Goal-Driven Exploration for Android Applications

Duling Lai
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
dlai@ece.ubc.ca

Julia Rubin
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
mjulia@ece.ubc.ca

***Abstract*—This paper proposes a solution for automated goal-driven exploration of Android applications – a scenario in which a user, e.g., a security auditor, needs to dynamically trigger the functionality of interest in an application, e.g., to check whether user-sensitive info is only sent to recognized third-party servers. As the auditor might need to check hundreds or even thousands of apps, manually exploring each app to trigger the desired behavior is too time-consuming to be feasible. Existing automated application exploration and testing techniques are of limited help in this scenario as well, as their goal is mostly to identify faults by systematically exploring different app paths, rather than swiftly navigating to the target functionality.**

The goal-driven application exploration approach proposed in this paper, called GOALEXPLORER, automatically generates an executable test script that directly triggers the functionality of interest. The core idea behind GOALEXPLORER is to first statically model the application's UI screens and transitions between these screens, producing a Screen Transition Graph (STG). Then, GOALEXPLORER uses the STG to guide the dynamic exploration of the application to the particular target of interest: an Android activity, API call, or a program statement. The results of our empirical evaluation on 93 benchmark applications and the 95 most popular GooglePlay applications show that the STG is substantially more accurate than other Android UI models and that GOALEXPLORER is able to trigger a target functionality much faster than existing application exploration techniques.

I. INTRODUCTION

Mobile applications (apps) have grown from a niche field to the forefront of modern technology. As our daily life becomes more dependent on mobile apps, various stakeholders, including app developers, store owners, and security auditors, require efficient techniques for validating the correctness, performance, and security of the apps.

Dynamic execution is a popular technique used for auditing and validating mobile apps [1], [2], [3], [4], [5]. For example, a security auditor often monitors the network traffic generated by an app to ensure that user-sensitive info is sent encrypted over the network and that it is sent to recognized third-party servers only [6], [7], [8], [9], [10]. It is common for the auditor to know which part of the app is responsible for the functionality of interest, but dynamically triggering that functionality is still a challenging task.

Consider, for example, a popular mobile personalization app, Zedge, which provides access to a large collection of wallpapers, ringtones, etc. and has more than a hundred million installs on Google Play [11]. A security auditor exploring this app can quickly determine that it uses the Facebook SDK [12] and allows the users to log in with their Facebook accounts. Such determination can be done by simply browsing the app code: the Facebook login is triggered via a particular API from the Facebook SDK [13]. Now, the auditor wishes to check what information is transmitted during the login process and what information is granted by Facebook when the user logs in into their account [14].

To perform this investigation, the auditor needs to generate a sequence of user gestures that navigate the app to the Facebook login screen. For Zedge, that entails a nontrivial sequence of steps shown in Fig. 1: first, scroll all the way down in the menu of all possible operations in Figs. 1(a-b) to the 11th menu item, then click the "Settings" button in Fig. 1(b) to open the "Settings" view, then click "ZEDGE account" in Fig. 1(c), and only then click on "Continue with Facebook" in Fig. 1(d).

Such a manual exploration, especially when one needs to analyze hundreds or thousands of different apps, is time-consuming. Existing testing frameworks, such as Espresso [15], UIAutomator [16], and Robotium [17], are not helpful in this scenario as they are designed to provide the user a way to *manually* script test cases and later repeatedly execute these test cases.

Automated app exploration techniques that exercise the app by generating user gestures and system events [18], [19], [20], [21], [22], [23], [24], [25], [26], [27] are of limited help as well. The goal of these techniques is to detect faults through a thorough exploration of different app behaviors [28]. However, when there is a large number of feasible paths to explore, these techniques will have difficulties to quickly navigate to the functionality of interest. For the Zedge example in Fig. 1, both the most prominent automated app exploration and testing tools, Sapienz [26] and Stoat [27], failed to reach the Facebook login screen after two hours of execution, even when configured to prioritize previously unexplored paths. That is because these techniques lack knowledge of which of the unexplored paths is most likely to lead to the functionality of interest and thus end up exploring a large number of possible feasible paths.

