Variable-Strength Combinatorial Testing of Exported Activities Based on Misexposure Prediction★

Xi Deng[a,c], Jiwei Yan[b,c,*], Shixin Zhang[d], Jun Yan[a,b,c] and Jian Zhang[a,c]

[a] *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[b] *Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[c] *University of Chinese Academy of Sciences, Beijing, China*
[d] *School of Software Engineering, Beijing Jiaotong University, Beijing, China*

ARTICLE INFO

Keywords:
Android Application
Exported Activity
Static Analysis
Robustness Evaluation
Combinatorial Testing

ABSTRACT

Exported Activity (EA), a kind of activities in Android apps that can be launched by external components, is one of the most important inter-component communication (ICC) mechanisms. In combinatorial testing of EAs, although exhaustive testing of all possible combinations of input elements is ideal, it is often not feasible due to the combinatorial explosion of test cases. This paper presents ExaDroid, a novel variable-strength combinatorial testing framework for generating test suites for exported activities. ExaDroid is based on two observations: many activities are unintentionally exposed, and the complexity of input interactions in activities can be very limited. ExaDroid uses misexposure prediction and complexity analysis to decide the (default) testing strength of an EA. It also leverages input interactions to focus testing resources on important combinations by setting stronger (variable) test strengths on certain attributes. Our experiments have confirmed that ExaDroid is capable of trigger many unique crashes using a dozen or so test cases. The tool successfully found 100 unique crashes across 135 EAs in 30 apps, at an average cost of 14.2 test cases per EA.

1. Introduction

The Android app market has seen an increase in specialized and collaborative app functionality. For example, an electronic payment app can be invoked by multiple third-party e-commerce apps to perform the payment process. The Exported Activity mechanism (EA for short) allows for collaboration between apps, but it can also lead to malicious manipulation and data leakage [20, 44]. Thus, activities need to be carefully implemented to avoid errors such as accepting unexpected data and throwing uncaught exceptions. This paper proposes an efficient approach to detect such defects by studying how Exported Activities are exported and implemented.

Statistics for real apps show that about two-thirds of apps have at least one exported activity (EA), with EAs accounting for about 8.6% of all activities [51]. The first question that arises is: *Are all such exposures necessary?* The recent Android 12 framework also mandates that developers be aware of the exposure state [11], but we found some EAs arise from copy-pasting or display debugging screens and may not be necessary. The second question is: *How do the exported activities interact with callers?* In this paper, we define a concept of Intent-handling complexity to express possible interactions between EAs and callers, where the Intent is the basic data structure for inter-component communication in the Android system. We found that many EAs are quite simple and generating hundreds or thousands of test inputs is a waste of resources.

The current methods for testing Android apps lack awareness of these characteristics of EAs and do not utilize them to guide testing. Random generation or mutation-based fuzzing [45, 33] is time-consuming and hinders manual review of test results, as hundreds or thousands of test intents for an activity is generated. Symbolic execution-based approaches [52, 30] traverse execution paths in the activity but rely on constraint solving and cannot handle complex intent structures well. This paper presents ExaDroid, a novel variable-strength combinatorial testing (CT) framework

★This work is supported by the National Natural Science Foundation of China (Grant No. 62102405 and No. 62132020) and the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJSSW-JSC036).

★★We are grateful to Yajun Zhu for proofreading.

*Corresponding author

dengxi@ios.ac.cn (X. Deng); yanjw@ios.ac.cn (J. Yan); zhangsx@bjtu.edu.cn (S. Zhang); yanjun@ios.ac.cn (J. Yan); zj@ios.ac.cn (J. Zhang)

ORCID(s):

that efficiently generate test suites for EAs and adapts testing strengths according to EAs' characteristics. As shown in Figure 1, this paper extends our previous work [51] of misexposure characteristic prediction (marked in orange) to facilitate the combinatorial testing.

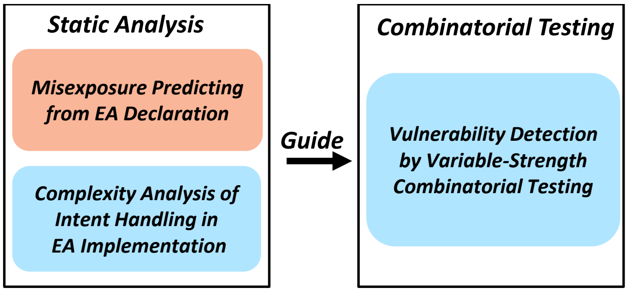


Figure 1: Misexposure Prediction and Intent Analysis Guided Combinatorial Testing on EAs

It is acknowledged that modeling of the complex input intents is of great importance. This is due to the flexibility of Intents in Android, as the Inter-Component Communication (ICC) mechanism allows for dynamic target selection and runtime binding [33]. Consequently, statically analyzing activity implementations to obtain specific structures is necessary for accurately modeling incoming intents for a given EA. We propose obtaining *function summaries* to express the input structures and to assist in model building. The next step is to select representative test intents from the model. While fuzzing-based methods are random and symbolic execution-based techniques are constrained by analysis and solving techniques, this paper adopts combinatorial testing technique to sample representative test cases. It is based on the observation that many faults are caused by combinations of a few input fields [40]. Combinatorial testing can achieve the coverage goal of combinations with as few test cases as possible.

Considering the characteristics of the EA, we optimize the coverage target setting by employing an adaptive variable-strength strategy. The strength in combinatorial testing represents the size of the combination to be covered [23]. We select model elements based on the functional summary of the EA and adjust the strength according to the EA's characteristics. Misexposed activities have a lower strength, resulting in fewer test cases, while properly exposed activities undergo thorough testing with higher strength. We believe that the exposure or non-exposure of an activity imposes varying requirements on the activity's robustness. The strength is also determined adaptively based on the complexity of the EA indicated by the function summary. During static analysis, we extract the activity path summary, which helps in generating structured caller intents and identifying dependencies among Intent attributes. By focusing more testing resources on these attribute combinations, ExaDroid effectively triggers errors and explores program behaviors.

We have developed ExaDroid, a tool that takes an app as input and maintains a dataset of caller Intents. It provides misexposure prediction results and detects bugs. In our evaluation, we utilized comparative datasets and an open-source application benchmark [53]. The experimental results demonstrate the following findings: (1) ExaDroid identifies properly exposed EAs (MustEA), which exhibit greater robustness, and misexposed EAs (MustIA), which are more vulnerable. (2) Existing approaches that do not incorporate static analysis of EA implementations or adapt testing strategies to EA characteristics can result in numerous ineffective tests. (3) ExaDroid evaluates EAs effectively, requiring an average of 14.2 test cases. Furthermore, ExaDroid successfully uncovered 100 unique errors across 30 applications, surpassing the results of symbolic execution-based approaches in terms of error detection.

In summary, this work contributes in the following three aspects:

1. Uncovering the phenomenon of misexposed and low-complexity EAs. We design a fully automated testing framework ExaDroid [1], based on the phenomenon, to improve the efficacy of testing approaches by considering variable-strength combinatorial testing;

2. Employing multiple techniques for EA analysis. The framework utilizes techniques such as misexposure prediction, Intent-handling complexity analysis, and function summary abstraction to analyze EA declarations and implementations.

[1] Both our tool and the related data are publicly available on GitHub (https://github.com/LightningRS/ExaDroid).

Table 1
Intent Attributes

Category	Manifest	Java	Attribute	Value Type
Basic	✓	✓	action, data, type	`String`
			category	set of `String`
Extra	X	✓	extra	set of ⟨*key*, *type*⟩ pair, key: String, type: Primary, Object or Bundle

3. Implementing a prototype and experimenting on different benchmarks. The results demonstrate that ExaDroid effectively discovers unique crashes using an average of a few test cases.

The paper is organized as follows. Section 2 presents the background information on Android Activity and Intents. Section 3 and 4 offer a motivating example to illustrate the challenges addressed in the paper and provide an overview of the ExaDroid testing framework. Section 5 presents the combinatorial testing strategy backed with static analysis results, which relies on the misexposure prediction technique and complexity analysis in Sections 6 and 7. Section 8 describes the implementation and experimental evaluation of ExaDroid. The paper concludes with the threats to validity, an overview of the related research, and a discussion of our future work.

2. Background

This section provides the background knowledge about Android system and exported activity.

2.1. Android Activity

The Android operating system, which is open-source and Linux-based, is primarily used on portable devices. Android apps are mainly written in Java and compiled into Dalvik byte-code, with some configuration files like the manifest file. Those apps have four types of components, including *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*, with *Activity* being the most commonly used component [29]. Activities can be internal (IA) or exported (EA), with the latter allowing other apps to launch it, making it an effective way for inter-component communication among multiple apps. This paper focuses on analyzing the activity declaration in manifest files, the activity's Java code that handles ICC messages, and the caller's Java code that sends ICC messages.

2.2. Intent Attributes

Intent [15] is the composition used for activity invocation and input wrapping. Intent attributes are divided into two types: basic and extra attributes. *Basic* attributes, such as `action`, `category`, `data` and `type`, represent the functionality of an activity and can be declared in the intent filters in the `manifest` file and used in `Java` files. The caller activity can attach an `intent` with different types of attributes via a series of overloaded APIs, as shown in Table 2. Column Manifest and Java use ✓and X to indicate whether the attribute is present. Often, owing to the Activity Exposing and Launching mechanism in Section 2.3, the manifest declares more values of attributes than those which are actually received and handled in the code[52]; however, the code can also accept values other than declared. There are often mismatches between the attribute declaration and usage.

Besides basic attributes, ICC messages also accept extra attributes. The `extra` attribute has many fields in the form of key-value pairs, which can be categorized into primary, object, and bundle types based on the value type. This attribute is only used in code and can be retrieved by the receiver activity using Android APIs. According to Android API document [15], the value can be any type of the Java primitive data type, e.g., `Integer`, `Boolean`, or other types like `String`, `Array` and `ArrayList`, etc. For example, the use of API `getIntExtra("city")` is to retrieve an integer value according to the key `city`. The value of an `extra` field can also be an object type (`Serializable` and `Parcelable`) or a bundle type (`Bundle`). 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 field.

2.3. Activity Launching and Exposing

In Android programming, an activity that can be called from outside the application is called an exported activity (EA), while an activity that cannot be accessed externally is called an internal activity (IA). The Android system

Table 2
Common Attribute Assignment APIs of Intent

Intent Basic Attribute	Intent Attribute Assignment APIs
component	setClass(Context, Class), setClassName(String/Context, String), setComponent(ComponentName), Intent(Context, Class)
action	setAction(String), Intent(String)
category	addCategory(String)
data	setData(Uri)
action, data	Intent(String, Uri)
action, data, component	Intent(String, Uri, Context, Class)

allows activities to be exposed to other applications through the use of the `android:exported` attribute and `intent filter`. There are two rules to expose an activity, according to the Android reference [13].

- An activity can be exposed if its `android:exported` attribute is set to `true`. This element sets whether the activity can be launched by components of other applications
- If the attribute `android:exported` is not set, an activity will also exposed if it contains at least one **intent filter**. Each intent filter includes one or more instances of `action`, `category` and `data`. [2]

Activities in Android can be invoked through *explicit* or *implicit* intents. Explicit intents use the fully-qualified activity class name, while implicit intents can invoke a component without knowing the exact component name [15]. Intent filters are used to match implicit intents with activities, and the Android system compares the `action`, `category`, and `data` attributes of the intent with those declared in the intent filters. Any fields in an implicit intent must be included in an intent filter, then, the Android system determines that they are a matched. It is important for developers to include the default category to ensure the mapping with implicit intents, because `android.intent.category.DEFAULT` will be added by default when an intent is created without a category. Additional criteria, such as permissions and priority, can also be used for filtering insecure callers. If multiple intent filters are compatible for an intent sent by a caller, the system displays a dialog showing options for users to pick which component to start.

3. Motivation

This section explains the declaration, implementation, and launching process of Android EAs using a simple example. It also discusses the challenges and proposed solutions by analyzing the characteristics of the Android EA mechanism and the limitations of related works.

3.1. Motivating Example

Figures 2 and 3 show the declaration and implementation of an EA `FooActivity`, respectively. In the manifest file, this EA declares the element `android:exported=“true”` and an intent filter. The intent filter enables the EA to be launched by implicit Intents that contain the action `com.intent.action.getDrink` and have no `category` or only the default one. Whether in response to an explicit or implicit call, when the EA in Figure 3 is launched, its life-cycle method `onCreate()` will be called. In line 4, the activity gets the incoming `Intent` through the API `getIntent()`. 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` and using API `getStringExtra(String str)` to get the value of primary `extra` field with a specific key. These values are typically used for branch picking, logging, or other purposes.

