an adaptive exploration.

Algorithm 4 adaptive_exploration

```
Input: application app, activity launching context set lcs
i = 1
execution_info = ∅
atg_1 = getSATG(app)
while not timeout do
    atg_i = get_ATG(app, execution_info)
    for each lc in lcs do
        LA(lc) = launching target of lc
        sv_lc = subview of LA(lc) in atg_i
        SF_lc = activities in sv_lc that are failed to be launched
        weight_lc = get_weight(LA(lc), sv_lc, SF_lc)
    end for
    for each lc in lcs do
        calculate the priority and event number of lc
    end for
    perform testing in the i^th round
    update the activity launching results into execution_info
    update the execution traces into execution_info
    i = i + 1
end while
```

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 subviews are empty. For a crash-irrelevant ALC, we record the subview sv_{lc} as the sub-graph of the current ATG, which contains all the reachable activities starts from activity $LA(lc)$. Set SF_{lc} is the set of activities that failed to be launched in sv_{lc}. For each ALC in the launching context set *LCs*, its weight will be recalculated in multiple rounds. The weight of *lc* in the i^{th} round exploration can be calculated by formula 1:

$$Weight(lc, i) = \sum \frac{\theta}{Dis(LA(lc), a_j)} + \sum N_m(a_k) + \gamma \qquad (1)$$

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 SF_{lc}. 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 i^{th} $(i > 0)$ round exploration, set SF_{lc} and subview sv_{lc} are updated by the dynamic transition information in the previous *i-1* rounds. We use parameter θ to balance the distance to unvisited activities as well as the contribution of the launching target. Parameter γ 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 $1 \leq m \leq |LCs|$.

$$Priority(lc, i) = \frac{Weight(lc, i)}{\sum Weight(lc_m, i)} \qquad (2)$$

According to the exploration event number in each turn, we can get the exploration event owned by each activity using formula 3. We use function $E_n(i)$ to denote the number of event in the i^{th} round. And in our tool, the number of total events will increase with the refinement of exploration model in multiple rounds.

$$Event(lc, i) = Priority(lc, i) \times E_n(i) \qquad (3)$$

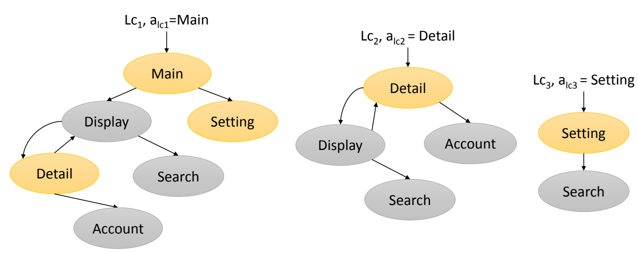

Figure 5: Subviews from Different Entries

For example, consider the example in Fig. 1. If we have three lcs (see Fig. 5) that can successfully launch activities: `Main`, `Detail` 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(lc1)=11+6+1=18, Weight(lc2)=15+4+1=20, Weight(lc3)=6+2+1=9. Priority(lc1) =18/47, Priority(lc2)=20/47, Priority(lc3)=9/47. If we have 470 exploration events in the first round, we have Event(lc1)=180, Event(lc2)=200, Event(lc3)=90, rather than the assignment Event(lc1)=470 in SET.

7 EVALUATION

We implement our approach in tool Fax [10] (**F**air **A**ndroid e**X**plorer). As shown in Fig. 3, Fax contains two strategies: the activity launching strategy Fax_{la} and the adaptive exploration strategy Fax_{ex}. In the preprocess part, we adopt the decompilation tool *ApkTool* [5] and the instrumentation tool *InsDal* [43] for activity exposing and coverage statistic. The static analysis part is built on top of the data-flow framework *Soot* [26] and *Androlic* [50] to construct ALM and ATG. We use Android *ADB* [1] to install apks and make use of the build tool *Ant* [3] to build app-form test cases.

In this section, we collected 20 popular open-source apps from Github to evaluate the effectiveness of our tool. All of our analysis processes are performed on an Intel Core i7-3770 CPU @3.40 GHz machine, with 16 GB memory and Windows 7 operating system, as well as a mobile phone (Samsung S7) in the version of Android 8.0. On our benchmark, the static analysis and the test case generation modules take 6540 seconds in total. The generated 20 dummy apps containing 2185 launching commands. Each launching command is an exported activity that can be invoked directly by ADB. Our evaluation aims to address the following four research questions:

- **RQ1 (Context Construction):** What is the effectiveness of the activity launching context construction?
- **RQ2 (Activity Launching):** What is the effectiveness of the activity launching ability of Fax?
- **RQ3 (App Exploration):** Can the event reassignment mechanism of Fax help to improve code coverage?
- **RQ4 (Crash Detection):** Can Fax find more real bugs by supporting multiple-entry testing?