In this paper, we introduce a *goal-driven app exploration approach* and the associated tool, called GOALEXPLORER, which directly triggers a particular functionality of interest. GOALEXPLORER combines a novel approach for modeling an app User Interface (UI) as a *Screen Transition Graph* (STG) with a dynamic STG-driven app exploration technique. Given

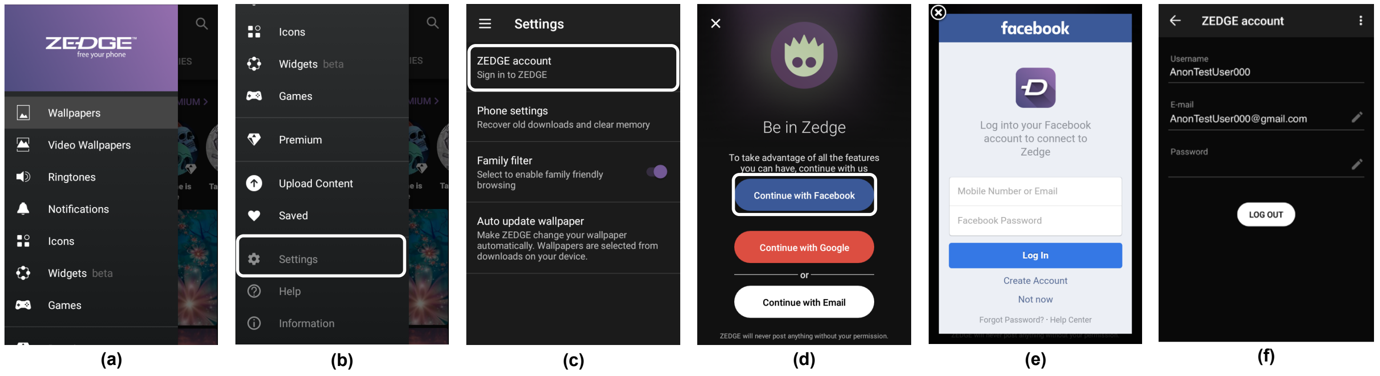


Fig. 1: Example App.

a particular target of interest – an Android activity, API call, or a code statement, it automatically builds an executable script that swiftly triggers the target.

The idea of building a static UI model of an Android app is not new by itself. To the best of our knowledge, A3E [19] was the first to build a static model of an Android app GUI, which only focused on Android activities. Gator [29] further extended this model to capture information about the menu items and the widgets that facilitate transitions between the modeled UI components, e.g., activities and menus.

However, the main building blocks of both these models are UI components rather than their composition into UI screens. As such, they do not accurately model the current Android UI framework [30] that allows composing screens from multiple components, such as fragments, menus, drawers, etc. Thus, they will misrepresent transitions originating in fragments, e.g., the transitions between screens in Figs. 1(b) and (c). Moreover, they do not model states and transitions triggered by background tasks and events (*broadcast receivers*, in Android terminology). For example, an app relying on the Facebook login mechanism can use a broadcast receiver to deliver login information to the interested app components, as done in Zedge example in Fig. 1: after the login is triggered in Fig. 1(e), the app broadcasts the login status and, in the case of a successful login, additional meta-information such as the user email address, to the *Controller* activity in Fig. 1(f). Without modeling broadcast receivers, a goal-driven exploration tool will fail to reach that screen.

Our analysis shows that more than 90% of the 100 top free Android apps on Google Play contain at least one fragment, background task, and broadcast receiver; 87% of the apps contain menus and 64% – drawers. Moreover, 63% of the apps have at least one screen transition originating from a fragment and 34% – from a background task or broadcast receiver. Modeling these behaviors is thus critical for the goal-driven exploration task and we do so in our work. Moreover, we introduce the concept of modeling UI screens as composition of UI components rather than individual UI components. We empirically show that, for our task, such approach is superior to the existing concepts of modeling app UI.

When compared to state-of-the-art dynamic exploration and testing techniques, i.e., Sapienz [26] and Stoat [27], GOALEXPLORER is able to explore a similar portion of each app, which attests to the accuracy of its statically-built model of screens and transitions. However, GOALEXPLORER is able to reach the functionality of interest substantially faster than the dynamic exploration tools. As such, GOALEXPLORER provides an efficient solution to the goal-driven exploration problem in mobile apps.

Interestingly, our analysis of network traffic associated with Facebook login in Zedge helped revealing a potential security vulnerability in this app: after receiving the OAuth [31] authentication/bearer token from Facebook, the app openly sends it to its server. Such behavior is discouraged by the Internet Engineering Task Force (IETF) [32] because any party holding the token can use it to access the user private information, without any further proof of possession. According to IETF, when sent over the network, bearer tokens have to be secured using a digital signature or a Message Authentication Code (MAC) [32]. Zedge is not following these guidelines; thus, a man-in-the-middle can steal the token and access private information of users logged into the app.

