COLUMBUS: Android App Testing Through Systematic Callback Exploration

Priyanka Bose*, Dipanjan Das*, Saastha Vasan*, Sebastiano Mariani†, Ilya Grishchenko*, Andrea Continella‡, Antonio Bianchi§, Christopher Kruegel*, and Giovanni Vigna*
*University of California, Santa Barbara {priyanka,dipanjan,saastha,grishchenko,chris,vigna}@cs.ucsb.edu
†VMware, Inc. {smariani}@vmware.com
‡University of Twente {a.continella}@utwente.nl
§Purdue University {antoniob}@purdue.edu

***Abstract*—With the continuous rise in the popularity of Android mobile devices, automated testing of apps has become more important than ever. Android apps are event-driven programs. Unfortunately, generating all possible types of events by interacting with an app's interface is challenging for an automated testing approach. Callback-driven testing eliminates the need for event generation by directly invoking app callbacks. However, existing callback-driven testing techniques assume prior knowledge of Android callbacks, and they rely on a human expert, who is familiar with the Android API, to write *stub* code that prepares callback arguments before invocation. Since the Android API is very large and keeps evolving, prior techniques could only support a small fraction of callbacks present in the Android framework.**

In this work, we introduce COLUMBUS, a callback-driven testing technique that employs two strategies to eliminate the need for human involvement: (i) it automatically identifies callbacks by simultaneously analyzing both the Android framework and the app under test; (ii) it uses a combination of under-constrained symbolic execution (*primitive* arguments), and *type-guided* dynamic heap introspection (*object* arguments) to generate valid and effective inputs. Lastly, COLUMBUS integrates two novel feedback mechanisms—*data dependency* and *crash-guidance*—during testing to increase the likelihood of triggering crashes and maximizing coverage. In our evaluation, COLUMBUS outperforms state-of-the-art model-driven, checkpoint-based, and callback-driven testing tools both in terms of crashes and coverage.

I. INTRODUCTION

Android is the most popular mobile operating system, with 2.8B active users and a global market share of 75% as of 2021 [2]. Android apps cater to diverse users' needs, such as emailing, banking, gaming, *etc.* The Google Play Store, the official Android app market, witnessed enormous growth—it currently hosts 2.9M apps, and more than 100K apps are added every month [1]. In order to provide a smooth user experience, these apps need to be thoroughly tested before developers push them to the market. Modern Android apps use rich user interface (UI) and complex app logic, thus making automated exploration challenging.

Android apps are event-driven programs, *i.e.*, each interaction with the UI of the app generates an event, which drives the app through different states. Therefore, synthesizing a correct sequence of events is essential to efficiently explore the state space of an app. Many prior techniques rely on UI testing frameworks [8], [10], [46], [41], [17], [48], [22] to exercise the app by generating appropriate events. However, a large class of events is widget-specific, and requires multiple user actions to be taken in a specific order at specific UI coordinates. As we explain in Section III, the `onDateChanged` event of the `DatePickerDialog` widget is one such example. Generating such events *deterministically* is challenging for a UI-based testing tool, unless it has been equipped with the knowledge of how to generate all the correct events. Given the variety of the Android widgets, and the different types of events they support, this is non-trivial. To address this, callback-driven approaches [39] leverage the fact that when a UI event is triggered, the associated *event handler*, also known as *callback*, is executed. Callbacks are the methods in the app typically invoked by the Android framework on the occurrence of an event, *e.g.*, `click` on a widget. Callback-driven techniques call those callbacks directly—essentially eliminating the need for event generation altogether.

Existing callback-driven approaches suffer from two main limitations. **(L1)** They assume the knowledge of both the Android callbacks and the APIs to determine *what* to call and *how*, respectively. Given an app, the first challenge is to identify its callbacks. For that, existing tools maintain a fixed and often small list of supported callbacks. Once a callback is identified, it has to be invoked with arguments that match the types that the callback expects. Callbacks accept two types of arguments: *primitive*, *e.g.*, `int`, and `float`, or *objects*. Object arguments are harder to deal with. Prior techniques depend on a human expert for writing the necessary *driver* code, which would leverage widget-specific Android APIs to retrieve live objects from the app context, so that those can be supplied as arguments. Since adding support for a callback requires a non-trivial manual effort, it is hard to extend the support for all the callbacks in the framework. Quite understandably, while there are approximately 19,647 callbacks in Android 4.2 [16], the state-of-the-art callback-driven testing tool EHBDROID [39] supports only 58 of them. **(L2)** Apps accept user-supplied data as input, *e.g.*, text. Only generating event sequences, which existing tools focus on, is not enough, because certain functionalities may only be reachable under specific input. For example, a payroll app calculates tax differently depending on the income of an employee.

This paper presents COLUMBUS, an Android app testing technique that addresses both the challenges. To address **L1**, COLUMBUS adopts a two-phase approach. First, we statically identify all the callbacks present in the app (*what* to call). Specifically, our *callback discovery* module statically extracts all

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00121

the callback signatures $\mathcal{L}$ supported by the Android framework. Since an app has to override a framework callback to provide its own implementation, we use $\mathcal{L}$ to identify the callback implementations present in the app. Once callbacks are identified, then we dynamically prepare arguments (*how* to call) to invoke them with. Unlike previous techniques that rely on manually-written, callback-specific driver code to generate object arguments, we resort to a hybrid approach. Our *exploration* module performs a dynamic introspection of the app's heap at run-time, followed by a *type-guided* object filtering to supply appropriate arguments to the callback. This callback discovery and argument generation strategies together insulate COLUMBUS from the complexity of the Android API and obviate the need for any prior knowledge. To address **L2**, we leverage the fact that many user inputs are of primitive types, and often appear as the arguments to the callbacks. Therefore, the *argument generation* module symbolizes the primitive arguments of a callback, and performs an under-constrained symbolic execution to generate the possible values of those arguments to drive the execution along all paths. Symbolic execution is scoped within a single callback instead of the entire app to maintain a balance between precision and scalability.

In addition to tackling those two limitations, we integrate two novel feedback mechanisms into our exploration loop. **(i)** The *callback dependency* module passes on statically-identified data-dependencies between callbacks as feedback, which enables COLUMBUS to generate callback sequences that increase the likelihood of triggering crashes due to uninitialized objects, *e.g.*, `NullPointerException`. **(ii)** We design a *crash-guided* dynamic scoring mechanism that gradually deprioritizes crash-inducing paths in the app to drive the exploration towards unexplored code. In effect, COLUMBUS is incentivized to discover more crashes than rediscovering the already found ones.

We evaluated COLUMBUS on 60 apps of the AndroTest [37] benchmark, and top 140 real-world apps from the Google Play Store. Compared to the state-of-the-art model-based techniques STOAT [41] and APE [22], checkpoint-based technique TIMEMACHINE [19], and callback-driven technique EHB-DROID [39], COLUMBUS achieves 12%, 5%, 6%, and 31% more in average coverage, and discovers 4.42, 1.23, 1.57, and 5.48 times more crashes on the AndroTest apps, respectively. COLUMBUS is also able to find 70 crashes in 54 real-world apps.

In summary, this paper makes the following contributions:

Callback exploration. We propose a callback-driven Android app testing approach by presenting **(i)** a generic technique to extract all the callbacks present in an app (Section IV-A), and **(ii)** an analysis based on under-constrained symbolic execution (primitive arguments) (Section IV-B), and *type-guided* dynamic object filtering for generating valid arguments to invoke callbacks.

Feedback mechanism. Further, we make the app exploration systematic by integrating two novel feedback mechanisms: **(i)** a *data dependency* feedback that increases the probability of triggering bugs (Section IV-C) due to uninitialized variables, and **(ii)** a *crash-guided* dynamic scoring mechanism that prevents us from rediscovering the same bugs (Section IV-D).

Tool & evaluation. We implement the proposed technique in a practical tool called COLUMBUS, and we make it publicly available [9]. Our evaluation demonstrates that COLUMBUS outperforms the state-of-the-art tools both in terms of code coverage and the number of unique crashes that it identifies (Section V).