Figure 4 illustrates how to construct an intent in another activity to launch FooActivity. An intent instance is created with a string that denotes the `action` attribute. The `category` attribute can be set by invoking the method `getCategoryStr()` (line 4). If not set, a default value will be added automatically. Additionally, line 5 attaches an

[2] After our previous paper [51] pointed out the misuse of this mechanism, Google's new generation of Android system (Android 12, API level 31) requires the element `android:exported` to be set to an explicit value [13]. It suggests developers specify the element `android:exported` as `true` for activities that contain intent-filters. That is, only the former exposure rule is allowed in the latest system. Considering that the old versions of Android systems occupy most of the market (running on 86.7% of devices [27]), this article analyzes activities exposed in both ways. The mis-exposure patterns under them are different but similar.

```xml
<activity android:name="FooActivity" android:exported="true" >
 <intent-filter>
  <action android:name="com.intent.action.getDrink"/>
  <category android:name="android.intent.category.DEFAULT"/>
 </intent-filter>
</activity>
```

Figure 2: Manifest Declaration of FooActivity

```java
1  public class FooActivity extends Activity {
2   @Override
3   protected void onCreate(Bundle savedInstanceState) {
4    Intent intent = getIntent();
5    String action = intent.getAction();
6    String drink, cake;
7    if(action.equals("android.intent.action.getDrink")){
8     drink = intent.getStringExtra("drink");
9    }else if(action.equals("android.intent.action.getFood")
10       && intent.getBundleExtra("food")!=null){
11    cake = intent.getBundleExtra("food").getString("cake");
12   }
13   if(drink!=null){
14    logger.info("haveDrink "+
15      intent.getBooleanExtra("haveDrink"));
16   }else{
17    logger.info("cake "+ cake.trim());
18   }
19  }
20 }
```

Figure 3: FooActivity Implementation

```java
public class FooLaunchActivity extends Activity {
 public void startWithData(String uriData) {
  Intent intent = new Intent ("com.intent.action.getDrink");
  //intent.addCategory(getCategoryStr());
  intent.putExtra("drink", 0);
  this.startActivity(intent);
 }
}
```

Figure 4: Activity for Launching FooActivity

integer value of 0 to an `extra` field that has the key “drink”. Finally, the API `startActivity` is used to send the intent. It is important to note that the intent is essentially a data container that has a set of optional attributes.

The objective of this paper is to automatically assess the robustness of EAs, which refers to their ability to handle unexpected data with sufficient testing. Robustness issues in EAs arise due to missing or inappropriate key-value pairs (in the basic attributes or fields of the extra attribute), which can lead to `NullPointerException`. For example, if a value is absent, the invocation `getAction()` in Figure 3 will return `null`. Further dereferencing manipulations like `.equals()` on the unexpected value could trigger an exception. Additionally, the input intents’ special values can cause wrong program execution, resulting in app crashes. For example, line 6 of Figure 3 declares two `String` variables `drink` and `cake`. As shown in the following lines, the value of `action` attribute is used in branch picking conditions. In the two branches, either the variable `drink` or `cake` gets values from the intent fields. The intent in Figure 4 carries `action=‘‘android.intent.action.getDrink’’`, so it will assign the variable `drink`. However, its extra field of the key “drink” is assigned an inappropriate type int, therefore the value of variable `drink` is `null`. Hence, line 13 to 18 of Figure 3 accept unexpected cases. The FooActivity in line 17 expects the `cake` variable is assigned in line 11. However, the `cake` variable is not initialized and calling `trim()` throws an exception.

3.2. Challenges and Solutions

The first challenge in effectively detecting robustness defects lies in the modeling of the complex attributes of the input Intents. Under dynamic target selection and runtime binding [33], Intent can have a flexible structure, but an activity only responds to certain inputs reflected in its declaration and implementation. Additionally, the declaration and implementation may not be consistent, as shown in an example where the code uses a candidate value of `action` (`android.intent.action.getFood`) that is not declared in the manifest. Not to mention that the valid `extra` attribute structures and keys are only pictured in the implementation. Hence, to effectively model incoming Intents for a given EA, static analysis of the activity implementation is required to obtain specific attribute values and structures.

The second challenge is how to select representative test intents from the model. Two existing approaches are generation or mutation-based fuzzing and symbolic execution-based approach. Fuzzing results in hundreds or thousands of test intents for each activity [45, 33], which is costly and hinders manual review of test results. Symbolic execution-based approach [52, 30] collects constraints along each execution path, but constraint solvers don't handle attributes' string values very well (operations like `split()` and `lastIndexOf()` can not be converted into SMT constraints [47]). Not to mention the complex data types of `extra` attribute and exception throwing conditions. So the symbolic execution-based works also use mutation to enhance testing with `null` value, boundary values, etc. The article proposes the use of combinatorial testing to generate a limited number of test cases from a test model. Given an EA's Intent structure and attribute candidate values, combinatorial testing generates the fewest test cases that satisfy a heuristic coverage metric. It is based on the empirical finding that across various domains, all failures could be triggered by combinations of maximum *four to six* values [23]. Therefore, the heuristic t-way (t is an integer less than or equal to 6) coverage can sample a small number of tests to achieve error detection. The same finding also occurs in EA testing. On BenchFdroid that we adopt for experiments, with models that contain 10 to 38 parameters (flattened from intent attributes, see Section 5.1 for detail), we find that 32% of the failures are triggered by only a single parameter value, 88% by three-way combinations, and 98% by four-way combinations. An example of a two-way combination is action=android.intent.action.getDrink, extra_drink=null, which makes FooActivity throw exceptions for any intents that contain the combination.

The third challenge is determining the coverage goal of each EA in combinatorial testing. While not all input fields have interactions [48], potential interactions must be identified as coverage targets. In our static analysis, we collect execution traces and trace attribute operations to form a summary of the EA to be tested. Attributes appearing on a path are believed to have interactions. The summary helps refine variable-strength testing. However, determining the value of t for t-way coverage remains a problem. Existing works always take each EA equally, but it may lead to a waste of testing resources. From EA declarations, we find a set of EAs that are wrongly exported by developers. Many of EAs remove the `category` declaration, which reduces the possibility of such EAs being called implicitly. We identify such wrongly exported EAs from EA declarations and allocate a smaller t for them, as their unrobustness may not be exploited. Since the exposure state precedes the activity's input validation to secure the activity, it is more important to identify the misexposure and change the exposure state to an IA declaration to exclude all third-party callers. We also study EA implementations to set the number t. Thus, the coverage setting is adaptive to the EA under test.

4. Framework Overview

As shown in Figure 5, we have developed a tool called ExaDroid that takes an apk file as input, generates a group of test cases, and outputs the analysis and test execution results. The tool consists of two main modules: a static analysis module called **ExaDroid$_{mis}$** and a test generation module called **ExaDroid$_{ct}$**.

First, ExaDroid$_{mis}$ performs static analysis on the EA implementation to obtain a summary that describes the attribute usage information along execution paths. Then, ExaDroid$_{ct}$ converts the summary into a combinatorial testing model. Next, based on the summary of EA implementations and the analysis results of EA declarations, ExaDroid$_{mis}$ represents EAs as feature vectors for classification. The classification process takes as input a dataset of caller intents from tens of thousands of apps and classifies EAs into four vulnerability exposure categories and two complex categories. The model then takes the classification result and summary to guide the coverage setting.

5. Combinatorial Testing of Exported Activities

Intent is essentially a data container with optional attributes, and the interaction among a few attributes or attribute fields is often the root cause of defects. To achieve interaction coverage, combinatorial testing can efficiently generate

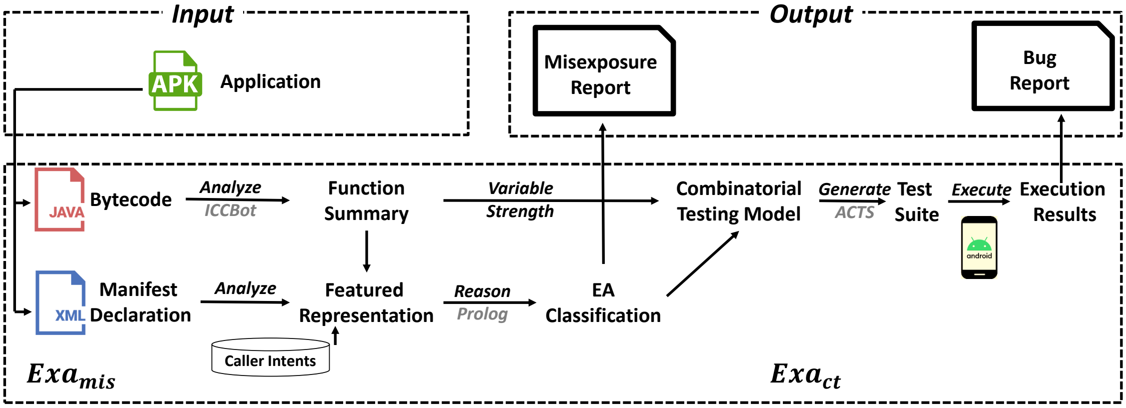


Figure 5: Overview of Variable-Strength Combinatorial Testing of EAs

tests compared to existing approaches. Therefore, the static analysis method is employed to build a combinatorial testing model that represents the input structure of the exported activity and to guide the coverage goal setting.

5.1. Summary-Based Test Model Building

In this section, we consider an EA as a parametric model whose behavior is influenced by a limited number of input parameters, specifically intent attributes and attribute fields. The CT model $\mathcal{M}$ of an EA consists of a set of parameters *Param* and candidate values for each parameter *Value*. The coverage goal is a set of combinations, where each element is a t-size *combination* c ($t \leq |Param|$) that assigns values for t parameters.

It is important to note that each combination c must be covered by at least one test case, where a test case is a combination with all parameters assigned. Based on the model, the CT test generation process assigns values to each parameter to form a test case, and selects as few test cases as possible to achieve the coverage goal.

It has been proven that CT can effectively detect interaction defects with fewer test cases, but the effectiveness to a large extent depends on the quality of the model [7]. Therefore, this section statically analyzes EA implementations for building CT models. The modeling methodology works in two steps, namely, static analysis-based input structure identification and flattening-based parameter modeling.

5.1.1. Function Summary

We try to capture input structure by obtaining function summaries. Algorithm 1 describes the analysis of the application containing the EA under test and returns a function summary of the EA's entry method.

Line 2 builds an acyclic function call graph of the app and rearranges items in a bottom-up order to get *funcList*. Then, for each function (the variable *func*) in *funcList*, we traverse its control flow graph to get all paths and traverse the path instructions.

Variable *pathSummary* is a set used to store all the retrieved parameters (attributes and attribute fields) in a path, and it is initialized as an empty set. If the instruction (the variable *ins*) is an input intent obtaining instruction, that is, it belongs to relative Android APIs (such as getAction() and getStringExtra()), then function *getParam* extracts a triple $\langle param, type, canValue \rangle$ and adds it to *pathSummary*.

Function *getParam* relies on du-chains in current function *func*, the result of an intra-procedural reaching definition analysis [50]. Thus, it can track the transfer of attribute receiving variables as well as the key declaration and value types of extra fields. A basic attribute retrieving statement, e.g., `String action = intent.getAction()`, will be recorded as param=action and type=String; an extra field retrieving statement, e.g., `drink = intent.getStringExtra(‘‘drink’’)`, will be recorded as param=extra_drink and type=String, where symbol "_" indicates that the extra attribute has a field with key "drink". Then, function *getParam* collects a group of statements in *path* which use the variable that stores basic intent attribute for comparison, and updates the candidate value set *canVar* with the comparing objects (e.g., the string *"android.intent.action.getDrink"* in the statement `action.equals("android.intent.action.getDrink")` for variable *action*). Apart from these direct input intent obtaining instructions, if the instruction invokes function, then we add the summary of the callee function *func′* (*summaryMap*[*func′*]) to *pathSummary*, which could be a set as a function summary may contain multiple paths.

The method *merge* operates on each element in *pathSummary*. Then, line 12 aggregates all paths to get the function summary. Finally, we get the function summary of the EA's entry method (the variable $func_{entry}$) from *summaryMap*. We use the ICC resolution tool ICCBot [54] for static analysis. Please refer to this tool for detailed Android APIs and the maximum number of paths limitation to avoid path exploration, etc.

Algorithm 1 EA Implementation Analysis

Input: an apk
Output: the summary
1: *summaryMap* ← ∅
2: get function traverse sequence as *funcList*
3: **for** each function *func* in *funcList* **do**
4:     *summaryMap*[*func*] ← ∅
5:     get all paths in the function as *pathList*
6:     **for** each *path* in *pathList* **do**
7:         *pathSummary* ← ∅
8:         **for** each *ins* belongs to input intent obtaining instructions in *path* **do**
9:             *pathSummary*.merge(getParam(*ins*, *path*, *func*))
10:        **for** each *ins* belongs function call instructions in *path* **do**
11:             *pathSummary*.merge(*summaryMap*[$func'$])
12:        *summaryMap*[*func*] ← *summaryMap*[*func*] ∪ *pathSummary*
13: **return** *summaryMap*[$func_{entry}$]

For the example provided in Section 3, we can get the function summary as summaryMap[func_{*entry*}]={
{⟨action, String,{"getDrink"}⟩,⟨extra_drink, String, ∅⟩,⟨extra_haveDrink, Boolean, ∅⟩},
{⟨action, String,{"getDrink"}⟩,⟨extra_drink, String, ∅⟩},
{⟨action, String,{"getDrink","getFood"}⟩,⟨extra_food, Bundle, ∅⟩,⟨extra_haveDrink, Boolean, ∅⟩},
{⟨action, String,{"getDrink","getFood"⟩,⟨extra_food, Bundle, ∅⟩},
{⟨action, String,{"getDrink","getFood"}⟩,⟨extra_food, Bundle, ∅⟩,⟨extra_food_cake, String, ∅⟩,⟨extra_haveDrink, Boolean, ∅⟩},
{⟨action, String,{"getDrink","getFood"}⟩,⟨extra_food, Bundle, ∅⟩,⟨extra_food_cake, String, ∅⟩}}.