Contributions. This paper makes the following contributions:
1) It defines the problem of goal-driven exploration for mobile apps: efficiently triggering a functionality specified by the user.
2) It shows that existing approaches are insufficient for performing goal-driven exploration.
3) It presents a technique for constructing a static model of an app UI called *Screen Transition Graph* (STG) and using STG to generate an executable script that triggers a specific functionality of interest defined by the user.
4) It implements the proposed technique in a tool named GOALEXPLORER and empirically shows its effectiveness on the 93 benchmark apps used in related work and 95 top Google Play apps. Our implementation of the GOALEXPLORER and its experimental evaluation package are available online [33].

II. BACKGROUND

In this section, we give the necessary background on app component structure, lifecycle, and UI. We then discuss the static Android UI models used in earlier work [19], [29].

A. Android Applications

App Components. Android apps are built from four main types of components [34] – activities, services, broadcast

receivers, and content providers. *Activities* are the main UI building blocks that facilitate the interactions between the user and the app. *Services* perform tasks that do not require a UI, e.g., prefetching files for faster access or playing music in the background after the user has switched to other activities. *Broadcast receivers* are components that respond to notifications. A broadcast can originate from (i) the system, e.g., to announce that the battery is low, (ii) another app, e.g., to notify that a file download is completed, or (iii) from other app components, e.g., to notify that the login was completed successfully. *Content providers* manage access to data, e.g., for storing and retrieving contacts.

Component Lifecycle. Unlike Java desktop programs, an Android app does not have a main method; instead, each app component implements its own lifecycle events. The activity lifecycle [35] is shown in Fig. 2. Once an activity is launched, the system calls its `onCreate` method, followed by `onStart` and `onResume`. Developers specify the behavior of these methods, e.g., the UI setup is typically done in `onCreate`. The activity completes the initialization after `onResume`. It then enters a running state, and begins handling callbacks, e.g., those triggered by user inputs or by system changes. The activity is paused when it loses focus, and can eventually be stopped and destroyed by the operating system.

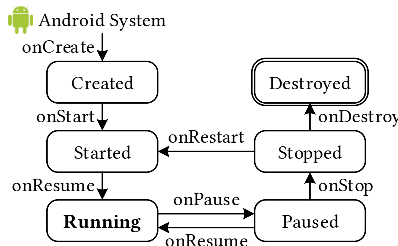


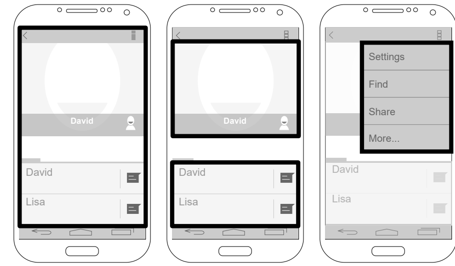
Fig. 2: Activity Lifecycle Events.

The lifecycle of services is similar to that of activities, except additional methods that handle service bindings, i.e., cases when other app components maintain a persistent connection to a service. Broadcast receivers have only one lifecycle event, `onReceive`, to handle received events. Content providers do not expose any lifecycle events.

The transition between Android components relies on `Intents`, which explicitly or implicitly specify the target component and the data to be passed to that component. Broadcast receivers are registered with intent-filters, which specify the types of events that the broadcast receiver handles.

App UI Structure. An application screen is represented by an activity "decorated" by additional UI components, such as fragments, menus, dialogs, and navigation drawers [34]. Fig. 3 depicts such a modular composition of screens. The solid box in Fig. 3(a) highlights the container activity; its two hosted fragments are highlighted in Fig. 3(b).

A fragment is a re-usable UI component with its own lifecycle and event handling. A fragment can be instantiated in multiple activities and must always be hosted by at least one activity. Its lifecycle is directly affected by the owner activity, e.g., when the owner activity is destroyed, all of its hosted fragment instances are also destroyed. Fragment instances can be added, replaced, or removed from the host activities through the `FragmentTransaction` APIs, forming different screens of the same activity. Multiple fragments can exist in a single screen, as in Fig. 3(b); a screen can have any number of fragments or no fragments at all.



(a) Activity. (b) Fragments. (c) Menu.

Fig. 3: Application Screens.