II. BACKGROUND

Android events. Android apps are event-driven programs. That is, apps behave as state machines, and events cause a transition from one state to the other. An event is generated in response to one or more user actions (UI events), or by Android itself (system events). Examples of UI events include `click`, `drag`, `pan`, `pinch`, `zoom`, *etc.* Modern Android devices are equipped with peripherals, such as, Bluetooth and WiFi, and sensors like motion sensors and accelerometers. Any change in the state of these devices is detected by the OS, which then generates a system event to notify "interested" apps. Examples of system events are Bluetooth disconnected, phone tilted, and low battery level.

Based on the number of actions needed to generate an event, we define two types of events: *primitive* and *composite*. Primitive events are either system events or UI events generated due to a single action. For example, `MotionEvent` (`ME`) reports the movement of an input device like a mouse, pen, finger, trackball, or `KeyEvent` reports key and button related actions. A composite event consists of multiple primitive ones, which are sequenced with strict spatial and temporal requirements. Say, we want to drag an object from point p_1, and drop it at point p_n along the trajectory $[p_1,p_2,p_3,...,p_n]$. In order to programmatically generate a `drag` event, the following sequence (temporal) of primitive events need to be fired at those exact coordinates (spatial): `ME.ACTION_DOWN` $(p_1) \rightarrow \{$`ME.ACTION_MOVE` $(p_i) \mid 2 \leq i \leq (n-1)\} \rightarrow$ `ME.ACTION_UP` (p_n). Without the support for a composite event, it is nearly impossible for a UI testing tool to generate most of them just 'by chance'. To make matter worse, numerous such composite events are widget-specific, *e.g.*, the `DateChanged` event recognized by `DatePickerDialog`. Therefore, adding support for individual events in a UI testing tool is nearly impossible.

Android callbacks. An Android *callback*, also known as an *event handler*, is a piece of code that the framework invokes when a specific event takes place, for example; the `onClick` callback is called when a `click` event occurs. Typically, the framework only provides empty callbacks, which an app selectively overrides to respond to the respective events. When an event is generated, it is broken down into `Messages`, which are then put into a `MessageQueue` managed by the `Looper`, the entity that runs the message loop. The `Looper` processes the `Messages` in first-in-first-out order, and calls the associated callbacks. While invoking a callback, the framework supplies the appropriate arguments, which can be of two types—*primitive*, *e.g.*, `int`, `float`, *etc.*, or `object`, *i.e.*, an instance of a class.

Android activity: An activity is a UI element that acts as a container of other UI elements. It often presents itself in the form of a window. Activities are managed by maintaining an activity stack. When a new activity starts, it is placed on the top of the stack, while the previous one is paused, and remains

below the current one in the stack. A paused activity does not come to the foreground again until the current activity exits. An activity transitions through different states of its *lifecycle* as a user navigates through an app. Lifecycle callbacks, *e.g.*, `onCreate`, `onPause`, `onResume`, are the ones associated with such lifecycle events.

III. Motivation and Challenges

This section introduces a motivating example, the challenges it presents to the state-of-the-art callback-driven app testing tools, and how we tackle them.

The code in Figure 1 shows three callbacks that an Android app might implement. The callback functions are executed when the user interacts with specific UI elements, *i.e.*, clicks on a list item, clicks on a button, and sets a date using a `DatePickerDialog` (Figure 2), respectively. UI-based testing tools [8] generate events, *e.g.*, clicks, to interact with the UI of such apps. However, these tools are not widget-aware, meaning that, they are unable to *systematically* generate composite events unless they already know how to generate them. For example, the following events need to be generated in an exact sequence, on specific UI elements, to call the `onDateChanged` callback—**(i)** `DatePickerDialog` widget is clicked to bring up the spinner control, **(ii)** the day/month/year is changed by clicking on the up/down arrows, and **(iii)** the `Set` button is clicked. It is unlikely for a UI-based testing tool to be able to deterministically generate this event sequence without any guidance. Moreover, to set a particular date, the up/down arrows need to be clicked a specific number of times—which is hard as well. To overcome this limitation, callback-driven techniques [39] invokes the callback, *e.g.*, `onDateChanged`, directly bypassing the UI layer altogether. While callback-driven testing shows promise, it still suffers from the following limitations.

Identifying callbacks. The first step of callback-driven testing is identifying the callbacks. Unfortunately, the set of callbacks supported by the Android framework is huge. While previous research [16] identified approximately $19,647$ callbacks in Android 4.2; EHBDroid, the state-of-the-art callback-driven testing tool, supports only 58 callbacks. Columbus statically analyzes the app and the Android framework together to address this issue (Section IV-A).

Providing callback arguments. Callbacks accept either primitive arguments or objects. The primitive arguments are often involved in path conditions within the callback. Without the correct value of such primitives, part of the callback may never be exercised. In Figure 1, the `Toast` message appears only on a specific date. Existing callback-based testing tools use a set of predefined values to invoke callbacks. Therefore, Line 19 will possibly never be explored. Columbus symbolizes primitive arguments and employs under-constrained symbolic execution to infer values to make larger part of the callback code reachable (Section IV-B).

For object arguments, such as, the `ListView` and `View` arguments of the `onListItemClick` callback in Figure 1, callback-driven tools use the Android API (by statically instrumenting the app) to retrieve correct objects from the app context, as shown in Figure 3 (Line 2 and Line 7). However, this approach is not scalable, as the number of callbacks in the Android framework is huge, and the tool requires adding explicit support for all the arguments of all the callbacks. Instead, Columbus retrieves live objects from the app heap at runtime, and then applies *type-guided* object filtering to provide the correct arguments (Section IV-B). Type information comes from a one-time, static, pre-processing phase.

```
1  protected void onListItemClick
       (ListView l, View v, int position, long id) {
2    File f = (File)(mList.get(id).get(ITEM_KEY_FILE));
3    if (f.isFile()) {
4      mSelectedFile = f;
5      showDialog(DIALOG_IMPORT_FILE);
6    }
7  }
8
9  public void
       onClick(DialogInterface dialog, int whichButton) {
10   File f = mSelectedFile;
11   Intent i = new Intent(mContext, myActivity.class);
12   Uri u = Uri.fromFile(f);
13   i.setData(u);
14   startActivity(i);
15 }
16
17 public void onDateChanged
       (DatePicker view, int year, int month, int day) {
18   if (day == 15 && month == 6 && year == 2020)
19     Toast.makeText(context, "Success!", ...).show();
20 }
```

Fig. 1: Code containing three callbacks. Their data dependencies (■) and checks on the arguments (■) are highlighted.

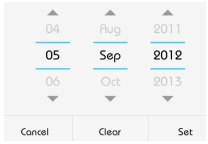


Fig. 2: A `DatePickerDialog` widget

```
1  void onCreate(Bundle bundle) {
2    ListView lv = getListView();
3  }
4
5  void ehbTest() {
6    for (int i=0; i<lv.size(); i++) {
7      View v = lv.getChildAt(i);
8      long id = lv.getAdapter()
9                  .getItemId(i);
10     this.onListItemClick(lv,v,i,id);
11   }
12 }
```

Fig. 3: EHBDroid instrumentation for `onListItemClick()`

Data dependency feedback. Variables are often shared among multiple callbacks. Shared data introduces data dependencies, which an app should either enforce by restricting available UI actions, or handle by placing a sanity check. In Figure 1, both the `onClick` and `onListItemClick` callbacks use the same variable `mSelectedFile`. Specifically, `onListItemClick` opens a file, and sets the file handle `mSelectedFile` (Line 4), which `onClick` uses in Line 10. This implies that `onListItemClick` has to be invoked before `onClick`, otherwise the `onClick` method would generate a `NullPointerException`. Columbus statically infers such data dependencies and passes the same as feedback during testing. While synthesizing a callback sequence, Columbus attempts to violate the expected order to increase the likelihood of inducing crashes (Section IV-C).