5.1.2. Combinatorial Testing Model

The complex intent attributes should be converted into parameters with simple value types and discrete values, along with their corresponding relations and constraints. Attributes of non-string types, such as `category` and `extra`, as well as the structured attribute `data` (URI := scheme/path?query) are flattened into multiple parameters in the model. In the example provided in Section 3, the `extra` attribute is modeled as 5 parameters: extra (in a different font to distinguish parameter from attribute), extra_drink, extra_food, extra_food_cake, extra_haveDrink. The model contains constraints that organize these parameters into structured inputs. For instance, a Bundle field of the `extra` attribute is composed of a String sub-field, so *extra_food* is an abstract parameter that can only take empty or non-empty values, and the following constraints need to be met: if extra_food takes an empty value, extra_food_cake is also empty; otherwise, if extra_food_cake takes a specific string value, extra_food takes non-empty. We automatically extract parameters and constraints from the summary of an EA under test.

For the *Value* function that maps a parameter to a finite set of candidate values, we support multiple value-taking strategies. The Base strategy is as follows. (1) For basic attributes, we traverse all path summaries in the function summary to obtain triples ($\langle param, type, canValue \rangle$), and update *Value(param)* with the set *canValue*. (2) Since the static analysis traverses all functions in an apk, explicit intents constructed in other components with the EA under test as the invocation target are also parsed for candidate values. (3) A `category` attribute is flattened into several parameters. Each candidate value of this attribute can either exist or not, so we represent its occurrence with a Boolean parameter. (4) From the model, we build a test case as an explicit intent, so each attribute is optional and value empty is considered as a candidate value for all attributes. (5) Considering that missing key-value pairs may cause the most common exception, we update *Value* of attributes of type String with the value null. The difference is that value empty does not call the API to set the attribute, while value null calls the corresponding API and passes in a null value. (6) For an `extra` field or sub-field of the Primary type, the *canValue* is empty but the field can take arbitrary values. We

Table 3
The Combinatorial Testing Model for FooActivity

Param	Value
action	[null, empty, getDrink, getFood]
category	[empty, non-empty]
category_null	[true, false]
category_empty	[true, false]
extra	[empty, non-empty]
extra_drink	[null, empty, "a"]
extra_food	[empty, non-empty]
extra_food_cake	[null, empty, "b"]
extra_drinkNumber	[empty, true, false]
data	[null, empty, non-empty]
path	[null, empty, non-empty]
scheme	[null, empty, non-empty]
type	[null, empty]
Constraints	
category = empty → category_null = false ∧ category_empty = false	
category_null = true ∨ category_empty = true → category = non-empty	
extra = empty → extra_drink = empty ∧ extra_food = empty ∧ extra_drinkNumber = empty	
extra_drink != empty ∨ extra_food != empty ∨ extra_drinkNumber != empty → extra = non-empty	
extra_food = empty → extra_food_cake = empty	
extra_food_cake != empty → extra_food = non-empty	
data = empty → scheme = empty ∧ authority = empty ∧ path = empty	
scheme != empty ∧ (authority != empty ∨ path != empty) → data = non-empty	
scheme = empty → data = empty	

generate a set of candidate values according to *type*, that is, true and false for Boolean type parameters, an empty and a random value for String type parameters, etc.

For the example in Section 3, we can get the model as Table 3. The model consists of 13 parameters, each with a discrete range of values, and 9 constraints that restrict these parameters to form a valid Intent object. A constraint is a predicate on the value of some parameters. Static analysis identifies input structures and candidate values to build a target testing model for an EA.

We offer various strategies, such as Manifest, Random, PresetBound, etc., to allow testers to customize their approach. The Manifest strategy utilizes the declared values of basic attributes in the intent filters in the manifest as candidate values. Unlike previous works [55, 58], we do not consider this strategy as the default due to the potential mismatch between declaration and implementation. The Random strategy constructs generators for Integer, String, URI, and other types. For example, the String generator enables testers to customize the maximum length (default is 5) and randomly generates multiple strings between the minimum and maximum length. The PresetBound strategy provides invalid values for the `category` and `data` attributes, boundary values for numeric `extra` fields, and non-empty values for strings.

5.2. Variable-Strength Coverage Setting

Given a model, the coverage setting is an important factor that affects the size and error detection ability of the generated test set. The testing *strength* of a CT model is the size t of combinations that should be covered. Black-box CT modeling often assumes that all parameters are likely to interact with the same strength, which is heuristic and leads to redundant testing. To address this issue, we propose a summary-based variable-strength coverage setting. The extracted summaries allow us to identify the dependencies among intent attributes, thus enabling us to focus testing on real interactions. A variable strength is a tuple $(Param_i, t_i)$ where $Param_i \subseteq Param$ and t_i is the strength on $Param_i$. Multiple such tuples could be given, denoted as $t^+ = \{(Param_1, t_1), (Param_2, t_2), \dots\}$.

5.2.1. Dependency of Intent Attributes

Two attributes are considered dependent if the combination of their values affects an application's control or data flow [37]. This is because the combination of their values can cause the program to execute a specific path that may contain incorrect code [34, 24] or result in incorrect calculations [43]. Figure 6 shows two examples of attribute dependency in the demo activity. In the left snippet, the value of the `action` attribute influences a string variable (the variable `drink`), whose value determines whether the extra field with the key "haveDrink" will be used. In the right snippet, the use of a Bundle field with the key "food" depends on the comparison operation of the `action` attribute, and the use of the field "cake" depends on both conditions.

```java
if(action.equals("android.intent.action.getDrink")){
 drink = intent.getStringExtra("drink");
}
if(drink!=null){
 System.out.println("haveDrink "+
   intent.getBooleanExtra("haveDrink"));
}
```

```java
if(action.equals("android.intent.action.getFood")
 && intent.getBundleExtra("food")!=null){
 cake = intent.getBundleExtra("food").getString("cake");
}
```

Figure 6: Dependency Types in the Example Code

Inspired by works on symbolic execution-based testing, we capture the potential dependencies from the extracted function summaries. The main idea of coverage setting is that dependent parameters will appear on the same program execution path, and thus, the interaction between parameters on a path should be covered.

5.2.2. Automated Strength Configuration

Algorithm 2 outlines the process of setting variable-strength coverage based on the function summary. In Line 2, the function summary is simplified to obtain a set of non-overlapping path summaries (the variable *simplifiedSummary*). For instance, the demo function summary in Section 5.1.1 is simplified as {{⟨action, String,{"getDrink"}⟩,⟨extra_drink, String, ∅⟩,⟨extra_haveDrink, Boolean, ∅⟩}, {⟨action, String,{"getDrink","getFood"}⟩,⟨extra_food, Bundle, ∅⟩,⟨extra _food_cake, String, ∅⟩,⟨extra_haveDrink, Boolean, ∅⟩}}. The remaining path summaries are then analyzed. In Line 4-6, the triple in *pathSummary* is traversed to collect a subset of parameters (the variable $Param'$). These parameters are retrieved in the same path for branch picking and other purposes, so there may be interaction defects. Therefore, in Line 7, they are added, along with a given strength t, to the variable strength set t^+. The tuple requires that test suite generated should cover all combinatorial interactions of any t (out of $|Param'|$) parameters in set $Param'$ at least once. For the motivating example, the algorithm returns $t^+ = \{(\{action, extra\_drink, extra\_haveDrink\}, t), (\{action, extra\_food, extra\_food\_cake, extra\_haveDrink\}, t)\}$.

Algorithm 2 Combinatorial Coverage Identification

Input: function summary $summaryMap[func_{entry}]$, variable t
Output: variable strength t^+
1: $t^+ \leftarrow \emptyset$
2: $simplifiedSummary \leftarrow$ simplify($summaryMap[func_{entry}]$)
3: **for** each *pathSummary* in *simplifiedSummary* **do**
4:     $Param' \leftarrow \emptyset$
5:     **for** each triple $\langle param, type, canValue\rangle$ in *pathSummary* **do**
6:         $Param' \leftarrow Param' \cup (param)$
7:     $t^+ \leftarrow t^+ \cup (Param', t)$
8: **return** t^+

The size of a combinatorial test suite increases rapidly as t increases [22]. This highlights the important question of how the strength value should be set. A reasonable choice of t requires experience with the software being tested. For example, within the NASA database application, 67% of the failures were triggered by only a single parameter value, 93% by two-way combinations, and 98% by three-way ones [23]. Setting $t = 3$ or fewer would cover one hundred percent of i-way combinations (where $i \leq t$) and also a good portion of larger combinations (e.g., a test case is a $|Param|$-way combination), providing a form of "pseudoexhaustive" testing. In fact, in many CT practices, choosing 3-way testing is the default choice for testers. In EA testing, we study the characteristics and classification of EAs and provide two t values (the common practice $t = 3$ and a progressive $t = 1$) for adaptive configuration, in contrast to assigning the same t to all EAs. The following two sections will introduce our study.

6. Misexposure Prediction from EA Declaration

Existing works test all activities with the same coverage goal regardless of discrepancies in activity risk levels. One source of these discrepancies is whether activities are improperly exported. We find that developers may misunderstand the declaration mechanism of EA, causing some activities to take the unnecessary risk of external calls. For example, an incorrectly exported activity with no default `category` value will never be invoked implicitly through the Android system mapping. As such, it can only be invoked by a caller who knows its component name, and of course its details, so it has less chance of accepting unexpected data in actual use; moreover, changing its exposure state can prevent it from malicious external exploitation. Therefore, the research in this section focuses on identifying patterns of misdeclared EAs, for two purposes. First, to issue a report to help developers fix the improper exposure status. Second, to allocate a smaller t ($t = 1$) for combinatorial testing, that is, using a small number of tests to verify the report, but avoid wasting testing resources.

Usually only the developer can define whether an activity should be exported or not, but the misexposure may come from developers' misunderstanding or carelessness. Therefore, we conduct comparative and manual analysis to extract the misexposure patterns. In our previous work [51], we collected 13,873 Android apps from open-source app repositories including F-Droid, Google Play, and a Chinese app market Wandoujia to study the EA usage. From the benchmark, we obtained two small datasets using different picking criteria, in which Set MD contains EAs that belong to widely used apps and Set AR contains EAs that come from apps with abnormal ratio of EAs.

We begin by extracting the declarations from the app's manifest files. These declarations are then compared statistically to identify any differences. Next, a human inspector analyzes each EA declaration to investigate the cause of the differences and determine whether they represent spurious exposures. Finally, three human assessors collaborate to establish rules for identifying misexported activities.

6.1. Comparative Analysis

6.1.1. Construction of datasets

We aim to create two sets of apps with contrasting characteristics: one mature and the other abnormal. To achieve this, we utilize two metrics: an anomaly detection metric and a metric indicating app popularity. The first metric we use is the *Local Density Deviation* (LDD) [8], which is a metric for unsupervised anomaly detection. We represent each app as a data point in a coordinate system where the abscissa is the number of activities and the ordinate is the proportion of EAs in activities. The second metric we use is the number of downloads, which is an indication of how widely used the app is and likely correlates positively with app maturity.

Using these metrics, we obtain two sets of apps. The first set, called **Set AR (Abnormal Ratio)**, contains apps with abnormally high ratios of EAs compared to most apps. Intuitively, an app should expose as few well-defined interfaces as possible, and that is what most apps do. However, the percentage of EAs in some apps is abnormally high. We conducted a small empirical study on those apps to investigate whether their EAs might be improperly exported. We selected the top 5 apps with the most abnormal LDD from the large-scale benchmark and extracted 327 EAs from them. By launching each EA and inspecting its bytecode to judge the reasonableness of the exposure, the tester reported that 234 EAs (72%) are suspected to be misexposed. Thus, we assume that the apps with an abnormal ratio of EAs compared with most of the apps are more likely poorly programmed. is used to identify outliers based on density for LDD computation. However, apps with many activities and few EAs may also be identified as outliers by the LOF algorithm, even though they may be well-programmed apps. To address this issue, we filter out apps whose ratio of EAs is less than 0.1. Then, the top 50 outliers are labeled as *outlier apps* and added into Set AR.