Menus and navigation drawers are UI components used primarily to provide additional options to the user, such as adjusting settings or launching new activities. Fig. 3(c) shows the activity with a menu opened, which allows the user to access system-wide actions, such as settings and search. Fig. 1(a) shows a drawer that occupies most of the screen and contains buttons for navigating to other parts of the app.

Dialogs are small windows that prompt the user to take an action before the app can proceed with its execution, e.g., to agree to granting access to the device location. Menus, navigation drawers, and dialogs must be hosted by either an activity or a fragment.

UI components are composed of widgets, such as `Buttons` and `TextViews`. `ViewGroups` are invisible containers used for grouping the widgets and specifying their layout before placing them inside the UI components. Users mostly interact with UI widgets and are typically unaware of the app and UI components and their lifecycle.

B. Existing Android UI Models

We now describe existing approaches for modeling Android UI. Activity Transition Graph (ATG) introduced by Azim and Neamtiu in A3E [19] captures app activities and transitions between these activities. It does not model other app components and does not model actions that trigger the transitions, i.e., which user input triggers the transition between activities.

Window Transition Graph (WTG) introduced by Yang et al. in Gator [29] extends ATG by addressing some of these limitations: it models the event handler for each transition and also incorporates menus and dialogs as distinct nodes in the model. However, it does not consider fragments, drawers, services, and broadcast receivers.

Due to these reasons, for the Zedge example in Fig. 1, both ATG and WTG will only include three nodes that correspond to the app activities: "Controller", which corresponds to the activity in Figs. 1(a-c) and (f), "Authenticator", for the activity in Fig. 1(d), and "Facebook", for the activity in Fig. 1(e), which is implemented inside the Facebook SDK. ATG and WTG models will include no transitions, as all transitions in this app originate from fragments. As such, none of these models will contain the path that reaches the Facebook activity.

Moreover, the models do not account for broadcast receivers and thus no activity that follows the Facebook login, e.g., the one in Fig. 1(f), is reachable.

An even more severe drawback of the existing UI models is that they treat UI components, such as activities and menus, as separate nodes. These models do not accurately represent the composition of UI components into screens, which hinders their applicability for the goal-driven exploration scenario: as the representations do not include enough details to extract an executable path. In Sect. III-A, we will discuss this limitation in more detail, outline alternative ways to model Android UI components and their relationships, and then present our proposed solution, Screen Transition Graph.

III. Goal Explorer

The high-level overview of GoalExplorer is presented in Fig. 4. It obtains two inputs: the APK file of the app and the target functionality of interest, which can be provided as an app activity, an API call, or a code statement. For example, when monitoring the network traffic related to the Facebook login scenario in Zedge, we set as target a Facebook API call, as discussed in Sect. I.

In the first step, GoalExplorer statically constructs the Screen Transition Graph (STG) of the app (the *STG Extractor* component in Fig. 4). It then maps the target(s) to (possibly multiple) nodes of the graph (the *Target Detector* component). Finally, it uses the graph to guide the dynamic exploration to a reachable target node, starting with the one that has the shortest path from the initial screen (the *Dynamic Explorer* component). In what follows, we first introduce the STG and discuss our design choices. We then describe the *STG Extractor*, *Target Detector*, and *Dynamic Explorer* components.

A. Screen Transition Graph

As discussed in Sect. II, both Window Transition Graph (WTG) and its predecessor, Activity Transition Graph (ATG), model UI components as distinct nodes in the graph [19], [29]. Moreover, they do not model fragments, drawers, services, and broadcast receivers. As such, for the example app in Fig. 1, these graphs will only contain three activity nodes: elements #2, #3, and #4 presented with solid lines in Fig. 5.

Our "direct" extension to this model, which we refer to as WTG-E, introduces a number of additional elements, presented with dotted lines in Fig. 5. For conciseness of the discussion, we include in this snippet only parts of the Zedge app shown in Fig. 1:

– Assuming drawers are handled similar to menus, as they correspond to similar UI concepts, element #1 in Fig. 5 would represent the drawer in Figs. 1(a,b). Adding drawers would also lead to creating transitions labeled "open drawer" and "close drawer" between elements #1 and #2.

– The transition labeled "settings" from element #1 to #2 would be contributed by an analysis of the drawer behavior that launches a new activity.

– The "Account Manager" broadcast receiver, responsible for notifying other app components when Facebook authentication is completed, would be represented by element #5, and its corresponding incoming and outgoing transitions.