IV. The Columbus Framework

In this work, we propose Columbus, a framework to test Android apps by directly invoking their callbacks. For a given Android app, Columbus first identifies its callbacks

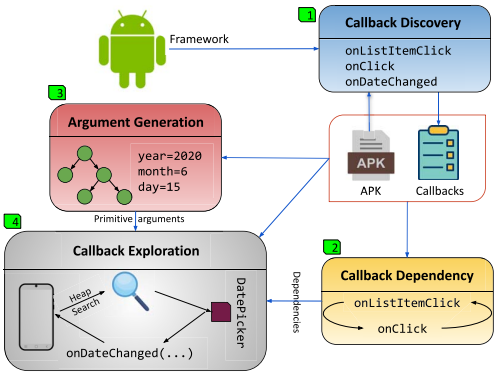


Fig. 4: Overview of COLUMBUS with reference to the motivating example in Figure 1

(Section IV-A). It then obtains the primitive argument values that correspond to different execution paths in these callbacks (Section IV-B) and identifies inter-callback dependencies (Section IV-C). Finally, our tool invokes the identified callbacks—**(i)** in orders that initially violate (to increase the chances of triggering uninitialized data-related bugs), and later respect their dependencies, **(ii)** with their expected arguments during the exploration (Section IV-D). COLUMBUS keeps track of the callback-defining classes explored during the app execution, and gives higher priority to exploring classes that have been less explored. Figure 4 depicts the high-level workflow of our system.

A. Callback discovery

Every Android app defines its own set of callbacks. Though state-of-the-art approaches [39] resorted to a predefined set of callbacks, the Android framework contains thousands [16] of callbacks, and the number is constantly increasing. In order to facilitate effective app exploration, in this work, we present an approach to automated callback discovery. COLUMBUS's callback identification is presented in Algorithm 1. At a high level, our callback discovery approach first statically analyzes the framework (Function `AndroidFrameworkAnalysis`) followed by an analysis of the app under test (Function `AppAnalysis`), and outputs a list of callbacks present in the app.

Android framework analysis. Our analysis is based on two observations. **(i)** As discussed in Section II, in order to perform the intended action once an event is generated, an app needs to override the respective callback present in the Android framework. To be overridden, a callback needs to be declared as either a *protected*, or a *public* method within the framework. **(ii)** Moreover, at runtime, callbacks are typically invoked within the framework through a series of internal method calls once an event is generated—meaning that, callbacks have caller(s) within the framework.

COLUMBUS first constructs the framework's callgraph CG_f. To build the call graph, COLUMBUS performs intra-procedural type inference [35] to determine the possible dynamic types of the object on which a method is called. When this fails, COLUMBUS then over-approximates the possible targets as

Algorithm 1: Static callback identification

```
1  Function AndroidFrameworkAnalysis
       Input  : Android framework JAR
       Output : Classes with callback candidates Δ
2      Δ ← {}
3      CG_f ← GetCallGraph(JAR)
4      CH_f ← GetClassHierarchy(JAR)
5      foreach class c_f ∈ GetClassesFromJar(JAR) do
6          M_f ← ∅
7          foreach method m_f ∈ GetMethodsFromClass(c_f) do
8              if IsPublicOrProtected(m_f) then
9                  if GetCallers(c_f,m_f,CG_f) ≠ ∅ then
10                     M_f ← M_f ∪ m_f
11                 end
12             end
13         end
14         Δ[c_f] ← Δ[c_f] ∪ M_f
15     end
16     foreach (c_f, M_f) ∈ Δ do
17         foreach subclass c'_f ∈ GetSubClasses(c_f) do
18             M'_f ← Δ[c'_f]; M'_f ← M'_f ∪ M_f; Δ[c'_f] ← M'_f
19         end
20     end
21     return Δ, CH_f
22 Function AppAnalysis
       Input  : App's APK, Framework classes with
                callback candidates Δ, Framework's class hierarchy CH_f
       Output : Application callbacks CB
23     CB ← ∅
24     foreach class c_a ∈ GetClassesFromApk(APK) do
25         ClassAndItsParents ← c_a ∪ GetSuperClasses(c_a)
26         foreach cp_a ∈ ClassAndItsParents do
27             foreach (c_f, M_f) ∈ Δ do
28                 if cp_a extends c_f ∨ cp_a implements c_f then
29                     foreach m_a ∈ GetClassMethods(cp_a) do
30                         foreach m_f ∈ M_f do
31                             if IsCompatible(m_f, m_a) then
32                                 CB ← CB ∪ m_a
33                             end
34                         end
35                     end
36                 end
37             end
38         end
39     end
40     return CB
```

all the subclasses of its static type. Now, for every method m_f in a framework class c_f, COLUMBUS considers m_f as a potential callback (Lines 7 – –13) if—**(i)** m_f is declared as either *protected*, or *public*, and **(ii)** m_f has at least one caller in CG_f. At the end, we compute a mapping Δ that maps each class c_f to their potential callbacks. Each callback m_f is a tuple, which consists of the defining class c_f, the method name, and the types of its arguments. Now, this mapping Δ is incomplete, because a class can inherit callbacks from its superclasses as well. Therefore, COLUMBUS computes the complete list of potential callbacks for every c_f by walking up the class hierarchy to consolidate superclass callbacks, too (Lines 16 – –20). The updated callback mapping Δ and the class hierarchy information CH_f are returned as the output. Note that COLUMBUS performs the framework analysis once per framework.

The above analysis is inspired by EdgeMiner [16]. The main goal of EdgeMiner is to detect framework callbacks, and using that to discover the registration methods within the framework. However, the end goal of Columbus is to detect application level callbacks by leveraging the framework callbacks.

Android app analysis. The goal of this phase is to find whether any app class method m_a is a valid overriding method of the framework class callback m_f. In order to override a callback

within an app, the app class c_a needs to either extend or implement the corresponding callback-defining class c_f of the Android framework. For example, in Figure 1, to override the `onListItemClick` callback, the app class needs to extend the `ListActivity` framework class. COLUMBUS identifies such pairs of classes (c_f, c_a) by statically analyzing the app. In the next step, it checks whether any app method $m_a \in c_a$ has the same name and the same number of arguments as any framework method $m_f \in c_f$, and the arguments of m_a are *type-compatible* with those of m_f (Lines 29−−35). We call a type t_1 to be *compatible* with another type t_2, if either $t_1 = t_2$, or t_1 is a subclass of t_2 according to the class hierarchy. To determine type compatibility, COLUMBUS constructs the full class hierarchy by unifying ($\oplus$) the framework class hierarchy CH_f with the app class hierarchy CH_a. Let $A \rightarrow B$ denote that A is a superclass of B. Now, if the relations $H_1 = A \rightarrow B$ and $H_2 = B \rightarrow C$ appear in CH_a and CH_f, respectively, then $H_1 \oplus H_2 = A \rightarrow B \rightarrow C$. Finally, we obtain the set of potential callbacks in an app. Our analysis would discover all three functions `onListItemClick`, `onClick`, and `onDateChanged` in Figure 1 as callbacks.

Identifying callbacks by analyzing either the app, or the framework alone is challenging. Since a callback is invoked by the framework, the callback methods do not have incoming edges visible from the call graph of the app. However, an analysis relying only on this fact alone will generate false positives—because, it could detect a non-callback method as a callback due to the inherent incompleteness of Java call graphs [36]. Similarly, our framework analysis is over-approximated in a way that will definitely contain the callbacks, but non-callbacks methods, too. Intuitively, therefore we 'intersect' the framework callback candidates and app methods to determine the true callbacks.

During this phase, we can encounter methods of a generic Android framework class `Object`, that are declared as `public`, and can therefore be overridden by the corresponding application-level classes inheriting the `Object` class. The number of such callbacks appearing as part of the final callback list was negligible (around 3%). We do not consider such methods as callbacks.

B. Generating arguments for callbacks

In order to invoke a callback, we need to provide argument values conforming to the correct types. In case of GUI-action-driven exploration strategies, the framework provides these arguments, which are derived from the events resulting from the GUI actions. Therefore, to invoke callbacks *without* relying on GUI actions, COLUMBUS needs to tackle the challenge of generating arguments for these callbacks, with a goal to explore the paths within a callback resulting in faster coverage and better crash discovery.

A callback argument can be one of two types: primitive or reference. For each type, COLUMBUS uses different strategies to generate the corresponding arguments.

1) Primitive type arguments.: Primitive type arguments, *e.g.*, `integer`, `long`, `string`, and `boolean`, are typically involved in program paths that can only be explored with a specific set of values. For instance, Line 19 of the `onDateChange` callback in Figure 1 will get executed *only if* the integer arguments `day`, `month`, and `year` are equal to 15, 6, and 2020. Therefore, to effectively explore all the paths in such a callback without resorting to a computationally expensive random search, COLUMBUS needs to provide these specific set of values to the callback during invocation. In this case, COLUMBUS symbolizes respective callback arguments, and performs an under-constrained symbolic execution (until termination, or time-out) to generate concrete values.