The second set is called **Set MD (Most Downloads)**. In order to conduct a comparative analysis between normal apps and outlier apps in Set AR, we aim to extract EAs that have a higher likelihood of being well-declared. Since widely used apps are more likely to be developed under strict code regulations by skilled programmers and may have been thoroughly tested, the EAs corresponding to such apps are more likely to be well-declared and suitable for comparison. We also conducted a small empirical study to generate this dataset. We selected 5 apps with the most downloads and extracted 47 EAs from them. By launching each EA and inspecting its bytecode to assess the reasonableness of the exposure, we found that only 5 EAs (10.6%) were labeled as misexposed activities. Additionally, most of these widely used apps (4/5) provide specific SDKs (e.g., WeChat's open SDK[49]) for external invoking.

6.1.2. Observation

The two selected sets are disjoint and each contains 50 apps. We analyze and compare the number and pattern of EA declarations in the two sets of apps to identify the statistical characteristics of misexposed EAs.

Table 4 lists the general information, where the columns App_size and #EA report the sum values across the 50 apps. Figure 7 displays the EA number distributions of the collected apps using box-plots. The number of EAs extracted from each app is on average 12 in Set MD and 113 in Set AR, as shown by the X mark symbol. Set MD contains one

Table 4
General Information about Comparative Sets of Apps

	#App	App_Size	#EA (percent.)
Set MD	50	1,332MB	598 (8.1%)
Set AR	50	1,522MB	5654 (38.5%)

outlier that is about four times greater than the median and Set AR contains two outliers that are about five times greater than the median, marked by the dot and solid median line.

As we can see, the first difference between these two sets is the number of EAs. While the size is similar, apps in Set AR have many more EAs than those in Set MD. The last column gives the ratio of EAs, which shows that EAs in Set AR have a larger proportion. We further investigate how developers expose an EA with different exposure modes and show the results in Figure 8. The exposure mode "ExTrue" indicates the EAs whose attribute `exported=true` is explicitly declared, and "NoEx" indicates the EAs without that attribute. The ratio of activities declared with `exported=true` varies greatly in these two datasets, which is 50% in Set MD but only 15% in Set AR. This shows that **the attribute `exported`, which demonstrates the intention of developers explicitly, is more often used in well-programmed apps.**

For EAs in mode "NoEx", we further separate them into three types: "SysActData", "SysActNoData" and "NonSysAct", according to whether they contain any data and non-system action or not. Actions can take system values that are defined in the `Intent.java` file of Android source code, or take non-system values that are provided by third parties. The exposure mode "SysActData" indicates the EAs whose `action` attribute takes system values and contains `data` attribute, "SysActNoData" indicates the EAs without `data` attribute, and "NonSysAct" indicates the EAs whose `action` attribute takes non-system values. The results show that **the mode "SysActNoData" is rarely used in both datasets**. In our previous work [51], we studied action values used in intent filters. It was observed that 69% of 63,758 used action values extracted from 13,873 Android apps are the officially provided system ones (57% are `android.intent.action.VIEW` used to display data to the user), so system action values are difficult to be taken as the identifier of an EA. Developers always add data, the URI object that assigns the data to be acted on, to limit the range of resolved activities. Therefore, the EA declaration that contains only system actions without data item required is likely to be misused. Thus, the mode "SysActNoData" is abnormal.

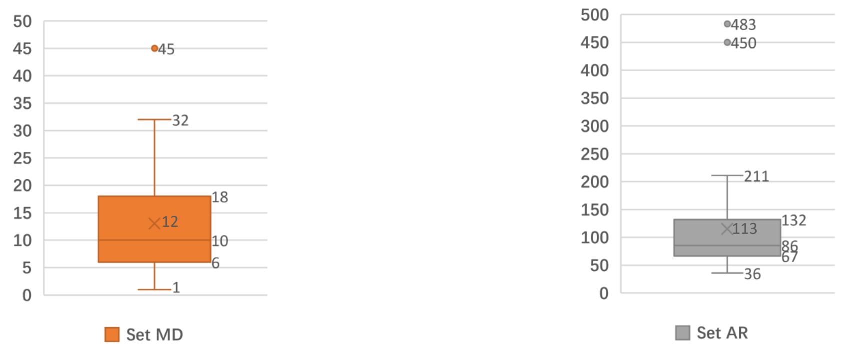


Figure 7: EA Number Distribution

6.2. Misexposure Prediction

6.2.1. Patterns of Improper EA Declarations

The comparative analysis results guide the extraction of misexposure patterns. Additionally, manual inspection of 100 apps from Set MD and AR, as well as the Android reference, is conducted to aid in the identification of these patterns. Five patterns of improper declaration are identified, which are unlikely to be invoked by third-party callers. These patterns are detailed in our previous paper [51] and are shown in Table 5. P1 and P2 prevent an EA from being called implicitly or reduce the possibility of it being called. P1 is extracted from the Android reference, while P2 is

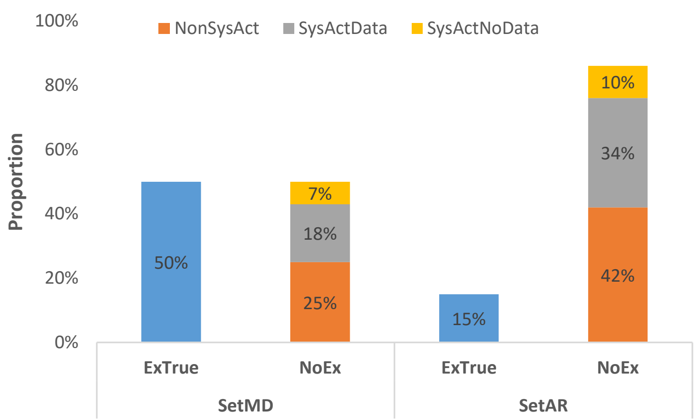


Figure 8: Exposure Mode Comparison

Table 5
Patterns of Improper Declaration

Pattern	Explanation	Case Study	Detection
P1: Missing Default Category.	An EA must include the `DEFAULT` category to be called by implicit Intents [14].	An example EA in *Mozilla* will show a blank window and give the `WindowLeaked` error in logcat.	Analyzing the intent-filers of each EA.
P2: System Action and No Data.	Mode "SysActNoData" is rare because the system `action` cannot identify one EA, so there must also be a `data`.	The declaration of an EA in *UCMobile* only uses the most frequently used system action `android.intent.action.VIEW`.	Analyzing the intent-filers of each EA.
P3: Abnormally High Percentage of EA.	The comparative results in Table 4 show that poorly programmed apps may have a higher EA percentage.	All functions in *Mobile collaboration* rely on log-in, but 58/59 of activities are EAs that can be easily accessed without login, which may violate the intention of developers.	EA ratio statistic.
P4: Copy-Pasted EA Declaration.	Copy-pasted EA declarations widely exist in Set AR and MD.	Up to 124/128 of EAs in app *ToolWiz Photos* are declared using mode "ExTrue". Developers do not expose the EAs deliberately since starting activities will cause app crash.	Calculating the ratio of each mode in one app.
P5: Debugging Functionality.	Some activities are designed and exposed to ease the debugging, according to their names. Forgetting to remove them in the release versions may threaten the database security.	An EA whose `android:name` attribute is "*com.dianping.debug.DebugDomainSelectActivity*" is used for domain testing.	Keyword retrieving

from our observation in Section 6.1.2. An EA that meets P1 is considered to be misexported, and an EA that meets P2 requires further analysis. P3 is the hypothesis used to extract Set AR and was confirmed by our manual analysis. P4 is a possible cause of P3. P5 is found through our manual analysis of the extracted EA declarations and is surprisingly one of the main functionalities used by developers. In [51] we manually categorize the 300 most used action values into several functionalities, including Display, Send, Other, SDK, Search, Setting, and Debugging. These patterns can all be automatically detected, as shown in the Detection column of Table 5. We provide further details on P2 and P4 in this section.

P2: System Action and No Data. ***1) Explanation:*** According to the Android reference and manual inspection, it has been found that the officially provided system action values are commonly used with a data attribute. Therefore, an EA declaration that contains only system actions without a required data item is likely to be misused. ***2) Case Study:*** To facilitate understanding, a real case in *UCMobile* is discussed. As shown in Figure 9 (a), it declares an EA with the most frequently used system action `android.intent.action.VIEW` only. When an implicit call that only contains this system action is sent, dozens of EAs are matched as candidate options available to the user, sorted by priority value. However, this intent filter only has default priority and may not even show up in the dialog. ***3) Detection:*** This pattern can be identified by analyzing the intent-filers of each EA. While this pattern reduces the likelihood of activities being invoked and is therefore a dubious usage, users are still authorized to use it in exported activities. To be conservative, only EAs that satisfy this pattern are considered misexported if the developer does not explicitly declare android:exported=true.

P4: Copy-Pasted EA Declaration. ***1) Explanation:*** When developers want to declare an activity correctly, the most convenient way is to imitate the last declared one, i.e., declare by copy-and-paste. By manual inspection of the manifest files of the selected apps in both datasets, it has been found that copy-pasted EA declarations are widely used. ***2) Case Study:*** An app called *ToolWiz Photos* has a total of 197 activities and 128 of them are EAs. Surprisingly, up to 124 EAs in this app are declared using the mode "ExTrue". In Figure 9 (b), some of them are shown and all the expose-irrelevant attributes are removed. If the activity `UserInfoActivity` is started using the adb command, the app will crash and throw an exception, which means that the developers did not deliberately expose it. And `SelectiveColorActivity` is used for adjusting the tone for a photo. External invocation is allowed to directly start it without a target image, which makes this activity invalid and even causes a crash. ***3) Detection:*** Misexposures in this pattern can be found by calculating the ratio of each exposure mode in one app. Some thresholds are set to classify a ratio as indicating misexposure or not.

```
<activity android:name="com.ucweb.activity.
LifeAssistantActivity">
 <intent-filter>
  <action android:name="android.intent.action.VIEW"/>
  <category android:name="android.intent.category.
  DEFAULT"/>
 </intent-filter>
</activity>
```

(a) System Action and No Data

```
<activity android:exported="true" android:name="
com.toolwiz.photo.community.UserInfoActivity" />
<activity android:exported="true" android:name="
com.btows.photo.editor.ui.SelectiveColorActivity"/>
```

(b) Copy-Pasted EA Declaration

Figure 9: Improper Declared EAs

6.2.2. Misexposure Characterization and Identification

We begin by abstracting an EA declaration to evaluate whether it conforms to the pattern mentioned above. ExaDroid$_{mis}$ takes an apk file and a Caller Intent database as input. From the manifest files in the apk, we identify all the $\langle activity \rangle$ tags and collect the activity attribute (e.g., `android:exported`) as well as the intent filter (e.g., `android:action`) information. The Intent database is obtained from the reaching definition analysis [50] of 13,873 apps. We use the Java bytecode analysis framework Soot [6] to determine how EAs are invoked and store all invocations used in the program. We query the database and map an EA declaration to caller intents to analyze whether the EA would be invoked externally. The parsing results of manifest files and the query results of the database are encoded into the features in Table 6, which are divided into three categories:

- The first 8 features represent the exposure mode of one EA, obtained by analyzing the manifest file.
- The following 3 features come from all EAs in the app, also obtained by analyzing the manifest file.
- The last 2 features come from querying the database and indicate the invocation of a specific EA.

Then, an EA declaration is encoded as a vector $ea_{d,c}$ (d for declaration and c for calling), where each element $\forall f \in ea_{d,c}, f \in \{0,1\}$ is a feature in Table 6. The second column describes the condition to make $f = 1$. We summarize a series of rules to determine the necessity and reasonability of the exposure. These rules apply to the featured representation of the EA. A function R will return the predicting result of an EA, i.e.,

$$R(ea_{d,c}) \in \{\texttt{MustIA}, \texttt{MayIA}, \texttt{Unclear}, \texttt{MustEA}\}.$$

The class Must and May differ in that May contains some coarse-grained rules that can not tell which specific EA is misexposed. The predicting follows the patterns of improper declaration, and the classification conditions are in

Table 6
Features of EA Declaration

Category	Feature	Description
Declaration	`exTrue`	declares `exported=true`
	`ifTrue`	contains `intent filter`
	`mainAct`	contains `android.intent.action.Main` and `android.intent.category.LAUNCHER`
	`noDefault`	omits the default category
	`sysActNoData`	declares only system action without data
	`priority`	contains priority setting of intent filter
	`permission`	contains permission setting of activity
	`debug`	contains keywords such as “test”, “debug”, “Test”, and “Debug” in the action, category or activity name
	`similar`	belongs to an app that declared EA with the similar exposure mode shown in Figure 8, including ExTrue, SysActData, SysActNoData and NonSysAct.
	`highRatio`	belongs to an app that has high value of #EA/#A
	`clsDeclare`	with classname that has been declared more than three times in manifest
Invocation	`clsInvoke`	with classname that has been externally invoked
	`actInvoke`	with non-system action that has been externally invoked

Table 7. The columns also show the priority (column Pr) of rules and the judgment conditions (column Condition), which uses single or the combination of features listed in Table 6. About the priority setting, three Android testers reach a conclusion after discussion. Classification results from high-priority rules are considered more credible. One EA may satisfy several conditions at the same time, when condition collision occurs, the final classification is determined by the priority value. For example, Figure 10 shows an activity declaration that contains `android.intent.action.MAIN` (Class MustEA, Pr 0), and omits the default category, and contains intent filters (Class MustIA, Pr 2). Obviously, the main activity is the component that the developer expects to be called externally, although it cannot respond to implicit calls. It will be classified as MustEA at last.

```xml
<activity android:name="ch.hgdev.toposuite.entry.MainActivity">
 <intent-filter>
  <action android:name="android.intent.action.MAIN" />
  <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
</activity>
```

Figure 10: Priority Setting Example

Compared to our previous work [51], the classification rules in Table 7 have been updated. Pr 9 and Pr 10 are Android mechanisms that allow third-party apps to invoke the activity. EAs that fall into these two categories do not satisfy any rule from 0-8, nor do they show a clear tendency for improper exposure or proper exposure. In the original version, we divided Pr 9 as MayEA and Pr 10 as MayIA, since in Section 6.1, the exposure mode ExTrue showed a significantly higher proportion in Set AR. However, the recent release of Android 12 requires that `android:exported` should be explicitly assigned to pass the compilation. The probability that a misexposed EA will be in the mode ExTrue also increases. Therefore, we introduce the category Unclear for Pr 9 and Pr 10.