7.1 Effectiveness of Context Construction

To evaluate the effectiveness of the context construction, we designed a benchmark *IntentBench* [15] by ourselves. 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 Fax (#TP), the misreported ones (#FP), e.g., give the wrong candidate of *action*, and the lost ones (#FN), lost one candidate of *action*. 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

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

In the category of *override*, the impreciseness is caused by the over-approximation, which can be avoided by path-sensitive analysis in the second phase. The loss of precision in the *sensitivity* category is due to several reasons: 1) there may be FPs when an attribute variable 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 [36, 61], 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 [39] which aims to replace Monkey. And intentFuzzer [16], 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 functionality requirements.

Table 5: Characteristics of Tools

Name	Target	Entry	Strategy
Monkey	GUI Exploration	Single	Random
Ape	GUI Exploration	Single	Model-based
IntentFuzzer	Intent Fuzzing	Multiple	None
Faxc	Both	Multiple	Random

Fig. 6 gives the number of activities of each app in our benchmark as well as the coverage reached by all tools. There are totally 391 activities in our benchmark. After one hour testing, Monkey covered 147 of them and Ape covered 208. As we can see, Monkey reaches high activity coverage when the number of activities in an app is small but become ineffective when an app has a large number of activity. Ape has similar tendency as Monkey, but it usually reaches higher coverage than Monkey does. The tool IntentFuzzer only covered 158 activities (40.4%). The performance of IntentFuzzer is not stable, due to its improper implementation. And Fax with strategy Fax_{ex} covered 377 (96.4%), which works well regardless of the size of app. Note that, some activities failing to be launched, e.g., while can be counted as covered activities, but they are ineffective for app exploration. Thus, we also count the successfully launched ones with strategy Fax_{la} by detecting whether the activity on display is the same as the launch target, and compare the result with IntentFuzzer. Under the strategy Fax_{la}, we can successfully launch 237 activities, which reaches 60.6%.

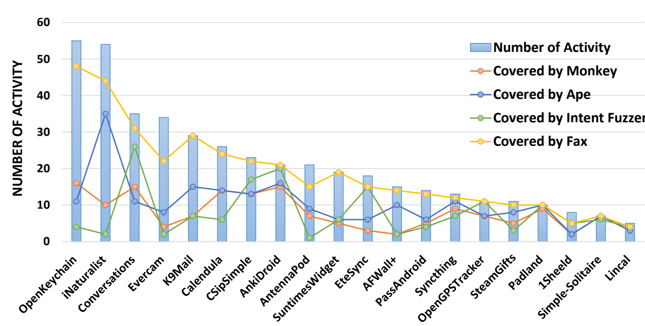

Figure 6: Activity Launching Comparison

7.3 Effectiveness of Exploration

We use method coverage to show the exploration effectiveness of each application. Table 6 displays the experimental results, in which the second column gives the total number of methods. The method coverage of Monkey, Ape, IntentFuzzer (IF for short) and Fax are shown in the following four columns. The tool which achieves the highest coverage among others on each app is labeled with green.

Table 6: Method Coverage of Monkey and Fax

App	#Method	Monkey	Ape	IF	Fax
1Sheeld	3948	22.11	30.17	6.46	25.94
AFWall+	1578	0.82	3.17	0.25	3.23
AnkiDroid	2133	50.16	61.13	23.91	54.95
AntennaPod	3599	4.67	5.20	0.03	4.56
Calendula	3277	2.32	2.41	0.40	3.02
Conversations	5088	1.57	1.04	1.43	2.20
CSipSimple	3540	28.39	26.92	13.14	31.47
EteSync	2013	0.45	0.70	1.09	1.14
Evercam	1709	0.53	1.40	0.06	2.93
iNaturalist	3306	1.94	6.00	0.18	5.99
K9Mail	6733	39.91	45.79	4.11	46.35
Lincal	325	34.77	37.54	14.46	38.46
OpenGPSTracker	899	42.94	43.83	34.37	52.73
OpenKeychain	7146	1.40	1.41	0.11	2.40
Padland	448	7.59	6.92	4.91	8.03
PassAndroid	881	47.79	50.28	14.30	38.80
Simple-Solitaire	1396	5.52	5.23	3.03	5.59
SteamGifts	1451	20.95	54.03	3.79	53.96
SuntimesWidget	3401	58.37	68.42	1.36	64.04
Syncthing	1074	3.82	4.75	1.40	4.38
Average	656.75	18.80	22.81	6.44	22.44