Precisely, COLUMBUS starts the symbolic execution at the entry point of each of the callbacks, and collects constraints on the arguments corresponding to each of the execution paths. It then solves these constraints and generates concrete argument values, which when provided as arguments to the callback during invocation, result in exercising those paths within the callback. During symbolic execution, we track constraints on objects that modify the program state, such as **(i)** callback arguments, and **(ii)** API return values.

Callback arguments. COLUMBUS executes the callback with symbolic and unconstrained arguments. It then collects the constraints in each of the execution paths that involve operations on the symbolic arguments. For example, if one of the arguments is an object, and during execution, one of its fields is set to 5, COLUMBUS's symbolic execution engine will automatically add a constraint stating that the specific attribute needs to be equal to 5 (to follow a particular program path of interest).

API calls. COLUMBUS's symbolic execution engine generates summaries for common functions, for example, the Java runtime function `exit()`. These summaries capture the side effects of these APIs that modify the program state. For APIs without a summary, we return a fresh symbolic value conforming to the return type of the API.

COLUMBUS's symbolic execution engine is capable of generating concrete values of `integer`, `float`, `boolean`, and constant `string` types.

2) Reference type arguments: Reference type argument objects frequently represent UI elements where a user performs certain actions. In Figure 1, when a user clicks on `AlertDialog` (a subclass object of `DialogInterface`), the framework invokes the `onClick` callback with an argument object of type `AlertDialog`. Therefore, to invoke the `onClick` callback without relying on the Android framework, we need to provide an object of type `DialogInterface`, or a subclass of `DialogInterface`—as an argument.

App heap search. During the app exploration (Section IV-D), as and when new `Activities` are visited, these object instances are created in the app heap. Therefore, in order to invoke a callback that requires reference type arguments, COLUMBUS monitors the app heap by dynamically instrumenting the app under test. In many cases, the argument type present in the callback signature is not the one created in the app heap. In Figure 1, the `onClick` callback has an argument of type `DialogInterface`. However, the object created will be of type `AlertDialog`, a subclass of `DialogInterface`. To account for this scenario, *i.e.*, if an object instance of a reference type inferred from the callback signature is not available in the app heap, COLUMBUS searches for object instance(s) that is a subclass of the required type.

Custom object creation. It may still happen that no object instances of the required type or its subclass are found in the heap. For example, certain types of objects required as a callback argument, *e.g.*, `KeyEvent`, and `MotionEvent`, that are created by the Android framework *only* when it registers touch, or key-press on UI elements. Therefore, in order to invoke such callbacks, COLUMBUS leverages Java reflection. Specifically, for such a reference, COLUMBUS creates the object using its public constructor. If the constructor expects primitive type arguments, COLUMBUS uses either a random value, or a value from a pre-defined set as the argument. For example, to create `KeyEvent`, or `MotionEvent` objects, COLUMBUS uses pre-defined values as they should be valid screen coordinates in order to successfully explore the callback. If a constructor expects reference type objects, COLUMBUS either finds these objects through app heap search, or creates recursively through Java reflection. For example, if we were to create an object of type `A` which has a constructor that accepts an object of type `B`, then we create objects bottom up (*i.e.*, first `B`, then `A`). In case multiple such constructors exist, COLUMBUS picks the one which requires the least number of reference type arguments.

C. Inter-callback dependency

Callbacks within an app can share variables resulting in *read-write* data dependencies. As discussed in Section III, for `onListItemClick` and `onClick` callbacks (Figure 1), prioritizing dependency-violating order, *i.e.*, invoking `onClick` before `onListItemClick`, brings us faster to a crash discovery. Whereas invoking the callbacks in the dependency-respecting order allows for a better code coverage. For example, the execution of the Lines $13--14$ in `onClick` happens only if the reference `mSelectedFile` accessed at Line 10 is defined by a prior execution of `onListItemClick`.

Based on this observation, COLUMBUS computes callback pairs having shared variable dependencies by performing a field-insensitive analysis of the app. The intuition is to first compute a set of class variables $vars$ that are *not* initialized through a *default initializer*. The *default initializer*s are the methods that get automatically invoked whenever a class or activity gets created, *e.g.*, the life cycle methods of an activity, class constructors, *etc.* These variables $vars$ are our target candidates, since they are defined and accessed only through callbacks. Next, for every such variable $var \in vars$, COLUMBUS searches for callback pairs (cb_1,cb_2) where one of them *read*s (`R`) var, and the other *write*s (`W`) var. The output of this phase will be a set of variables with their dependent callback pairs. For the example in Figure 1, the output will be $\{$`mSelectedFile`, (`'R'`, `onClick`), (`'W'`, `onListItemClick`)$\}$.

These dependency pairs are used as feedback during the exploration phase detailed in Section IV-D. In order to accelerate crash discovery, COLUMBUS implements a weighted-score based exploration strategy, which initially prioritizes executing callbacks that write to variables over the callbacks that read from the same variables—inducing the dependency violating callback invocation orders. However, during the exploration, COLUMBUS dynamically adjusts the scores, *e.g.*, penalizes the callbacks that frequently result in a crash, or prioritizes the callbacks that are executed less frequently, in order to explore newer or less explored program paths as well.

D. Callback-guided exploration

To explore an app under test, we first statically obtain its callbacks (Section IV-A), their dependencies (Section IV-C), and the primitive argument values (Section IV-B). Then, COLUMBUS spawns the app, dynamically instruments it to inspect the app heap, and starts exploring its functionalities. COLUMBUS invokes a callback whenever an instance of the activity, or the class defining the callback appears in the app's heap. If the callback expects reference type arguments, COLUMBUS then generates such argument objects using the strategy detailed in Section IV-B. Algorithm 2 gives an overview of our app exploration strategy. COLUMBUS's exploration strategy is composed of the following components:

Activity monitor. As the app is being explored, two kinds of entities get created, or destroyed in the heap: **(i)** activities and related UI element objects, and **(ii)** regular class objects, as the side-effect of calling a callback that instantiates the class. The activity monitor records such events by monitoring the invocation of the lifecycle callbacks of the activities, and the class constructors. For example, invocation of `onCreate()` signals an activity creation, and `onDestroy()` is invoked when an activity is destroyed. The activity monitor maintains an activity stack $\mathcal{S}$ by pushing an activity to $\mathcal{S}$ when a new activity is created, and popping an activity off $\mathcal{S}$ when it is destroyed. Therefore, the most recently created activity, which we call as the *live* activity, always remains at the top of $\mathcal{S}$.

The app is explored in a depth-first manner, and runs in continuous *cycles*. For a live activity act, the activity monitor retrieves all the class objects $newClasses$ created in the app heap (Line 18), passes it on to the *selector* for choosing the next callback cb, which is then executed by the *executor*. The function `getNewClasses()` returns only those classes for which at least one callback is still unexplored. If a callback creates a new live activity act', the activity monitor puts act on hold, and switches to act'. When all the callbacks of an activity or its associated classes have been executed, the activity monitor destroys the activity, removes it from $\mathcal{S}$ (Lines $19--22$), and starts exploring the next live activity. One testing cycle ends, and the next one begins when $\mathcal{S}$ becomes empty.

Selector. The selector module receives the candidate classes $newClasses$ to be explored from the activity monitor, and chooses a callback cb to be executed next (Line 24). While choosing cb, it considers the class weights ClW, callback weights CbW, inter-callback dependencies Dep, and the visited status $explored$ of the callbacks. The $explored$ map is cleared when a testing cycle begins. All the weights are initially set to zero, and are dynamically adjusted during the exploration based on how frequently the classes and the callbacks have been explored. Similarly, when a callback is explored, the $explored$ map is updated (Line 26).

To choose a callback, the selector employs multiple strategies in the following order: **(i)** In the beginning, when none of the

callback is explored, the selector uses Dep to choose the callback cb with the read (R) dependency, and its defining class cl. **(ii)** The selector consults the $explored$ map to prioritize unexplored callbacks over the explored ones. **(iii)** A class or callback with lower weight (ClW or CbW) has been explored the least; therefore it is prioritized next for execution. The tie among multiple unexplored classes, or callbacks with the same weight is broken randomly.