We implemented *R* using Prolog, a logic programming language that automatically identifies misexposures. The program logic is expressed in terms of relations, represented as *facts* and *rules* [10, 35]. A fact consists of an attribute and its value, while a rule is in the form of `Head:-Body.`, in which the *Head* is the conclusion and the *Body* contains several facts. If the facts in Body are true, the Head is also true. Figure 11 shows part of our implementation using Prolog, where the declaring order of rules determines the priority of matching. We attributes of fact listed in Table 6, whose values can be extracted through EA declaration and invocation analysis. For each EA, we can obtain 11 facts to aid in classification. We define ten rules using conditions and their classes in Table 7, where the IA classification conditions are linked to misexposure patterns. For example, line 4-5 in Figure 11 represent a rule, which means if the fact `clsDeclare(true)` is satisfied, the class of corresponding activity is `MustEA`. For other rules that contain several facts, these facts are combined, where the comma indicates “and”, and the semicolon indicates “or”. For instance, lines 1-3 mean that if an EA satisfies `(noDefault(true) and not ifTrue(true) and not exTrue(true)) or (debug(true))`, it belongs to class `MustIA`.

Table 7
The Classification Conditions of EA Declaration

Class	Pr	Condition
MustIA	1	`sysActNoData and ifTrue and not exTrue`
	2	`noDefault and ifTrue`
	3	`debug`
MustEA	0	`mainAct`
	4	`clsInvoke or actInvoke`
	5	`clsDeclare`
	6	`priority or permission`
MayIA	7	`similar`
	8	`highRatio`
Unclear	9	`exTrue`
	10	`ifTrue and not exTrue`

```
activity(mustIA):- noDefault(true), not(ifTrue(true)), not(exTrue(true)); debug(true);
activity(mustEA):- clsDeclare(true).
```

Figure 11: Part of the Implementation Using Prolog

7. Complexity Analysis of Intent Handling

In this section, we will demonstrate that many EAs rarely process input Intents. With this in mind, we propose an EA classification method based on the number of paths that handle Intents and the retrieved Intent attributes along paths. It evaluates whether the EA will respond to inputs from external invocation. We also describe the reasons for setting different testing strengths t for different categories.

7.1. Comparative Analysis

7.1.1. Construction of Datasets

Using the EA declaration predicting on apps of Set AR and Set MD, we obtain different sets of EAs to analyze their implementations. The analysis is based on function summaries, so 15 apps are excluded out of 100 apps in Set AR and Set MD: 1 app failed Soot analysis, and the other 14 apps were obfuscated so that the activity names in code could not correspond to the manifest declaration. The other apps contain 2,834 EAs: 439 are classified as MustEA. Table 8 shows different sets of EAs classified by misexposure prediction. Column #Path comes from the function summary. A path is a list of triple $\langle param, type, canValue\rangle$, e.g., {⟨action, String,{"getDrink"}⟩,⟨extra_drink, String, ∅⟩,⟨extra_haveDrink, Boolean, ∅⟩} in the example in Section 5.1.1. The list is non-empty because the summary has filtered out intent-independent paths that do not retrieve the input intent (getIntent) or any intent attributes (e.g., getAction and getStringExtra(key)). The number of such paths (#Path) is the cardinality of the function summary set, i.e., |summaryMap[func_{*entry*}]|. If #Path= 0, column Empty Summary indicates the EAs whose execution will not retrieve intent attributes. Otherwise, we count the mean and median of the number of paths for different categories of EAs. Figure 12 displays the path number distributions of EAs whose #Path> 0 using box-plots. As we can see, the median shown by the solid lines in the four sets ranges from 2 to 5, indicating that half of the EAs have path counts that do not exceed these values. The mean ranges from 12.5 to 26.9, as shown by the X mark symbols. In each plot, the median and mean are not similar because there are some particular outliers in the set. Under the path number threshold set by ICCBot to prevent path explosion, we still got some EAs with hundreds or thousands of paths. The entire inter-quartile range box represents half (i.e., 75%-25%) of the data in a set, and the height of the inter-quartile range box shows the degree of data concentration. Of the four sets, Set Unclear has the most scattered data, and Set MustIA has the most concentrated data.

7.1.2. Observation

The first interesting finding from Table 8 is that **over half (53%) of the EAs do not process any input Intent attributes**. Figure 13 (a) shows an example of EA that will throw an exception when the app's configuration variable Constants.DEBUG is set to False. Figure 13 (b) shows another example where a main activity is designed to execute without external inputs. External invocation may not be able to change the value of app configurations or internal

Table 8
Information about Comparative Sets of EAs

	#EA	Empty Summary #EA (percent.)	Non-Empty Summary #EA (percent.)	Mean. #Path	Median. #Path
Set MustEA	439	210 (48%)	229 (52%)	26.5	3
Set Unclear	643	284 (44%)	359 (56%)	26.9	5
Set MayIA	1377	766 (56%)	611 (44%)	15.8	3
Set MustIA	375	252 (67%)	123 (33%)	12.5	2
SUM	2,834	1,512 (53%)	1,322 (47%)	20.3	3

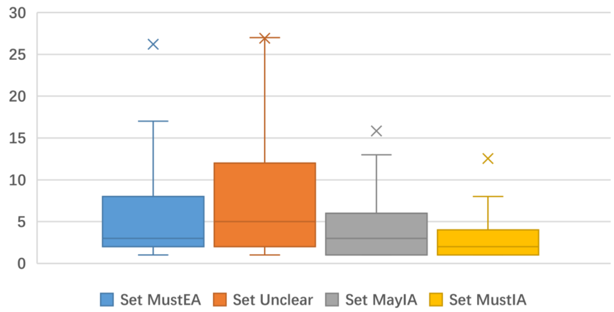


Figure 12: Path Number Distribution

variables that an app maintains during runtime. It may be useless to generate more invocation Intents for these EAs because no further program behavior can be performed.

```java
public class DebugActionsActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (!Constants.DEBUG) {
            throw new UnsupportedOperationException();
        }
        // do something without read intent object
    }
```

(a) Misexposed Debug Activity

```java
public class LoginActivity extends BaseActivity{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        // do something without read intent object
    }
}
```

(b) Login Activity

Figure 13: EAs without Intent-dependent Paths

Then, we compare Set MustEA with Set MustIA. Figure 12 shows that the first difference between Set MustEA and Set MustIA is the number of paths. **Properly exported activities usually have more paths.** On that account, test evaluation of real EAs needs to cover those paths. The *Median. Path* of all sets also indicates that exported activity might have a simple structure. If there is only an intent-dependent execution path, it is very likely that an EA will also not differentiate any inputs. The EA with fewer paths may not provide rich functions.

We further investigate how exported activities retrieve and use Intent attributes by comparing Set MustIA and MustEA. Figure 14 shows the results. In the inner circle, the blue and gray denotes whether EAs will process any Intent attributes or not. For EAs in #Path>0, we further study their usage of two specific Intent attributes: `action` and `extra`. Category “ActExtr”, “ActNoExtr” and “ExtrNoAct” denotes whether an EA calls any `action` and `extra` related API or not.

Actions denote the functionality an activity can provide, and the extra structures pack complex external data. The exposure mode “ActExtr” indicates the EAs that retrieve both attributes in their implementations. Comparing Set MustEA and MustIA in the #path>0 section, the ratio of activities using `action` as well as `extra` attributes (represented in yellow) varies a lot, which is 48% (25% / 52%) on Set MustEA but only 6% (2% / 33%) on Set MustIA.

Table 9
Features of EA Implementation

Category	Feature	Description
Implementation	`noIntent`	does not retrieve intents in any execution path
	`exceedT`	contains number of paths exceed a threshold ϵ
	`useExtra`	uses extra in at least one execution path
	`useAction`	uses action in at least one execution path

It shows that **the attribute `action` representing functionality and the attribute `extra` carrying functionality-specific data are used together to enrich the exposed interface of an app.**

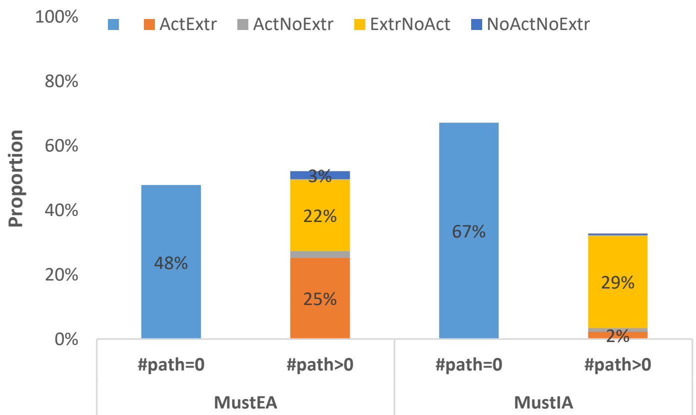


Figure 14: Intent Attribute Handling Comparison

7.2. Complexity Analysis

Through observation, we have defined Intent-Handling Complexity (IHC) as a means of describing whether an EA will provide rich functionality to external invocations. IHC is a binary classification (HighIHC and LowIHC) based on an EA's function summary, which is measured by the number of paths and whether action and extra attributes are retrieved. Table 9 lists the features/elements used for classification, which are divided into two categories.

- The first two features represent the richness of intent-dependent paths, by comparing $|\text{summaryMap}[\text{func\_}\{entry\}]|$ with 0 and ϵ;
- The last two features represent the usage of intent attributes, by retrieving triples whose *param* starts with "extra_" or equals "action" in a function summary.

Then, an EA implementation is encoded as a vector ea_i (i for implementation), where each element $f \in \{0, 1\}$ represents a feature in Table 9. We classify an EA into HighIHC or LowIHC classes, and assign $t = 1$ to significantly reduce the number of tests for LowIHC EAs, and assign $t = 3$ to HighIHC EAs. To be conservative when assigning $t = 1$ to EAs, we follow the priorities and rules shown in Table 10. An EA is classified as LowIHC only if the number of paths does not exceed ϵ and the EA does not use operations and extra attributes.

In the example in Section 3, FooActivity is classified as (MustEA, HighIHC). The classification results in $t^+ = \{(\{action, extra\_drink, extra\_haveDrink\}, t = 3), (\{action, extra\_food, extra\_food\_cake, extra\_haveDrink\}, t = 3)\}$.

8. Evaluation

To evaluate the effectiveness of our approach, we have implemented the proposed approach as a tool called ExaDroid. We conducted experiments with ExaDroid to answer the following research questions:

Table 10
The Classification Conditions of EA Implementation

Class	Pr	Condition
HighIHC	1	`useAction` and `useExtra`
	2	`exceedT`
LowIHC	0	`noIntent`
	3	not `exceedT`

- **RQ1 (Misexposed EA Behavior):** Is there a difference in the behavior of EAs that ExDroid classifies as misexported and those correctly exported?
- **RQ2 (Classification Distribution):** What is the distribution of misexposed EAs and the Intent-handling complexity of EAs? Do the results vary in different datasets?
- **RQ3 (Testing Effectiveness):** How effective is ExaDroid$_{ct}$ in terms of detecting bugs with fewer test cases? How do different strategies perform?

All of our static analysis is performed on a machine with an Intel(R) Xeon(R) E5-2680 v4 CPU @ 2.40 GHz, 256G RAM memory, and Ubuntu 20.04 operating system with OpenJDK 9. For dynamic testing, we use an emulator LDPlayer(64) 4.0.83 with a 4 Core CPU, with 6144MB RAM memory, and Android 7.1.2 operating system.