As we can see, Fax outperforms Monkey on 18 of 20 apps and achieves the highest coverage in 12 apps, which shows it can explore apps effectively, it achieves a relatively 19.4% high coverage compared with Monkey. For some apps, such as *EverCam* and *iNaturalist*, Fax can visit an order of magnitude greater number of methods than using one-hour Monkey testing. Overall, Ape achieves the highest coverage and Fax reaches slightly lower coverage. The root cause is the differences in their exploration strategy. We can further try to adopt the model-based strategy to improve the code coverage of Fax. Fax achieves lower coverage on *AntennaPod* and *PassAndroid* than Monkey and Ape. The reason we inferred is that some ALCs in them have high priority by static ATG analysis, but some activity transitions cannot be triggered in testing, i.e., they 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 745 unique crashes, among which 655 are launching related bugs by triggering 1303 launching commands, and 64 can be 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 20 apps, and it finds 81 ones by testing all activities after exposing.

We categorize the crashes detected by Fax into **Errors** and **Warning** according to their triggering entries, error for EA and warning for IA. The detail 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 attack the target app to 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 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 [56], 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 round 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. For Errors, EAs are taken as launching and exploration entries. For Warnings, we take IAs as launching and exploration entries.

Crashes	Entry	Launching		Explore	Sum
		Normal	Object		
Error	101 EAs	49	109	22	180
Warning	290 IAs	107	390	42	539
Sum	391 Acts	156	499	64	745

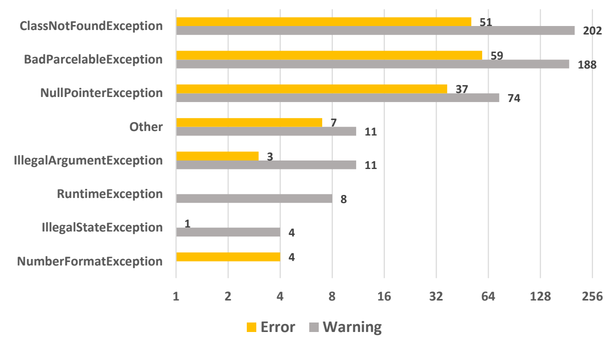

Figure 7: Detected Unique Crash Distribution

The distribution of exception types is shown in Fig. 7. The `ClassNotFoundException` is the most common one, which usually happens when the target activity received an object type parameter. The `BadParcelableException` happens when an activity received an unexpected object value. If the carried object cannot be resolved by the callee, the target activity will crash. However, this type of bug is hardly noticed by developers when they expose activities publicly. Another top exception is `NullPointerException`, which means the absence of empty input checking occurs frequently in developing.

By analyzing the composition of the 1303 crash-triggering commands by activity launching, we statistic the number of ICCs with

Table 8: Type of ICC Message that Trigger Crash

Type	Null	Basic	Primary	Bundle	Object
Number	132	107	1064	97	878
Ratio	10.1%	8.2%	81.7%	7.4%	67.4%

Table 9: Feedbacks of Issues about Activity Launching Crashes.

Project	#Star	Size	#Bug	Issue ID	Status	Fixing Revision	Reason
AnkiDroid	2025	8M	2	5401 [2]	Fixed	2c890c1	Inadequate Check
AntennaPod	2429	7M	2	3304 [4]	Fixed	f5956bc	Misexpose
Conversations	3291	10M	1	3512 [7]	Fixed	011bdd8	Inadequate Check
EteSync	96	4M	1	84 [8]	Fixed	d1d0865	Inadequate Check
iNaturalist	54	21M	1	684 [12]	Fixed	cc4a44e	Inadequate Check
K9Mail	4512	4M	3	4160 [21]	Fixed	4886f2f, 4815a2f, 16535af	Inadequate Check
Padland	33	2M	3	54 [23]	Fixed	d9709eb	Inadequate Check
PassAndroid	481	3M	4	228 [24]	Fixed	b81f79d	Inadequate Check
SuntimesWidget	68	6M	14	353 [27]	Fixed	4a6d761, 2efe94a	Inadequate Check
Synthing	1158	26M	2	1382 [28]	Fixed	c762c18	Inadequate Check