Algorithm 2: Callback driven exploration

```
1  Function CallbackExploration
       Input  : Application callbacks AC, their dependencies Dep,
                class hierarchical information CH_f and CH_a, duration t
       Output : Crash dumps crashes
2      crashes ← ∅, explored ← {}, testingCycle ← 0
3      CbW ← ∅   // callback weights
4      ClW ← ∅   // class weights
5      foreach callback cb ∈ AC do
6          cl ← GetclassDefiningMethod(cb)
7          CbW ← CbW ∪ (cl,cb,0.0)
8          ClW ← ClW ∪ (cl,0.0)
9      end
10     while until t is reached do
11         spawnApp ()
12         testingCycle ← testingCycle + 1
13         foreach callback cb ∈ AC do
14             explored[cb] ← false
15         end
16         while until no new activity left to explore do
17             act ← getLiveActivity()
18             newClasses ← getNewClasses(act,explored)
19             if newClasses = ∅ then
20                 RemoveActivity(act)
21                 go to Line 16
22             end
23             cl ← getNextClass(newClasses ∪ act,ClW,Dep)
24             cb ← getNextCallback(cl,explored,CbW,Dep)
25             if cb = ∅ then
26                 explored ← explored − (cb,false) ∪ (cb,true)
27                 go to Line 16
28             end
29             allargs ← generateArguments(cl)
30             foreach args ∈ allargs do
31                 inst ← getInstance(cl)
32                 newCrash ← ExecuteCallback(inst,cb,args)
33                 if newCrash ≠ ∅ then
34                     crashes ← crashes ∪ newCrash
35                     UpdateAndPenalizeWeights(ClW,CbW,cl,cb)
36                     restartApp () and go to Line 10
37                 end
38                 else
39                     UpdateWeights(ClW,CbW,cl,cb)
40                 end
41             end
42         end
43     end
44     return crashes
```

Executor. The executor executes the callback selected by the selector. The executor searches the app heap for an instance of a class, or an activity that overrides the callback (Line 31). If an instance is found, the executor generates the arguments for the callback respecting their types (Section IV-B). However, an argument can have multiple possible values executing different paths (primitive), or depending on the availability of objects in the heap (reference). The executor, therefore, schedules the callback for execution for each combination of such inferred values. After each execution, the class weight for a class cl and the callback weight for a callback cb are updated as shown in Figure 5.

Intuitively, the executor updates the weights to reflect what percentage of callbacks are executed with respect to the total number of possible invocations—since a crash, or a creation

$$CbW_{cb} \;:-\; CbW_{cb} + \frac{ex_t}{sch}$$
$sch \leftarrow$ number of scheduled executions of cb
$ex_t \leftarrow$ number of executions of cb at time t

$$ClW_{cl} \;:-\; avg(CW_{cb}) \forall cb \in cl$$

Fig. 5: New class and callback weights after each execution

of new activity may interrupt the processing of the rest of the scheduled executions. The class weights are accordingly adjusted such that the least explored class, and its callbacks are prioritized to be executed the next time the activity comes live.

Crash detector. After the execution of a callback, the crash detector monitors whether it results in a crash of the app. We do not want to rediscover the same crash repeatedly. Therefore, if a crash happens, the `UpdateAndPenalizeWeights()` (Line 35) function updates the class weights to deprioritize the callback cb, and its defining class cl—the callback weight CbW_{cb} is increased by δ (an empirically determined constant), and accordingly the class weight ClW_{cl} is adjusted. The idea is to gradually increase the callback weight in order to account for the case when *only* a specific set of argument values results in a crash, and all other values should still be able to explore the callback. Therefore, instead of not choosing the callback at all, the selector deprioritizes the callback for some time.

V. Evaluation

In our evaluation, we aim at answering the following research questions: **RQ1.** How does Columbus compare with the state-of-the-art testing tools in terms of both code coverage and discovered crashes? **RQ2.** How effective is Columbus in finding crashes in popular, real-world apps? **RQ3.** What is the benefit of leveraging dependency feedback?

A. Experimental setup

Dataset. To answer **RQ1** and **RQ3**, we used AndroTest [37], a collection of 68 apps. This dataset has become the de facto standard benchmark for Android app testing, and it has been used in the evaluation of a large number of tools [37], [41], [31], [19], [29], [33], [13], [11], [18], [46], [49], [30]. However, we had to remove 8 apps that were not fully compatible with Android 9 (which is the environment we used for Columbus). For example, the `ListView` in the `netcounter` app does not appear in Android 9. Therefore, we used the remaining 60 apps for all our experiments.

For **RQ2**, we created a dataset of popular, real-world apps. We will refer to this dataset as the *real-world* dataset. To build this dataset, we first compiled a list of Google Play Store [6] apps with a minimum of 500,000 installs and a user rating of at least 4.5 stars. Then, we collected first 140 apps compatible with Frida instrumentation. As we show in Table II, these apps are quite diverse and belong to 14 broad categories.

Environment. Our experiments were conducted on a system with an Intel(R) Core(TM) i9-10885H @ 2.40GHz processor (16 cores), 128GB of memory, and 1TB of solid-state drive

(relevant for the snapshot save and restore mechanism used by TIMEMACHINE), running a 64-bit Ubuntu 20.04 operating system. For testing, we used 8 Google Pixel 3a phones running Android 9 (Pie, API level 28), with the Internet and Bluetooth connectivity enabled. We did not create any accounts for those apps that allow user logins. We ran each tool for 3 hours on each app, repeated each experiment 5 times, and averaged out the results to minimize the effect of any inherent randomness. Before testing each app, we first brought the phones to a *clean-slate* state by wiping its `sdcard` contents, and then pushed the `sdcard` files used by STOAT in their experiment to the phones. All the tools except TIMEMACHINE, which requires a virtual machine (VM) to operate, were tested on real hardware (phone).

Pre-exploration. Before the dynamic exploration could begin, COLUMBUS prepares an app by running the first three static pre-processing phases. We provide relevant results for the 60 apps of the AndroTest dataset: The *callback discovery* module identified a total of 30,682 and 4,991 callbacks in the Android framework and the apps, respectively. Out of 4,991 app callbacks discovered, 1,566 callbacks had at least one primitive argument, thus necessitating the invocation of the *argument generation* module. With a timeout of 5 minutes, the argument generation succeeded for 1,332 callbacks, while it timed out for the remaining 234 callbacks. Additionally, 4,147 callbacks have at least one reference type argument, and in total 4,857 reference type arguments. Out of them, 4,650 objects were always found on the heap, and the remaining 207 objects needed to be created. Finally, the *callback dependency* module discovered a total of 2,456 dependency relations between 975 variables across all the apps.

Coverage and crash collection. We used EMMA [4] to collect statement coverage. The coverage data was collected every minute for all tested tools. EMMA injects its own instrumentation code into the apps. Unfortunately, its coverage reports do include coverage data from its own packages, which can either inflate, or deflate the overall coverage. Therefore, we excluded EMMA-specific classes from the coverage calculation.

We detect crashes by parsing **(i)** LOGCAT [7] logs fetched by the log watcher, a long-running process that streams logs from the devices (phones) in real-time, and **(ii)** logs of the crashes captured by the FRIDA server. We used the widely adopted practice of computing the stack hash to determine the *uniqueness* of crashes. Crashes that do not contain the app's package name were filtered out. For FRIDA reports, we occasionally observed that certain crashes that originate from the dynamic instrumentation contain an app's package name. Therefore, we manually inspected and removed those irrelevant crashes after the initial package-name-based filtering. Then, we normalized the stack traces for the remaining crashes by removing irrelevant and ephemeral information, *e.g.*, timestamp, process id (PID), *etc.* Finally, we compute hashes over these sanitized stack traces.

Implementation. We implemented the first three phases of our analysis, *viz.*, callback identification, callback dependency discovery, and primitive argument generation using the ANGR [3] binary analysis framework. All these phases are performed offline, before the testing begins on the device.