Benchmark. To evaluate our approach, we consider BenchFdroid from an existing work on ICC resolution evaluation [53]. It includes 31 open-source apps in F-droid, ranging in size from 1M to 93M, with an average of 1,010 GitHub stars. We successfully ran ExaDroid on 30 of the 31 applications in this benchmark. The failure case observed in the app *Conversations* specifically pertains to the ICC resolution tool *ICCBot*. It is important to note that the experimental data presented below are derived from the remaining 30 applications, as the failed app named *Conversations* was excluded due to the issue with *ICCBot*. Finally, we performed combinatorial testing on 135 EAs across these 30 apps.

8.1. Implementation

Hyperparameters. When implementing the two rules of mayIA, certain thresholds were set, including the number of EAs in an app and the ratio. For the coarse-grained rule `highRatio`, we identified apps that have more than 50 EAs and a ratio of EAs larger than 0.4. For rule `similar`, we only detected apps that have more than 30 EAs and used thresholds ranging from 0.5 to 0.7 for different exposure modes. For rule `exceedT`, we chose the path number threshold ϵ to be 3, which is the Median. #Path value in Table 8.

We set hyperparameters with conservative values to more accurately identify misexposed and low-complexity activities, and to avoid missed defects caused by setting $t = 1$ to generate fewer test cases. For instance, rule `highRatio` is extracted from Set AR, whose comparative dataset (Set MD) consists of apps that have no more than 50 EAs (as shown in Figure 7). Therefore, this rule only applies to apps that have more than 50 EAs. These hyperparameters are user-configurable. If users want to find more possible misexposures or to reduce the number of test cases more aggressively, they can adjust by reducing these thresholds.

Test Case Generation and Execution. We utilize ACTS [60] for generating combinatorial test cases. The in-parameter-order-general strategy [26] adopted by ACTS can be described as follows: for a testing model with t or more parameters, where t denotes the size of combinations to cover, the strategy builds a t-way test set for the first t parameters, extends the test set for the first $t+1$ parameters, and then continues to extend the test set until the coverage goal is achieved and all the parameters are included.

To execute the generated test cases, we develop a test bridge app, which is installed on the emulator. Each test case from the combinatorial testing model is transformed into a caller intent. We did not choose the widely adopted adb-form command for test execution because it has limited capability in carrying parameters. If a test case contains any `Java` object, such as `Bundle` or `ArrayList` object, it cannot be sent through adb. Instead, within the test bridge app, intents with richer types and structures can be constructed through the native API. For each `extra` field, we create objects according to its type. For bundle type, we reconstruct the proper data structure. The caller intents will not be affected by special characters.

Table 11
The Performance of ExaDroid on BenchFdroid

Cat.	#Test	#P	#F	#EA	$\#EA_{1-F}$	$\#EA_{A-F}$	#Test/#EA
MustIA	208	128 (62%)	80 (38%)	45	23 (51%)	12 (27%)	4.6
Unclear	1099	695 (63%)	272 (25%)	54	21 (39%)	4 (7%)	20.4
MustEA	609	531 (87%)	78 (13%)	36	11 (31%)	1 (3%)	16.9
All	1916	1354 (71%)	430 (22%)	135	55 (41%)	17 (13%)	14.2

The test bridge app not only generates intents but also executes them. It is connected with test scripts in ExaDroid through a socket. After the test case transformation is complete, all tests will be automatically executed by calling the `startActivity` function to start the target component. Test results are automatically recorded and evaluated.

Test Result Identification. ExaDroid determines the execution results of a test caseby monitoring the foreground activity through the dumpsys command and analyzing the execution logs through the logcat command. Two types of test execution results are considered as failures (*Fail*): (1) when the called activity throws an exception, which is captured by logcat, and crashes, or (2) when the called activity returns to the test bridge app or the android launcher, and logcat may not capture the stack trace. Otherwise, the test passes (*Pass*): the foreground activity either remains the target component for several seconds, or the EA jumps to other activities other than the test bridge and launcher.

8.2. RQ1: Misexposed EA Behavior

Table 11 shows the test execution and misexposure prediction results on BenchFdroid. The table is summarized according to misexposure identification categories (column Cat.). Column #Test, #P, and #F denote the number and ratios of generated, passed, and failed test cases. Column #EA denotes the number of identified EAs. Column $\#EA_{1-F}$ and $\#EA_{A-F}$ represent EAs that contain at least one failed test and EAs that fail for all generated test cases, respectively. Column #Test/#EA shows the average number of test cases generated for each EA. Cells with aqua and gray color in the failure-related columns highlight the highest and lowest values of ratios, respectively. On this dataset, no activity is identified as MayIA.

We observe that **EAs identified by ExaDroid as misexposed (MustIA) are more vulnerable**. The MustIA category has the highest ratios in columns #F and $\#EA_{1-F}$, and approximately 27% of all EAs in this category crashed completely for all external invocations, as shown in column $\#EA_{A-F}$. It is important to note that EAs that crash for any caller intent may not necessarily be the ones the developer intended to expose. On the other hand, **EAs identified by ExaDroid as properly exposed (MustEA) are more robust**. The MustEA category has the lowest ratios of failed tests, EAs that fail at least once, and EAs that fail for all external invocations. The Unclear category has the highest ratios of failed tests and EAs that fail at least once. The sum of values in #P and #F is not equal to 1, because ExaDroid generated 135 tests for component `org.inaturalist.android.ObservationEditor`, but only 3 of them are executable, and the others require the system camera app, which is not supported by the emulator.

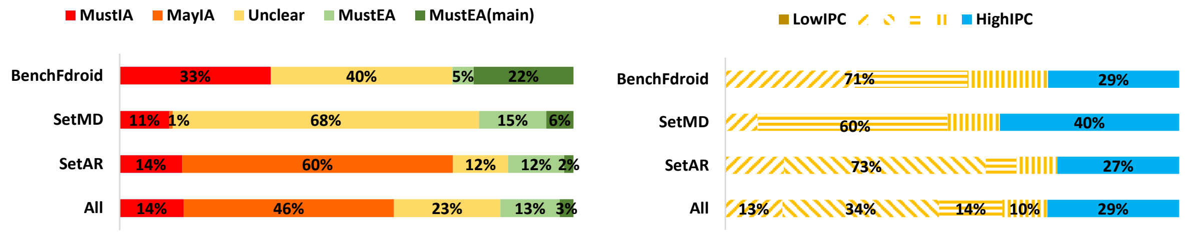
In summary, some activities are misexposed by developers and have apparent testing behavior, i.e., no caller can successfully invoke them. Our misexposure prediction method can distinguish these components.

8.3. RQ2: EA Distribution

We conducted a static analysis of EAs in apps from three sets: SetMD (Most Downloads), SetAR (Abnormal Ratio), and BenchFdroid. Figure 15 shows the results of our misexposure prediction from EA declarations and Intent-handling complexity analysis from EA implementations. Note that, as in Section 7.1.1, we excluded activities in obfuscated apps. The statistics of misexposure classification rules and complexity analysis rules are shown in Figure 16 and Table 12, respectively.

Our comparison of misexposed EA ratios in the three sets conforms to our assumptions. In Set AR, 14% of EAs are classified as MustIA with high certainty. Overall, about three-quarters (74%) of EAs are suspected to be IAs, whose exposure may not be suggested. In Set MD, only 12% of EAs are suspected to be IAs. In BenchFdroid, the proportion of such EAs is 33%, somewhere in between.

To gain an intuitive understanding of the rules by which we classify EAs as MustIA (rule 1-3 in Table 7) or MayIA (rule 7 and 8), we counted the hitting number of each rule on the three sets. Figure 16 shows the hit ratio calculated by the formula where the numerator is the number of hits for a rule, and the denominator is the number of activities classified as MustIA and MayIA. We observed that rule 7 and 8 of MayIA hit top for Set AR, which illustrates the high proportion of MayIA in Set AR. In contrast, BenchFdroid does not contain any MayIA nor the use of rule 7



(a) Misexposure Distribution

(b) EA Complexity Distribution

Figure 15: Static Analysis Results on Three Datasets

Table 12
Statistics of Each Complexity Analysis Rule

Rule	HighIHC		LowIHC	
	1 useAction and useExtra	2 exceedT	0 noIntent	3 not exceedT
SetAR	193 (8%)	424 (18%)	1290 (56%)	392 (17%)
SetMD	82 (15%)	131 (24%)	223 (41%)	103 (19%)
BenchFdroid	21 (16%)	18 (13%)	59 (44%)	37 (27%)
All	296 (10%)	573 (19%)	1572 (53%)	532 (18%)

and 8, mainly because open-source apps contain fewer EAs. Another observation is that for the MustIA category, rule 2 constitutes the majority of EAs declared in ExTrue mode (`android:exported=true`), while rule 1 only hits the NoEx mode. Both rules indicate that the EA will not or is unlikely to be called implicitly. As ExTrue is the suggested mode in Android 12, it is expected that the proportion of rule 2 might increase in more and more updated apps. Since ExTrue is a proposed mode in Android 12, users can choose a less conservative strategy to have rule 1 also apply to EAs declared in this mode. This can be achieved by removing "ifTrue and not exTrue" in the condition of rule 1 in Table 7.

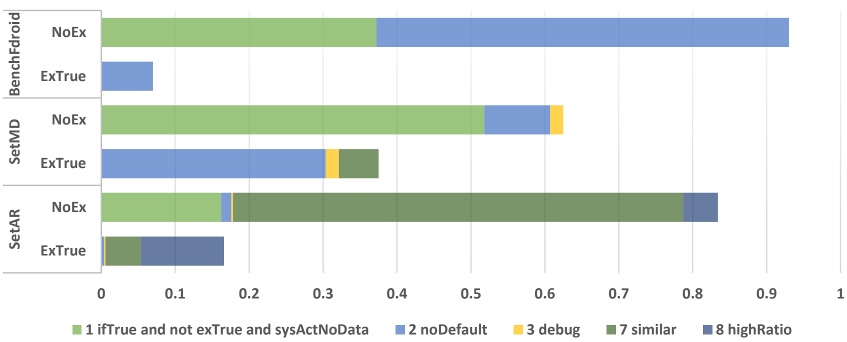


Figure 16: Statistics of Each Misexposure Prediction Rule for MustIA and MayIA Category

The EA complexity distribution in BenchFdroid is consistent with our observations in the EAs of Set MD and AR, as shown in Figure 15. The vast majority of EA implementations for each set have low Intent-handling complexity, with LowIHC ratios ranging from 60% to 73%. Table 12 indicates that more than half (53%) of EAs do not retrieve nor process any Intent attributes from external calls.

We further investigate the relationship between EA complexity and misexposure classification. **Although each misexposure category contains EAs that belong to LowIHC, the conditional probabilities of LowIHC show a correlation with the misexposure category.** The shape fills of the legend in Figure 15 represent the four classification categories in misexposure prediction. The EAs that constitute LowIHC come from all classes. We then compute the conditional probability under each class. The likelihood of an EA having low complexity based on the fact that the

Table 13
Error Triggering Comparison

Tool	#App	Duration (minutes)	#Error
ExaDroid	30	(16.4, 2.7)	100
Fax	26	(3.8, 60)	58
IntentFuzzer	30	-	24

EA belongs to MustIA is $P(\text{LowIHC}|\text{MustIA}) = 13\%/14\% = 0.93$, where the values 14% and 13% come from All in Figure 15. The conditional probability is greater than $P(\text{LowIHC}|\text{MustEA}) = 10\%/(13\% + 2\%) = 0.66$.

In conclusion, activities that are exposed by mistake or have low complexity in Intent handling are widely present in different datasets. Therefore, **although the Android market exposes many components, the interaction and cooperation among applications are not sufficient**.

8.4. RQ3: Testing Effectiveness

In this section, we compare ExaDroid with related works on BenchFdroid. The related works employ two approaches: symbolic execution and fuzzing. For the first type, Fax [52] introduces the concept of constructing an activity testing model through symbolic execution, which leads to the successful launch of a larger number of activities. In comparison to state-of-the-art Android testing tools, Fax significantly improves testing coverage. For the second type, IntentFuzzer [36] is a fuzzing-based tool that is readily available and widely used by Android developers. It generates intents with null values and serializable data.

Table 13 provides an overview of the error triggering capabilities of each tool examined in this study. Column #App shows the number of successfully analyzed apps, Duration shows the average time cost for static analysis and dynamic execution on each app, and #Error indicates the number of unique errors being triggered by generated test cases. Whenever an error-level exception is encountered, we record the runtime log information and collect the stack trace. Uncaught exceptions have the potential to crash the application, resulting in a negative user experience. Therefore, we use the number of uncaught stack traces to represent the number of errors.