no attribute or no parameter (Null) and with not null basic attribute (Basic). For ICCs that contains extra parameters, we also count the number of which with primary parameter only (Primary), with bundle item (with Bundle) as well as with serializable or parcelable object item (Object). Note that, multiple test cases may trigger the same crash. As we can see in Table 8, 10.1% crashes are triggered with empty ICC, while 8.2% need specific basic attribute and up to 81.7% contain extra parameters. 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 attributes, including one basic attribute and five extra parameters.

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. In the selection phase, we exclude the apps that did not update within one year; we only select the crashes that are triggered by launching EA; and we only submit the normal-type crashes, which can be triggered by test cases that do not carry complex objects, to make a quick confirming. Among the 46 reported crashes, 33 have been confirmed and fixed, in which two bugs in app *AntennaPod* were fixed in a recent released version before our report. The results are shown in Table 9, in which the issues without developer's reply are dropped out. The developer of project *SuntimesWidget* replied that the "intent resolving" was pretty much untested before and they decide to add test cases to avoid this problem. And one developer declared that the caller is responsible for the correctness of the invocation. They ask us to test on the latest version and have not reply to the new submission until now.

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, it is allowed by 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 cannot 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. In random testing, the test events will be generated randomly with less care of 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 great deal of test events within a short time. There are works based on Monkey for detecting GUI bugs [41] and security bugs [45]. Several researches construct the models to guide their exploration process [30, 31, 35, 39, 40, 48, 58]. S. Yang et al. [58] provided a model called Window Transition Graph, with an accurate static callback analysis. Su et al. [53] proposed a model-based approach recently, they use 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 manifest. Systematic testing techniques [34, 42, 47] 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, any event picking strategy can be integrated.

Intent Fuzzing. Some researchers adopt fuzzing technique [11] to find out the poorly designed exported components, which also need to simulate the proper ALC. For example, tool *Null Intent Fuzzer* [14] sends intents with the only input data `null`. And tool *DroidFuzzer* [60] focuses on activities that process `MIME` data (*e.g.*, `"video/*"`) passed via an `URI`. Besides, Maji *et al.* [46] present 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 "Semi-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 [46, 52, 57, 60] 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 [52] 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 cannot 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 [49] 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 [54] 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.* [51] 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 form path divergence without simulating the behavior of Android libraries. To verify Android apps precisely, Merwe et al. extend JPF [18] to JPF-Android [55]. They model core libraries in the Android framework semi-manually and symbolically execute apps on Java Virtual Machine. Gao et al. [38] then proposed dynamic symbolic execution engine for Android apps, which automatically synthesis libraries without manual modeling. Our approach also adopt a symbolic-execution-like analysis and collect path constrains about ICC attribute variables. Concentrating on the ICC attributes modeling, we do not perform analysis on complete paths but the ICC attributes related paths. Therefore, we do not perform precise analysis on all libraries in a complete path but only model the libraries about ICCs and Strings.

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).

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_{la} and an exploration strategy Fax_{ex}. 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.