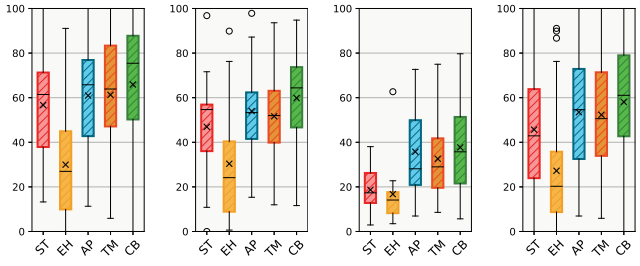
Apps	Line coverage						Crashes					
	ST	EH	AP	TM	CB	CB_{wd}	ST	EH	AP	TM	CB	CB_{wd}
mileage	38	23	58	40	60	57	2	0	15	9	4	4
bomber	61	56	66	97	88	87	0	0	0	0	0	0
mirrored	31	16	38	46	47	47	0	0	0	1	1	1
batterydog	59	5	72	73	72	72	0	0	0	1	0	0
triangle	90	91	90	91	91	91	0	0	0	0	1	1
translate	46	29	48	48	49	49	1	1	1	0	1	1
anymemo	26	18	50	42	52	46	2	1	6	6	7	7
zooborns	18	17	19	25	26	26	3	0	3	3	1	1
qsettings	40	23	50	40	47	46	1	1	1	0	1	0
wchart	57	24	32	51	85	83	2	1	0	0	3	3
addi	17	16	21	19	18	18	1	0	8	1	3	3
LNM	49	3	34	48	50	50	4	0	4	7	2	1
gestures	32	32	32	50	78	78	0	0	0	0	0	0
MNV	35	13	64	42	68	68	2	1	4	4	1	1
wikipedia	24	21	25	31	19	19	0	0	0	0	0	0
dialer	66	53	65	40	73	73	1	1	1	3	2	2
photost	24	9	26	28	12	12	2	1	1	3	3	3
battery	92	55	55	93	88	88	0	0	0	3	0	0
aCal	18	8	28	29	22	19	3	0	5	3	3	1
tomdroid	55	24	57	53	61	59	0	0	4	0	2	2
RMP	82	87	83	65	92	92	1	0	0	1	2	2
SpriteText	62	63	62	63	61	59	0	0	0	0	0	0
LPG	63	37	89	82	0	0	0	0	0	0	0	0
ringdroid	0	40	42	23	47	47	1	2	4	2	2	2
sftp	11	5	15	12	18	18	0	0	0	0	3	1
PWMG	3	6	7	16	6	6	0	1	0	0	2	2
fbubble	49	49	56	82	74	72	0	0	0	0	3	3
myexp	55	1	33	46	65	63	0	0	0	1	7	7
sanity	13	8	26	27	36	35	1	0	2	1	2	1
SMT	87	2	87	63	87	85	0	0	0	0	0	0
alogcat	65	33	73	79	60	53	0	0	0	0	2	2
worldclock	97	90	98	94	95	95	1	1	0	1	2	2
mlife	87	35	86	84	92	92	0	0	0	0	2	2
lbuilder	22	28	28	26	37	35	0	1	0	0	4	4
CDT	63	31	65	85	87	87	0	0	0	0	0	0
bites	26	15	42	36	54	54	2	0	5	8	3	3
multisms	40	26	74	57	78	78	0	1	0	1	1	1
yahtzee	69	3	46	6	51	46	1	0	3	1	3	3
nectroid	40	27	44	38	46	46	0	0	0	2	2	2
anycut	70	12	71	71	66	66	0	2	0	0	3	3
PMM	66	27	62	56	65	62	4	0	11	3	4	4
manpages	40	20	54	77	78	74	0	0	0	1	3	3
zoffcc	18	15	16	20	16	16	3	0	4	1	4	4
amazed	62	64	76	52	84	84	0	0	1	1	1	1
alarmclock	72	15	76	68	71	71	6	0	4	4	5	5
hndroid	13	5	11	8	15	15	0	1	0	2	2	2
sboard	100	58	100	100	100	100	0	0	0	0	0	0
hotdeath	16	63	73	75	80	76	1	3	2	0	5	5
dalvik-exp	23	6	72	70	64	64	1	0	5	3	4	4
jamendo	10	13	28	9	30	30	5	3	0	0	5	5
importcont	57	2	53	42	78	74	0	0	0	0	1	1
blokish	36	35	49	52	45	45	0	0	2	0	2	2
Book-cat	4	4	33	35	38	38	0	1	2	4	4	0
Templaro	55	76	87	60	86	83	0	1	0	2	3	3
DAC	53	48	76	88	94	91	0	0	0	0	0	0
Agrep	37	8	58	63	61	58	0	0	7	2	7	7
Syncmypix	15	18	21	25	26	26	1	1	0	1	3	3
tippytipper	72	9	86	84	89	89	0	0	0	0	2	2
WHAMS	80	0	77	69	79	79	0	0	0	1	1	1
A2dp	29	14	40	45	47	42	6	0	6	0	3	3
Avg/Sum	46	27	53	52	58	57	58	25	111	87	137	126

TABLE I: Coverage and the number of crashes reported by all the tools in the AndroTest dataset. ST: STOAT, EH: EHBDROID, AP: APE, TM: TIMEMACHINE, CB: COLUMBUS, CB_{wd}: COLUMBUS without dependency feedback

For exploration, the final phase, we leveraged the FRIDA [5] dynamic instrumentation toolkit.

B. Experimental results

1) Performance on benchmark apps: To investigate how our technique performs with respect to prior work, we use the AndroTest benchmark apps. Specifically, we compared the achieved code coverage and the number of crashes found by COLUMBUS with the state-of-the-art model-based techniques STOAT [41] and APE [22], checkpoint-based technique TIMEMACHINE [19], and callback-driven technique EHBDROID [39]. Unfortunately, we could not make the publicly available version of EHBDROID work on our test apps due to the incompatibility of their instrumentation module with our test subjects. Instead, we implemented their testing strategies by modifying COLUMBUS in three ways: **(i)** we consider only those 58 callbacks supported by EHBDROID, **(ii)** we disabled dependency and crash guidance, and **(iii)** we restricted primitive



(a) <1K (30) (b) [1K,3K) (17) (c) ≥3K (13) (d) all (60)

Fig. 6: Coverage (Y-axis) achieved on AndroTest, grouped by app size (Lines of Code). Number of apps in a size group is indicated in parentheses. 'x' denotes the mean of a boxplot

argument values to those used by EHBDroid instead of the values computed by our argument generation module.

In Table I, we present the statement coverage achieved as well as the crashes triggered by all tools on the benchmark apps. **Coverage.** We find that Columbus achieves higher code coverage than Stoat, EHBDroid, Ape and TimeMachine for 45, 55, 41, and 41 apps, respectively. Moreover, Columbus achieves the best coverage in 36 apps, followed by TimeMachine (16 apps), Ape (10 apps), Stoat (5 apps), and EHBDroid (2 apps). To gain an overall view of the tools' performances, we report the average code coverage, achieved by each tool across all apps, in the last row of Table I. As can be seen, Columbus attains the highest (58%) coverage on average, followed by Ape (53%), TimeMachine (52%), Stoat (46%), and EHBDroid (27%). Figure 8 shows the progression of coverage over time for all the tools averaged across all the benchmark apps. Starting from the 5th minute, the coverage achieved by Columbus exceeds other tools. Until approximately the 20th minute, the coverage increases at a fairly fast rate, after that, it starts to slow down. Further, the boxplot in Figure 6 shows the *spread* of the coverage achieved by all the tools grouped by the size of the apps. We use group sizes identical to the ones used in previous work [19]. As the figure shows, Columbus exhibits significant improvement over other tools in terms of coverage for all size groups.

The improvement in coverage for Columbus can be attributed to its systematic exploration of the callbacks. While UI-based techniques struggle to generate complex events and appropriate user input, Columbus sidesteps this problem by directly calling the callbacks and supplying argument values (computed by the argument generation module) that are likely to explore additional code paths. In addition, the crash-guidance feedback helps Columbus to make the best use of the time-budget by preventing the exploration from getting stuck at individual crashes for a long time.