– The fragments contributing to each activity could be represented by inner boxes inside elements #2 and #3, as only these two activities have fragments in this example. Representing the fragments would allow the model to capture transitions from element #2 to #3 and #3 to #4, as these transitions are triggered from fragments embedded in each corresponding activity.

However, even the extended WTG-E model is sub-optimal for producing the execution scenario that leads to the Facebook login screen in the Zedge example. That is because WTG-E's main nodes are UI components rather than their composition to screens. Thus, the model cannot represent the actual screens, omitting important information used for exploration.

For example, from the WTG-E graph in Fig. 5, one can conclude that the "close drawer" action for element #1 that leads to element #2 – the "Controller" activity, can be followed by the "ZEDGE account" action to reach element #3 – the "Authenticator" activity. Such execution is not possible in practice as the "ZEDGE account" action is only enabled when the "Controller" activity is opened with the "settings" fragment. However, when the app execution starts, the "Controller" activity is rather opened with the "browse" fragment, thus there is no option to press the "ZEDGE account" button after closing the drawer. Without distinguishing between these different states of the "Controller" activity (element #2), one cannot successfully explore the application.

The Screen Transition Graph proposed in this work addresses this limitation. It models application screens by representing the composition of the container activity, hosted fragments, menus, navigation drawers, and dialogs on each screen. Fig. 6 shows the STG representation of the Zedge snippet in Fig. 1. Unlike in the WTG-E model, the "Controller" activity here is represented by five different elements: #1–4 and #8. These elements correspond to different compositions of fragments and drawers hosted by the activity. This representation clarifies that the only possible path to the Facebook login activity (element #6 in Fig. 6) is to start from the "Controller" activity (element #1), open the drawer (arriving at element #2), then press the 'settings" option (arriving at element #3). Only after that one can transition to elements #4, #5, and #6.

Moreover, after the Facebook login is completed, the execution arrives at element #8, which is another instance of the "Controller" activity, but with the "Account" fragment. It is clear from this representation that the user cannot press "ZEDGE Account" again, as can mistakenly be concluded from the WTG-E representation in Fig. 5.

This example demonstrates that the STG representation is more accurate than the original WTG, and even WTG-E, for producing execution scripts in the goal-driven exploration scenario. We empirically evaluate the differences between these representations in Sect. IV. Below, we give a more formal definition of STG.

In STG, the main nodes $\mathbb{V}_s$ represent app screens and the supporting nodes $\mathbb{V}_r$ and $\mathbb{V}_b$ represent services and broadcast receivers, respectively. Each screen node in $V_s \in \mathbb{V}_s$ consists

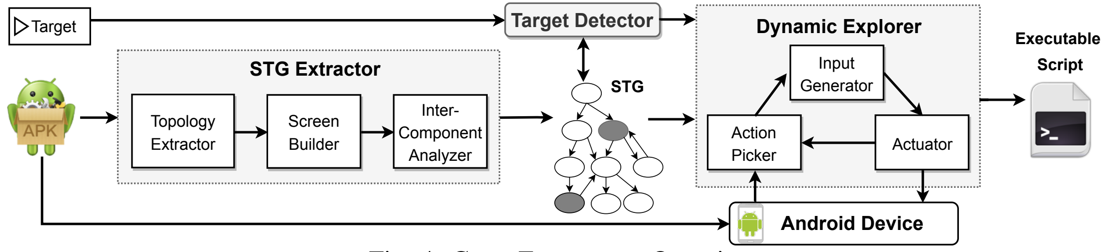


Fig. 4: GOALEXPLORER Overview.

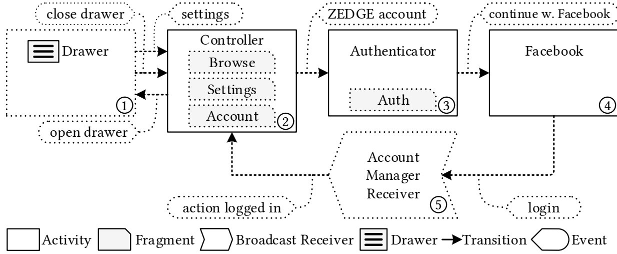


Fig. 5: WTG (solid lines) and WTG-E (all lines) of Zedge.