REFERENCES

[1] 2019. ADB shell - Android ADB Commands Manual. http://adbshell.com/. (2019).
[2] 2019. AnkiDroid. https://github.com/ankidroid/Anki-Android/issues/5401. (2019).
[3] 2019. Ant. https://ant.apache.org/. (2019).
[4] 2019. AntennaPod. https://github.com/AntennaPod/AntennaPod/issues/3304. (2019).
[5] 2019. Apktool - A tool for reverse engineering. http://ibotpeaches.github.io/Apktool/. (2019).
[6] 2019. Bundle | Android Developers. https://developer.android.com/reference/android/os/Bundle.html. (2019).
[7] 2019. Conversations. https://github.com/siacs/Conversations/issues/3512. (2019).
[8] 2019. EteSync. https://github.com/etesync/android/issues/84. (2019).
[9] 2019. F-Droid. https://f-droid.org/. (2019).
[10] 2019. Fax. https://github.com/hanada31/Fax. (2019).
[11] 2019. fuzzing. https://en.wikipedia.org/wiki/Fuzzing. (2019).
[12] 2019. iNaturalist. https://github.com/inaturalist/iNaturalistAndroid/issues/684. (2019).
[13] 2019. Intent | Android Developers. https://developer.android.com/reference/android/content/Intent.html. (2019).
[14] 2019. Intent Fuzzer. https://www.nccgroup.trust/us/our-research/intent-fuzzer/. (2019).
[15] 2019. IntentBench. https://github.com/hanada31/Fax/tree/master/IntentBench. (2019).
[16] 2019. IntentFuzzer Tool. https://github.com/MindMac/IntentFuzzer. (2019).
[17] 2019. Intents and Intent Filters | Android Developers. https://developer.android.com/guide/components/intents-filters.html. (2019).
[18] 2019. Java Path Finder. http://javapathfinder.sourceforge.net/. (2019).
[19] 2019. Java String. https://docs.oracle.com/javase/8/docs/api/java/lang/String.html. (2019).
[20] 2019. k9mail. https://github.com/k9mail/k-9/tree/GH-701_fix_special_use_folders_with_prefix. (2019).
[21] 2019. K9Mail. https://github.com/k9mail/k-9/issues/4160. (2019).
[22] 2019. Monkey. https://developer.android.com/studio/test/monkey. (2019).
[23] 2019. Padland. https://github.com/mikifus/padland/issues/54. (2019).
[24] 2019. PassAndroid. https://github.com/ligi/PassAndroid/issues/228. (2019).
[25] 2019. Reaching definition | Wikipedia. https://en.wikipedia.org/wiki/Reaching_definition. (2019).
[26] 2019. Soot. http://www.bodden.de/2008/09/22/soot-intra. (2019).
[27] 2019. SuntimesWidget. https://github.com/forrestguice/SuntimesWidget/issues/353. (2019).
[28] 2019. synthing. https://github.com/syncthing/syncthing-android/issues/1382. (2019).
[29] 2019. Use-define chain - Wikipedia. https://en.wikipedia.org/wiki/Use-define_chain. (2019).
[30] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *ASE 2012*. 258–261.
[31] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59.
[32] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *SIGSOFT/FSE 2012*. 59.
[33] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014*. 29.
[34] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA 2013, part of SPLASH 2013*. 641–660.
[35] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA 2013, part of SPLASH 2013*. 623–640.
[36] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *ASE 2015*. 429–440.
[37] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *ETAPS 2008*. 337–340.
[38] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 419–429.
[39] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *ICSE 2019*. 269–280.
[40] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys 2014*. 204–217.
[41] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *AST 2011*. 77–83.
[42] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *ISSTA 2013*. 67–77.
[43] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. 2017. InsDal: A Safe and Extensible Instrumentation Tool on Dalvik Byte-code for Android Applications. In *SANER 2017*.
[44] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *ESEC/FSE 2013*. 224–234.
[45] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. 2012. A whitebox approach for automated security testing of Android applications on the cloud. In *AST 2012*. 22–28.
[46] Amiya Kumar Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. 2012. An empirical study of the robustness of Inter-component Communication in Android. In *DSN 2012*. 1–12.
[47] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA, 2016*. 94–105.
[48] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *ICSE 2016*. 559–570.
[49] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *ICSE 2015*. 77–88.
[50] Linjie Pan, Baoquan Cui, Jiwei Yan, Xutong Ma, Jun Yan, and Jian Zhang. 2019. Androlic: an extensible flow, context, object, field, and path-sensitive static analysis framework for Android. In *ISSTA 2019*. 394–397.
[51] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making malory behave maliciously: targeted fuzzing of android execution environments. In *ICSE 2017*. 300–311.
[52] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *WODA+PERTEA 2014*. 1–5.
[53] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, 2017*. 245–256.
[54] Cong Tian, Congli Xia, and Zhenhua Duan. 2018. Android inter-component communication analysis with intent revision. In *ICSE 2018*. 254–255.
[55] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.
[56] Jiwei Yan, Xi Deng, Ping Wang, Tianyong Wu, Jun Yan, and Jian Zhang. 2018. Characterizing and identifying misexposed activities in Android applications. In *ASE 2018*. 691–701.
[57] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Hai-Xin Duan. 2014. IntentFuzzer: detecting capability leaks of android applications. In *ASIA CCS 2014*. 531–536.
[58] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *ASE 2015*. 658–668.
[59] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *ETAPS 2013*. 250–265.
[60] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *MoMM 2013*. 68.
[61] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *FSE 2016*. 987–992.
[62] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a z3-based string solver for web application analysis. In *ESEC/FSE 2013*. 114–124.