Figure 7 shows a code snippet from the `RandomMusicPlayer` app from AndroTest. This example shows an interesting case where Columbus naturally enjoys clear benefits over previous, more "heavyweight" techniques that use symbolic execution [13], and other UI-testing tools. To explore all the branches (`if` conditions), a UI-based tool would need to `click` on all corresponding buttons, which is challenging.

```
1  public void onClick(View target) {
2    // Send intent according to the button clicked
3    if (target == mPlayButton) {
4      startService(new Intent(MusicService.ACTION_PLAY));
5    } else if (target == mPauseButton) {
6      startService(new Intent(MusicService.ACTION_PAUSE));
7    } else if (target == mSkipButton) {
8      startService(new Intent(MusicService.ACTION_SKIP));
9    } else if (target == mRewindButton) {
10     startService(new Intent(MusicService.ACTION_REWIND));
11   } else if (target == mStopButton) {
12     startService(new Intent(MusicService.ACTION_STOP));
13   } else if (target == mEjectButton) {
14     showUrlDialog();
15   }
16 }
```

Fig. 7: Code snippet (redacted) from RandomMusicPlayer

ACTEve [13] solves this problem by concolically executing the app together with an instrumented version of the Android framework. Since, in our case, Columbus introspects the app heap to retrieve live objects, we observed the coverage of this app quickly going up, because Columbus invokes the `onClick` callback with all the button `Views` already present in the heap.

To better understand the challenges Columbus faces during exploration, we manually examined 10 of those apps where Columbus did not achieve the best coverage. We summarize our findings next: **(i)** For callbacks where the symbolic execution timed out, the *argument generation* module could not return any useful value. As a result, Columbus fell back to its default strategy of trying out random argument values, which negatively affected the coverage. **(ii)** There exist callbacks that are *stateful*. That is, the application logic is conditioned on `class` variables. Note that Columbus is not state-aware, therefore this challenge is orthogonal to what Columbus aims to solve. **(iii)** For unconstrained callback arguments, we use random values from a predefined list, which might be ineffective. For instance, the `yahtzee` app lists the game moves in a drop-down list. A move can be chosen by the `arg2` argument (unconstrained) of the `onItemSelected(_,_,arg2,_)` callback, which then looks up the appropriate UI object using that argument. Many such values of `arg2` that we supply could be invalid, while UI-based techniques can "blindly" `click` on the list item without being aware of the valid values of that argument.

Crashes. Columbus found a total of 153 crashes. After excluding the potential false positives, the total number of crashes become 137 (Table I). As presented in Table III, Columbus found crashes of 16 different types in 49 out of 60 apps in the AndroTest dataset. Compared to Stoat, EHBDroid, Ape, and TimeMachine, Columbus discovered 4.42, 5.48, 1.23, and 1.57 times more crashes, respectively. To acquire a better understanding of how the tools perform on individual apps, we calculated the number of apps for which each tool discovers the most number of crashes. While Stoat, EHBDroid, Ape, and TimeMachine finds the most crashes in 14, 10, 25 , and 21 apps, respectively, Columbus performs the best for the highest (45) number of apps.

False positive analysis. Our strategy of invoking callbacks directly, sometimes with artificially-prepared arguments, can potentially lead to false positives (FP), *i.e.*, generate spurious crashes that cannot be triggered when the app is normally exercised from the UI. Since Stoat, Ape, and TimeMachine are UI-driven testing tools, they always generate legitimate

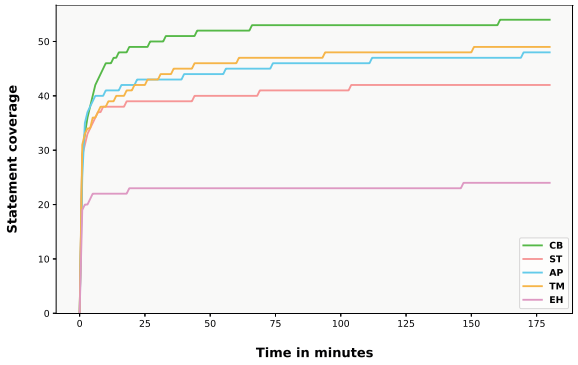


Fig. 8: Progression of coverage over time by all the tools on the AndroTest dataset. Tool codes are similar to Table I

crashes. For COLUMBUS, we identify two potential reasons for FPs and quantify their prevalence.

(i) Disabled UI elements. Since COLUMBUS does not access the UI state of the app, it may (incorrectly) invoke a callback cb_d associated with a widget W, which is disabled at the time of invocation. If such a callback cb_d exists in an app, then there exists another callback cb_e that calls `W.setEnabled()` to enable the widget. We found that only 71 (cb_e) out of 4,991 callbacks in our benchmark apps contain such calls. Now, `setEnabled` calls from inside the lifecycle callbacks are not problematic. Because, the latter is called by the Android framework, which enables the respective UI elements as part of the initialization of the app. Among those 71, only 4 callbacks are non-lifecycle ones, which is negligible with respect to the total number of callbacks.

(ii) Uninitialized nested object argument. If a callback expects an object argument of `classA` that we do not find in the heap, we create an instance a by invoking the class constructor C. However, instances created in this way may be partially uninitialized. Suppose, A contains a field $A.b$ of `classB`, which C leaves uninitialized. If the callback attempts to access $A.b$, then it will result in a `NullPointerException`. This is a spurious crash, because when the app is exercised from the UI, the framework would invoke the callback with a correctly constructed object. In case of the benchmark apps, we needed to create object arguments for only 207 (4.15%) out of 4,991 callbacks. Unfortunately, there is no straightforward way to estimate further how many of these callbacks require nested object arguments. Even then, since we already invoke object creation for a reasonably small number of callbacks, that makes the probability of such FPs minimal.

To investigate into our potential sources of FPs, we first collected all 55 crashes that are found only by COLUMBUS, but not by any of those tools. Then, we manually verified those reports to determine potential FPs. We call a report *legitimate*, if we can reproduce a crash with the same stack trace by exercising the app from the UI. To do that, we collected a sequence of callback invoked immediately before the crash from our tool's output log, and also reviewed the relevant part of the source code to seek further guidance. If we failed to reproduce the crash within a reasonable number of tries, we flagged the report as FP. Note that, this estimate is conservative and best-effort, because it includes true crash reports that we could not reproduce because of Android apps' inherent statefulness. At the end, we failed to reproduce 16 crashes out of total 153 crashes, which, even in the worst case, translates to a mere 10.46% FP rate. We argue that this amount of FPs is acceptable in practice, given the benefits (extra crashes, coverage) that our approach brings.

Category	Count
Education	27
Games	26
Personalization	18
Tools	17
Multimedia	11
Photography	4
Lifestyle	7
Health & Fitness	4
Food & Drink	4
Entertainment	6
Travel & Local	6
Business	2
Productivity	4
Others	4
Total	140

TABLE II: Real-world app categories

ID	Exception type	A	R
1	NullPointerException	52	22
2	IllegalStateException	16	26
3	ArrayIndexOutOfBoundsException	7	4
4	IndexOutOfBoundsException	10	2
6	CursorIndexOutOfBoundsException	10	-
7	UnsatisfiedLinkError	6	-
8	RuntimeException	1	2
9	IllegalArgumentException	15	4
10	ClassCastException	1	2
12	StaleDataException	3	-
13	ActivityNotFoundException	8	6
14	SQLiteDoneException	1	-
15	NumberFormatException	1	-
16	App Exceptions	6	2
	Total	137	70

TABLE III: Crashes found by COLUMBUS. A: AndroTest, R: Real-world dataset

RQ1: Compared to the state-of-the-art tools, COLUMBUS attains the highest coverage on average (58%), and discovers the most number of crashes (137) on the AndroTest dataset.

2) Performance on real-world apps: To understand the practicality of our approach, we tested COLUMBUS on the real-world dataset. In line with the previous approaches [31], [41], [19], we only considered the number of crashes discovered by our tool for this evaluation.

Crashes. As shown in Table III, we discovered a total of 70 crashes of 9 different types in 54 out of 140 apps, where `IllegalStateException` (37.14%) and `NullPointerException` (31.43%) are the most prevalent ones.

RQ2: COLUMBUS is able to find 70 crashes in 54 out of 140 real-world Play Store apps, belonging to 14 categories.