The results show that **ExaDroid demonstrates a significant capability to trigger a substantial number of unique crashes**. It triggers 100 errors, which is 42 more errors than Fax. It is also worth noting that the static analysis of the app *OsmAnd* required 269 minutes, leading to an increase in the overall average duration. As for Fax, it failed to analyze two apps and to instrument another two apps. Its dynamic execution threshold is one hour by default. Out of the 58 errors identified by Fax, ExaDroid successfully reproduces 48 of them. Upon manual analysis of the errors that ExaDroid failed to reproduce, we found that five of them were attributed to delayed crashes because Fax waits 10 seconds but ExaDroid will return in 4 seconds. Since Fax also employs random mutation strategy, we run the tool twice to obtain the average, but its performance is stable. IntentFuzzer triggers 24 errors after removing those caused by serializable input. Since it does not distinguish EAs from IAs and it requires manual operation on the simulator, so there is no running time statistics. It can be seen that some EAs throw exceptions for simple null inputs.

Besides, it can be observed that **ExaDroid is capable of effectively testing an EA with only a dozen or so test cases**. As shown in Table 11, ExaDroid generates a total of 1916 test cases for 135 EAs, with an average cost of 14.2 intents per EA. Table 14 illustrates the test reducing effectiveness of ExaDroid, based on static analysis results and testing behaviors. The value A-P (or A-F) represent that an EA pass (or fail) in all generated test cases. We mark the values in the last column that exceed the mean value of 14.2 in blue. It can be observed that MustIA and LowIHC have fewer blue markers compared to other categories. By setting the testing strength $t = 1$, ExaDroid generates 430 test cases for 102 EAs of MustIA or LowIHC, which account for the majority (76%) of all EAs in BenchFdroid, thus reducing the average number of test cases. However, the question remains whether these small test suites fully trigger all possible behaviors of the EA. Out of the 102 EAs, 68 belong to A-P, 16 belong to A-F, and only 18 have both pass and fail behavior. By experimentally assigning $t = 3$ to all EAs, we find that 4 A-P EAs now have failed test cases, but it brings 4783 more test cases, as shown in Table 15.

Table 15 presents the results of controlled experiments in which ExaDroid is purposefully configured for selected value-taking strategies (or testing strength settings). We compare the Base value-taking strategy with other user-customizable strategies described in Section 5.1.2, as well as compare the testing strength setting that adaptively assigns $t = 1$ or $t = 3$ based on EA classification with fixed strength settings. It is evident that keeping the testing strength setting the same, making the value-taking strategy additionally consider manifest, boundary, or random values brings

Table 14
Testing Behavior of BenchFdroid

Misexposure	Complexity	TestBehavior	#EA	#Test	#Test/#EA
MustIA	LowIHC	A-P	19	59	3.1
		A-F	12	36	3.0
		other	8	26	3.3
	HighIHC	A-P	3	36	12
		A-F	-	-	-
		other	3	51	17
Unclear	LowIHC	A-P	25	100	4.0
		A-F	3	12	4.0
		other	5	24	4.8
	HighIHC	A-P	7	262	34.7
		A-F	1	16	16.0
		other	13	685	52.7
MustEA	LowIHC	A-P	21	69	3.3
		A-F	1	6	6.0
		other	2	11	5.5
	HighIHC	A-P	4	141	35.3
		A-F	-	-	-
		other	8	382	47.8

Table 15
Controlled Experiments

	Base, Adaptive Strength	Value-Taking Strategy				Strength Strategy	
		Base+ Manifest	Base+ PresetBound	Base+ Random	Base+ ALL	$t = 1$	$t = 3$
#Error	100	106	106	104	114	77	118
#Test	1916	2971 (1.6×)	3630 (1.9×)	8206 (4.3×)	18264 (9.5×)	552 (0.3×)	6699 (3.5×)
Execution Time (hh:mm:ss)	01:22:04	02:24:21	02:53:01	05:51:55	13:26:22	00:27:11	05:01:52

many test cases but few new errors. In particular, only considering random values may result in many useless tests. For a given number of parameters in a CT model, the size of a combinatorial t-way test suite increases exponentially with the number of values that each parameter can take [22]. The size of a combinatorial t-way test suite also increases rapidly as t increases. The table shows the number of tests and error detection capabilities for fixing $t = 1$ or $t = 3$. The adaptive strength setting reduces the number of test cases for a fixed $t = 3$ and detects more bugs than $t = 1$. When t is increased to 4, the numbers of test cases for Base and Base+All strategies are 19488 and 1,006,040, respectively.

Based on the generated tests and execution results for a fixed $t = 3$ strength, we analyzed the root causes of 118 errors. Combinatorial testing researchers believe that a failed test run is caused by a specific combination, known as a failure-inducing interaction (FIC) [63]. The number of conditions required to trigger a failure is called the FIC size. One tester follows the practice [39] of manually analyzing FICs based on execution results. The tester distinguishes between passed and failed runs, finds combinations that are only covered by failed runs, and progressively excludes irrelevant conditions from them to find the smallest FIC sizes. For EAs that contain multiple types of errors/unique stack traces, we use whether one type of error occurs as the criterion for distinguishing between passed and failed runs. Initially, the tester was unable to analyze 13 errors in 13 EAs because all test cases failed with one error type and there were no passed runs. Another 22 errors in 5 EAs were also ruled out because the number of failed runs was insufficient for analysis or the execution results were unstable due to an implementation error of ExaDroid. Finally, we obtained 90 FICs for 83 errors in 41 EAs. Some errors can have more than one FIC. Figure 17 shows the results of the 90 identified FICs. The abscissa represents the FIC size, the main ordinate represents the number of FICs, and the sub-ordinate represents the cumulative distribution. The FIC size is usually less than or equal to 6, and the distribution of FIC appears to follow a power law, similar to the findings from the existing CT empirical research [23]. We found that 32% of the errors are triggered by only a single parameter value, 88% by three-way combinations, and 98% by four-way combinations.

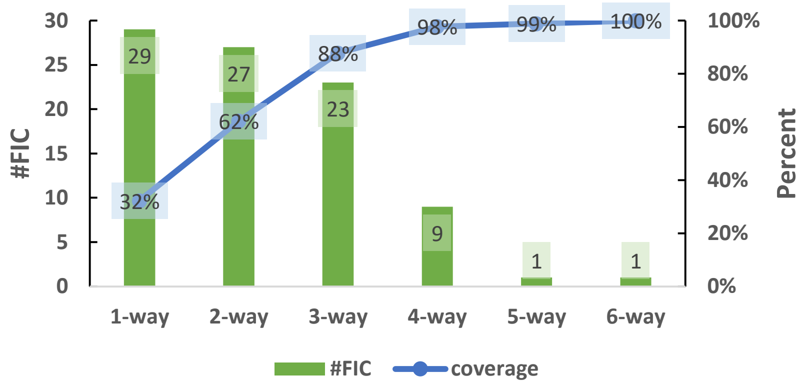


Figure 17: Failure Triggering Interactions, Cumulative Distribution

Experiments have demonstrated that ExaDroid has the ability to trigger many unique crashes with fewer test cases, utilizing variable-strength combinatorial testing strategies.

9. Threats to Validity

Construct validity This paper's definition of three concepts may lead to certain risks. The first concept is the activity exposure modes. Older Android versions, which are more widely used, allow exporting activities in two ways, while Android 12 only allows one way. Since the former occupies more market share, our misexposure reasoning analyzes both ExTrue and NoEx exposure modes. As shown in Figure 16, there are misexported activities under both modes. The second concept is misexported activity. Usually, only the developer can specify whether an EA is intended to be exported or not. Instead, our definition is based on the likelihood that an EA is invoked and whether the exposure is due to poor programming practices by developers. Additionally, a tester dynamically executed activities that fit the definition and confirmed that they usually cannot provide functionality to external invocations. The third concept is the unique error-level stack trace. The definition in Section 8.4 is accepted in the Android testing community. Based on this definition, we compare the results of three testing tools.

Internal Validity The validity of the study is at risk due to various factors such as the manual analysis phase, hyperparameters of the implementation, static analysis module, and the experimental dataset. To infer EA classification rules, comparative datasets are constructed and combined with manual analysis. The selection metrics for constructing SetAR and SetMD were verified on 10 apps by a tester who launched each EA and examined its bytecode. The accuracy of misexposure classification for constructing Set MustIA and Set MustEA was validated in previous work [51] by comparing it with manual classification on 50 randomly selected apps by three testers. The distribution of studied activities suggests that there is no data source imbalance. However, the identification results are limited by the Caller Intent Dataset and the hyperparameters. The dataset is collected across three repositories with a variety of categories, but it is currently outdated due to the time-consuming extraction of caller intents from each app. The values of hyperparameters in this paper are conservative, which reduces the probability of classifying an EA as MustIA or LowIHC, leading to more tests. Users can lower the thresholds or adjust the priority to be more aggressive in finding misexported EAs and reducing the number of test cases. The extracted function summary influences the test modeling as well as the test generation. However, it has been found that the static analyzer does not model the API `equalsIgnoreCase()` for string items, affecting the accuracy of the complexity calculation. Lastly, the limited scale of the apps used for experimentation may affect the answers to RQs. To mitigate this, the study follows related works [52, 30] and adopts the widely used BenchFdroid.

External Validity The external validity of our research is mainly determined by the scope of our study. In typical Android applications, there are four types of components that can be used. While our study focuses only on the most commonly used component, i.e., activities, our proposed method can be easily adapted for other components as well. These components are exported in similar ways, and we believe that the misexposure of other components has similar characteristics. We plan to investigate them in our future studies. Furthermore, although we have studied and tested exported activities in this paper, the complexity analysis and combinatorial testing strategy can also be applied to internal activities.

10. Related Work

ICC Attack Detection. The design of ICC has its limitations, which may cause bugs or security flaws. A study by Ahmad et al. [1] discussed the challenges it brings to Android development. Chin et al. [9] provide the tool ComDroid to describe application communication vulnerabilities caused by the misunderstanding of the intent passing system, such as unauthorized intent receipt and intent spoofing. The research [20] proposes an iterative test generation approach to detect the ICC vulnerabilities (e.g., XSS, SQL injection, etc.) of Android apps. In each iteration, they recover the custom fields (variables) of intent by instrumenting the APIs that are used to read such fields and monitoring the app execution. Bagheri et al. [5] implement the tool Covert that can detect the permission leakage caused by the lack of permission requirements of exposed components. They first perform static analysis techniques to obtain the model of program behavior, then use the alloy language (an object modeling notation) to model the combination of apps, and finally perform the formal analysis technique to verify the model.

In addition to a wide variety of approaches to identifying vulnerabilities, an exploit generation tool LetterBom [18] based on a path-sensitive static analysis (using symbolic execution) is provided, which can be used to reduce the number of false positives in vulnerability detection. To determine whether the vulnerability really exists, Zhou et al. [64] also propose a path-sensitive symbolic execution-based static analysis as well as a testing technique to reduce false positives. They detect the capability leak for illegal goals and utilize CFG reduction and CG search optimization to optimize symbolic execution.

A more recent research [44] uncovers an atypical ICC mechanism. It finds that a component with Android objects (e.g., PendingIntent or IntentSender) can be invoked through some methods whose role is not primarily to start a component but to perform some action, such as set an alarm or send an SMS. The vulnerability that PendingIntent could bring has been studied by GroSS et al. in [19]. Besides ICC, there are other inter-app code invocations for different reuse scenarios. Gao et al. [17] expose the general workflow that enables app developers to access and invoke functionality (either entire Java classes, methods, or object fields) implemented in other apps using official Android APIs. They showed a case that video database can be accessed even with security guards. They propose the tool DICIDer for detecting direct inter-app code invocations in apps.

In our work, we pay more attention to another aspect, i.e., detecting whether activities should be exposed or not.

ICC Resolution. ICC mechanism introduces implicit control flow, which makes generating precise call graphs and control flow graphs very difficult. In recent years, several researchers have aimed to expose such implicit transitions through intent analysis [42, 41, 28]. Octeau et al. provided the tool Epicc [42] for obtaining ICC methods and their parameters. They also provided IC3 [41] which modeled ICC messages with the proposed COAL language and implemented the associated solver that performs string analysis to figure out the ICC specification in Android apps. Based on Epicc and IC3, Li et al. [29] developed IccTA, a static analysis tool for detecting inter-component privacy leaks in Android apps. The links between components are detected by code instrumentation and static analysis techniques. Raicc [44] by Samhi et al. complements IccTA with atypical inter-component communication methods. Yan et al. resolved the component transitions connected by Android fragments, and provided context-sensitive tracking of data transfer among methods calls through ICCBot [54]. In this work, we employ ICCBot to obtain the object summaries for EAs to empirically study the processing of Intent attributes. In our previous version, we adopted intra-procedural reaching definition analysis to maintain the dataset of intents sent by caller apps and get the target variable related du-chains in each method. Thus, we can track the assignments of each ICC field as well as the key declaration and value of `extra` data items. Our caller dataset can be used not only for misexposure prediction but also for other ICC resolution tools to find vulnerable ICC links.

Intent Fuzzing. Fuzzing is the most widely adopted method for discovering intent vulnerabilities. Maji et al. [33] presented the first empirical evaluation of the robustness of ICC in Android through fuzzing methodology. They used straightforward strategies, such as “Semi-valid/Blank/Random Action and Data” to generate fuzzing test cases. However, the inherent weakness of fuzzing is that the number of test cases is very large. In their experiment, around 9,000 intents were sent to test an activity. Some works rely on static analysis to avoid aimless exploration with invalid parameters. Null Intent fuzzing and randomized approaches are applied to generate Intents, not only for cross-app communication [25], but also for cross-platform communications, e.g., Android Wear OS [59]. The declaration in manifest files enables many researchers to improve the fuzzing strategy [33, 55, 58]. However, according to [52], only collecting the declaration values is not sufficient for activity modeling. There are mismatches between the attribute declaration in manifest files and its usage in Java codes. The actually used attribute may not be related to implicit invocation and may not be declared. The approach we propose in this paper takes advantage of basic attribute values as well as `extra` types and structures in code. But the tool also enables users to configure the source of the value for various usage scenarios. Sasnauskas et al. [45] built a tool on Monkey [12] and FlowDroid [4]. Similar to our work, they extract key-type pairs of the `extra` attribute and then fuzz on top of empty intent templates. Instead, we define the combination coverage of Intent attributes to avoid endless fuzzing. It is a pity that all those tools do not consider allocating reasonable testing resources for different components. This paper shows in empirical research that it does not make sense to perform extensive fuzzing on simple components, mainly MistIA category. The misexposed components should be identified, preferably with their exposure status turned off.

Android GUI Testing. Due to the event-based nature of Android apps, test cases take the form of GUI events. To conduct GUI testing, automatic exploration approaches have been proposed, including random exploration [12, 61, 32], model-based exploration [57, 38, 56], and systematic exploration [3, 21]. These approaches aim to cover more components or transitions. Another approach is to adjust the single-entry testing explored from the default entry point to simplify the calling context construction of components. Wang et al. [46] propose test case decomposition and re-combination, while TimeMachine in [16] utilizes test state capture and resume. Most relevant to the topic of this paper is the multi-entry testing strategy in [52], which changes the `exported` attribute of an IA to directly invoke the activity for testing.

Combinatorial Testing Combinatorial testing has become an active field in recent year, with the major trends being the minimization of test set sizes for a given combinatorial criterion. Optimization algorithms proposed in [62, 2] use variable-strength combinatorial test generation. Another trend is the application of combinatorial testing in various fields, including Android testing. For example, TrimDroid [37] is an approach that statically extracts dependencies among widgets to reduce the number of combinations in GUI testing. Prefest [31] proposes the dependency of test cases to Preference, the setting options provided by Android, and uses both static and dynamic analysis to configure preferences for existing test cases. These works are all focused on Android GUI testing. However, our goal is not GUI testing for application-wide component coverage, but rather robustness testing for specific components. Since an EA is an additional entry point to the app, we avoid the difficulty of generating test sequences to reach an activity. Additionally, our use of combinatorial testing is more granular, taking advantage of the variable-strengths in combinatorial testing.

11. Conclusion

In this paper, we have investigated two characteristics of exported activities: the likelihood of misexposure and the complexity of Intent processing. Exported activities are vulnerable to malicious ICC attacks and thus require exhaustive testing. However, existing testing efforts that are unaware of such exported activity characteristics can lead to resource wastage. Therefore, the key challenge lies in identifying the misexposure and computing the complexity. With the help of static analysis, we have identified typical misexposed activities from tens of thousands of real-world apps. Through comparative analysis, we have extracted rules to automatically classify an EA into four misexposure categories and two complexity categories based on static analysis results. Then, based on the classification results, we have designed various strategies to improve the efficiency of dynamic combinatorial testing of exported activities to discover exposure vulnerabilities. We have implemented a tool called ExaDroid, and experiments on real-world apps show that it can reasonably allocate testing resources to different exported activities and can effectively trigger unique crashes with as few test cases as possible.

ExaDroid can improve the quality and robustness of Android applications by reporting exported activity characteristics and finding bugs. In the future, we will improve ExaDroid by using more concise static analysis and better value strategies. We will also conduct further studies on how to locate the failure-triggering combinations based on the test results of the test suite. This is useful for developers to fix bugs. We hope this tool could be widely used to help reduce exposure vulnerabilities in the Android market and enable richer interactions between components.

References

[1] Ahmad, W., Kästner, C., Sunshine, J., Aldrich, J., 2016. Inter-app communication in Android: developer challenges, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, pp. 177–188.

[2] Ahmed, B.S., Zamli, K.Z., 2011. A variable strength interaction test suites generation strategy using particle swarm optimization. Journal of Systems and Software 84, 2171–2185.

[3] Anand, S., Naik, M., Harrold, M.J., Yang, H., 2012. Automated concolic testing of smartphone apps, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 1–11.

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, in: Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 29:1–29:11.

[5] Bagheri, H., Sadeghi, A., Garcia, J., Malek, S., 2015. COVERT: compositional analysis of Android inter-app permission leakage. IEEE Transactions on Software Engineering 41, 866–886.

[6] Bodden, E., 2022. Soot. http://www.bodden.de/2008/09/22/soot-intra.

[7] Borazjany, M.N., Ghandehari, L.S.G., Lei, Y., Kacker, R., Kuhn, R., 2013. An input space modeling methodology for combinatorial testing, in: IEEE International Conference on Software Testing, Verification and Validation, pp. 372–381.

[8] Breunig, M.M., Kriegel, H., Ng, R.T., Sander, J., 2000. LOF: identifying density-based local outliers, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 93–104.

[9] Chin, E., Felt, A.P., Greenwood, K., Wagner, D.A., 2011. Analyzing inter-application communication in Android, in: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), pp. 239–252.

[10] Clocksin, W.F., Mellish, C.S., 2003. Programming in PROLOG. Springer Science & Business Media.

[11] Developers, A., 2021a. Behavior changes: Apps targeting android 12. https://developer.android.com/about/versions/12/behavior-changes-12.

[12] Developers, A., 2021b. UI/application exerciser monkey. http://developer.android.com/tools/help/monkey.html.

[13] Developers, A., 2022a. activity. https://developer.android.com/guide/topics/manifest/activity-element.html.

[14] Developers, A., 2022b. <category>. https://developer.android.com/guide/topics/manifest/category-element.

[15] Developers, A., 2022c. Intents and Intent Filters . https://developer.android.com/guide/components/intents-filters.html.

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

[17] Gao, J., Li, L., Kong, P., Bissyandé, T.F., Klein, J., 2020. Borrowing your enemys arrows: the case of code reuse in android via direct inter-app code invocation, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 939–951.

[18] Garcia, J., Hammad, M., Ghorbani, N., Malek, S., 2017. Automatic generation of inter-component communication exploits for Android applications, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pp. 661–671.

[19] Groß, S., Tiwari, A., Hammer, C., 2018. Pianalyzer: A precise approach for pendingintent vulnerability analysis, in: European Symposium on Research in Computer Security, Springer. pp. 41–59.

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

[21] Jensen, C.S., Prasad, M.R., Møller, A., 2013. Automated testing with targeted event sequence generation, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 67–77.

[22] Kuhn, D.R., Bryce, R., Duan, F., Ghandehari, L.S., Lei, Y., Kacker, R.N., 2015. Combinatorial testing: Theory and practice. Advances in Computers 99, 1–66.

[23] Kuhn, D.R., Wallace, D.R., Gallo, A.M., 2004. Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30, 418–421.

[24] Kuhn, R., Lei, Y., Kacker, R., 2008. Practical combinatorial testing: Beyond pairwise. It Professional 10, 19–23.

[25] Labs, M., 2021. Drozer. https://labs.mwrinfosecurity.com/tools/drozer/.

[26] Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2007. Ipog: A general strategy for t-way software testing, in: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS), IEEE. pp. 549–556.

[27] Li, A., 2022. Android 12 distribution numbers. https://9to5google.com/2022/08/12/android-12-distribution-numbers/.

[28] Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., 2015a. Apkcombiner: Combining multiple Android apps to support inter-app analysis, in: Proceedings of 30th International Conference on ICT Systems Security and Privacy Protection, pp. 513–527.

[29] Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P., 2015b. IccTA: Detecting inter-component privacy leaks in Android apps, in: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, pp. 280–291.

[30] Liu, A., Guo, C., Dong, N., Wang, Y., Xu, J., 2022. Dalt: Deep activity launching test via intent-constraint extraction, in: 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE), pp. 482–493.

[31] Lu, Y., Pan, M., Zhai, J., Zhang, T., Li, X., 2019. Preference-wise testing for android applications, in: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 268–278.

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

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

[34] Mathur, A.P., 2013. Foundations of software testing. Pearson Education India.

[35] Merritt, D., 2012. Building expert systems in Prolog. Springer Science & Business Media.

[36] MindMac, 2017. IntentFuzzer. https://github.com/MindMac/IntentFuzzer.

[37] Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S., 2016a. Reducing combinatorics in gui testing of android applications, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE. pp. 559–570.

[38] Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S., 2016b. Reducing combinatorics in gui testing of android applications, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE. pp. 559–570.

[39] Nie, C., Leung, H., 2011. The minimal failure-causing schema of combinatorial testing. ACM Trans. Softw. Eng. Methodol. 20, 15:1–15:38.

[40] Nie, C., Wu, H., Niu, X., Kuo, F., Leung, H.K.N., Colbourn, C.J., 2015. Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. Information & Software Technology 62, 198–213.

[41] Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P., 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis, in: Proceedings of the 37th International Conference on Software Engineering, pp. 77–88.

[42] Octeau, D., McDaniel, P.D., Jha, S., Bartel, A., Bodden, E., Klein, J., Traon, Y.L., 2013. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis, in: Proceedings of the 22th USENIX Security Symposium, pp. 543–558.

[43] Sabharwal, S., Aggarwal, M., 2017. A novel approach for deriving interactions for combinatorial testing. Engineering Science and Technology, an International Journal 20, 59–71.

[44] Samhi, J., Bartel, A., Bissyandé, T.F., Klein, J., 2021. Raicc: Revealing atypical inter-component communication in android apps, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE. pp. 1398–1409.

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

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

[47] Wang, M., Cui, B., Yan, J., Yan, J., Zhang, J., 2022. String test data generation for java programs, in: Proceedings of the International Symposium on Software Reliability Engineerings, pp. 251–262.

[48] Wang, Z., Nie, C., Xu, B., 2007. Generating combinatorial test suite for interaction relationship, in: International Workshop on Software Quality Assurance, pp. 55–61.

[49] WeChat, 2017. Open SDK. https://open.weixin.qq.com/cgi-bin/showdocument?action=dir_list&t=resource/res_list&verify=1&id=1417751808&token=&lang=en_US.

[50] Wikipedia, 2022. Reaching definition. https://en.wikipedia.org/wiki/Reaching_definition.

[51] Yan, J., Deng, X., Wang, P., Wu, T., Yan, J., Zhang, J., 2018. Characterizing and identifying misexposed activities in android applications, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 691–701.

[52] Yan, J., Liu, H., Pan, L., Yan, J., Zhang, J., Liang, B., 2020. Multiple-entry testing of android applications by constructing activity launching contexts, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE. pp. 457–468.

[53] Yan, J., Zhang, S., Liu, Y., Deng, X., Yan, J., Zhang, J., 2022a. A comprehensive evaluation of android icc resolution techniques, in: The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE).

[54] Yan, J., Zhang, S., Liu, Y., Yan, J., Zhang, J., 2022b. Iccbot: Fragment-aware and context-sensitive icc resolution for Android applications. The 44th International Conference on Software Engineering, (ICSE) Demo Track .

[55] Yang, K., Zhuge, J., Wang, Y., Zhou, L., Duan, H., 2014. Intentfuzzer: detecting capability leaks of Android applications, in: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 531–536.

[56] Yang, S., Wu, H., Zhang, H., Wang, Y., Swaminathan, C., Yan, D., Rountev, A., 2018. Static window transition graphs for android, in: Automated Software Engineering, Springer. pp. 833–873.

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

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

[59] Yi, E.B., Zhang, H., Maji, A.K., Xu, K., Bagchi, S., 2020. Vulcan: Lessons on reliability of wearables through state-aware fuzzing, in: Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, pp. 391–403.

[60] Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R., 2013. Acts: A combinatorial test generation tool, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 370–375.

[61] Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., Xie, T., 2016. Automated test input generation for android: Are we really there yet in an industrial case?, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 987–992.

[62] Zhang, Z., Yan, J., Zhao, Y., Zhang, J., 2014. Generating combinatorial test suite using combinatorial optimization. Journal of Systems and Software 98, 191–207.

[63] Zhang, Z., Zhang, J., 2011. Characterizing failure-causing parameter interactions by adaptive testing, in: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA, pp. 331–341.

[64] Zhou, M., Zeng, F., Zhang, Y., Lv, C., Chen, Z., Chen, G., 2019. Automatic generation of capability leaks' exploits for android applications, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE. pp. 291–295.