3) Effectiveness of dependency feedback: To show the effectiveness of the dependency feedback, we performed an ablation study by comparing COLUMBUS with COLUMBUS $_{\text{wd}}$, a modified version of our tool that runs without the dependency feedback. Table I presents the results of this experiment on the AndroTest dataset.

While the coverage attained by both COLUMBUS and COLUMBUS $_{\text{wd}}$ are comparable, the latter finds -3 fewer crashes than the former in 5 apps. By manually inspecting those apps—`Book-cat`, `qsettings`, `sanity`, `sftp`, and `aCal`, we can confirm that the additional crashes are correlated with the number of dependency relations discovered. In other words, due to higher than average (41 dependencies/app) number of dependencies being present in those apps, the dependency feedback could indeed help COLUMBUS in triggering more crashes. In addition, COLUMBUS achieved better coverage than any other tool for the first four apps.

RQ3: The dependency feedback used by COLUMBUS is useful for triggering crashes in apps, particularly for those ones with large amount of inter-callback dependencies.

VI. LIMITATIONS

Inferring correct value of the object fields. Currently, our *argument generation* module can only infer the correct values of the primitive arguments. However, it can be extended to support object arguments as well. Consider the callback: `onKeyDown(intkeyCode,KeyEventevent)`, which gets called when a key down event occurs. Now, `event.getUnicodeChar()` API returns the Unicode character c generated by that key event. If a callback has paths conditioned on c, we can infer its correct values by symbolizing the return value of the API. The inferred values can be used during testing to either dynamically set the correct value of the appropriate field of the `event` argument, or 'hook' the `getUnicodeChar()` API to alter its return value—exercising more paths in effect.

Creating values for login. There are Android apps which requires a userid and password to login first before one can explore its functionality. COLUMBUS in its current shape can not detect such a login prompt, and enter the username and password automatically to explore such an app. However, this is a limitation that we share with the existing state-of-the art tools, and an interesting direction for future work.

VII. RELATED WORK

Random. Random testing based techniques such as MONKEY [8] delivers random events. DYNODROID [29], in addition, considers system-level events, and monitors which events have registered listeners in the app to prioritize certain events depending on the context. PUMA [24] presents an automation framework that has support for custom dynamic exploration strategies. However, random testing strategies, though popular, often get stuck in a "local optima," making no further progress.

Model-based. Model-based testing approaches guide the exploration of the app by deriving a model of the app's UI. Though some techniques require this model to be provided manually [44], [51], [42], others reconstruct the UI model using dynamic app exploration [28], [40], [41], [26], [12], [32]. Other techniques also perform model abstraction via identifying the structural similarities between different layouts [20], model refinement by merging several UI interaction [22], and state recovery using snapshotting [19]. Model based testing techniques oftentimes suffer from state explosion if there are too many states in the app. Therefore, they need to strike a balance between model completeness and scalability.

Symbolic execution-based. Anand *et. al.* [13] concolically executes both the Android framework and the entire app, which is precise, but not scalable. In contrast, COLUMBUS does symbolic execution only within a callback to strike a balance between precision, and scalability. Another approach [25] starts the symbolic exploration in reversed order from the target blocks, and obtains the sequences of events to reach these targets. Additionally, several other techniques were introduced for the symbolic execution of the apps that include libraries as well [34], [21].

Hybrid. Similar to COLUMBUS, several approaches also employ hybrid techniques, *i.e.*, combination of static and dynamic strategies, for app exploration. In particular, [15], [27], [50], [47], [23] reconstruct the app model statically, followed by dynamic exploration. Other techniques use static analysis to discover dependencies between different application components, and use it during the dynamic exploration [15], [43], [25], [38], [23], [14]. Another guided exploration technique CAR [45] uses a static constraint analysis to keep the symbolic execution scalable and obviate the need for whole program symbolic execution. In contrast, COLUMBUS aims to maximize coverage similar to other app testing tools limiting the scope of the symbolic execution only within the callback and sets up the environment in an under-constrained manner. Moreover, during the dynamic exploration, COLUMBUS uses a type-guided object matching to supply an existing, well-formed object to the callback. Whereas, CAR resorts to a refinement-based construction of heap objects, guided by a crash-oracle. A crash resulting from a malformed object acts as a 'hint' to fix the shape of the object. EHBDROID [39] instruments the app statically to include callback invocations within the app code in order to invoke them directly. However, their technique is not generic, and suffers from limitations as discussed before.

VIII. CONCLUSION

This paper proposed COLUMBUS, a callback-driven Android app testing technique that improves over the state-of-the-art in three aspects: **(i)** systematically identifying the callbacks present in an app, **(ii)** inferring coverage maximizing primitive arguments, while generating object arguments in an Android API-agnostic manner, and **(iii)** providing *data dependency* and *crash-guidance* as 'feedback' to increase the probability of triggering uninitialized data related crashes, and preventing the tool from rediscovering same bugs, respectively. In our evaluation, COLUMBUS outperformed state-of-the-art model-driven, checkpoint-based, and callback-driven testing tools both in terms of crashes and coverage.

IX. ACKNOWLEDGMENTS

We want to thank our anonymous reviewers for their valuable comments and feedback to improve our paper. This research is supported in part by DARPA under the agreement number N66001-22-2-4037, by the NSF under award 2107101, Google ASPIRE Award, and by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project. The U.S. Government is authorized to reproduce, and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] Android app release statistics. https://www.statista.com/statistics/1020956/android-app-releases-worldwide/.

[2] Android statistics. https://www.businessofapps.com/data/android-statistics.

[3] angr, binary analysis framework. https://angr.io.

[4] Emma, a java code coverage tool. http://emma.sourceforge.net.

[5] Frida, dynamic instrumentation toolkit. https://frida.re.

[6] Google play store. https://play.google.com/.

[7] Logcat. https://developer.android.com/studio/command-line/logcat.

[8] Monkey. http://developer.android.com/tools/help/monkey.html.

[9] Source code of columbus. https://github.com/ucsb-seclab/columbus.

[10] Uiautomator. https://developer.android.com/training/testing/other-components/ui-automator.

[11] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Softw.*, pages 53–59, 2015.

[13] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12*, page 59, 2012.

[14] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, Martin Schäf, Ishan Banerjee, and Atif M. Memon. Lightweight static analysis for GUI testing. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012*, pages 301–310, 2012.

[15] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*, 2013.

[16] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[17] Wontae Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proc. of OOPSLA*, volume 2013, 2013.

[18] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[19] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 481–492. IEEE, 2020.

[20] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE, 2016.

[21] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. Android testing via synthetic symbolic execution. In *ASE 2018*, 2018.

[22] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[23] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. Improving automated GUI exploration of android apps via static dependency analysis. In *ICSME*, 2020.

[24] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the annual international conference on Mobile systems, applications, and services*, 2014.

[25] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis, ISSTA '13*, 2013.

[26] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for android test generation. In *ASE*, 2018.

[27] Duling Lai and Julia Rubin. Goal-driven exploration for android applications. In *ASE*, 2019.

[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *ICSE 2017*.

[29] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[30] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, 2014.

[31] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2016.

[32] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE 2003*.

[33] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[34] Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Softw. Eng. Notes*, pages 1–5, 2012.

[35] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[36] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.

[37] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). 11 2015.

[38] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. Patdroid: permission-aware GUI testing of android. In *ESEC/FSE*, 2017.

[39] Wei Song, Xiangxing Qian, and Jeff Huang. Ehbdroid: Beyond gui testing for android applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017.

[40] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps. *IEEE Trans. Software Eng.*, pages 1115–1137, 2022.

[41] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017.

[42] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. *ACM SIGSOFT Softw. Eng. Notes*, pages 1–5, 2012.

[43] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. Combodroid: generating high-quality test inputs for android apps via use case combinations. In *ICSE*, 2020.

[44] Lee J. White and Husain Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, 2000.

[45] Michelle Y. Wong and David Lie. Driving execution of target paths in android applications with (a) car. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2022.

[46] Baek Y and Bae D. Automated model-based android gui testing using multi-level gui comparison criteria. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[47] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. Multiple-entry testing of android applications by constructing activity launching contexts. In *ICSE*, 2020.

[48] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.

[49] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, 2013.

[50] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering (FASE) 2013*, 2013.

[51] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Trans. Software Eng.*, 2010.