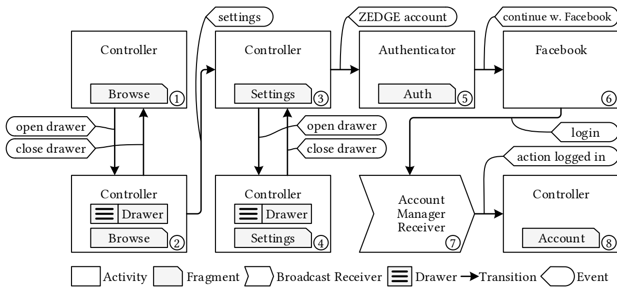


Fig. 6: STG of Zedge.

of one container activity V_s^A, zero or more fragments V_s^F, zero or one menu/drawer V_s^M, and zero or one dialog V_s^D. The collection of all screen nodes corresponds to possible screens allowed in the app. Intuitively, an activity with n fragments and one drawer could correspond to 2×2^n different screens (all combinations of fragments, with and without a drawer). Of course, not all combinations are possible in a particular app, and we describe our screen extraction algorithm in Sect. III-B.

Supporting nodes are required as transitions between screens can pass through these elements. The edges E in STG correspond to transitions between nodes $\mathbb{V} = \mathbb{V}_s \cup \mathbb{V}_r \cup \mathbb{V}_b$. Each edge $e \in E$ has a label e_τ which represents the event that triggers the transition: either a user action, e.g., pressing the "ZEDGE account" button to move from element #3 to #5, or an intent that triggers the broadcast receiver event, e.g., the notification received by the system for transitioning from element #7 to #8. Transitions that are triggered automatically, e.g., when a service opens a UI dialog without any notifications or clicks, have no event triggers in the model: $e_\tau = \emptyset$.

B. STG Extractor

We now describe our approach for constructing STG. The *STG Extractor* consists of three main components outlined in Fig. 4: *Topology Extractor*, *Screen Builder*, and *Inter-Component Analyzer*.

Topology Extractor. *Topology Extractor* identifies the main app components, i.e., activities, services, and broadcast receivers, and their call graphs. It relies on FlowDroid's implementation [36] to extract the components from both the app source code and xml files and to associate each component with its entry points, i.e., lifecycle events and application-specific callback methods. It uses FlowDroid to also build call graphs of all component's entry points. For example, an activity may register callbacks that get invoked when a button is pressed or a location update is received; the respective callback handler would then be linked to the activity's call graph and analyzed between the onResume() and onPause() lifecycle events of the activity.

App methods annotated with "@JavascriptInterface" can also be triggered by JavaScript-enabled WebViews [37]. As the invocation of these methods depends on the WebView content delivered at runtime, *Topology Extractor* conservatively extends the call graphs by assuming that any JavaScript-enabled method in a component can be called by any JavaScript-enabled WebView in the same component.

After the components and their call graphs are extracted, nodes corresponding to services and broadcast receivers are added directly to STG, in $\mathbb{V}_r$ and $\mathbb{V}_b$, respectively. App screens in $\mathbb{V}_s$ are constructed next.

Screen Builder. A key insight of our approach is modeling the UI screens and the transitions between the screens. To this end, *Screen Builder* analyzes the call graph of each activity, collecting the hosted fragments, menus, dialogs, and navigation drawers.

Fragments can be defined statically in the activity layout file or dynamically in code. *Screen Builder* starts by extracting the layout files using AXMLPrinter [38] and then identifies fragments they declare. To collect fragment declarations in code, *Screen Builder* scans the call graph of each activity, identifying fragment transaction methods that add, remove, or replace fragments. The full list of such methods, collected from the Android Developer website [39], is available in our online appendix [33].

The activity call graphs are scanned in multiple iterations. First, *Screen Builder* analyzes the activity lifecycle events triggered when an activity is initialized: from onCreate() to onResume(). For each event, it identifies fragment transaction methods m that add, remove, or replace fragments; it then performs an inter-procedural, context- and flow-sensitive analysis on the calling path of m to identify the Java type of the fragment(s) handled in m. Once all fragments from the

onCreate() to onResume() lifecycle events of an activity A are processed, *Screen Builder* creates the "base" screen node V_s in $\mathbb{V}_s$ for the activity A, with V_s^F containing a (possibly empty) list of the identified fragments. This "base" screen node is displayed when the activity is started.

Lifecycle methods issued when the activity is paused, stopped, or resumed can further add, remove, or replace fragments in the "base" screen. *Screen Builder* thus analyzes possible lifecycle method invocation chains: "onPause → onResume" and "onPause → onStop → onRestart → onResume" (see Sect. II) to extract the fragments from each of these chains. If the fragments in a chain differ from the fragments in the "base" screen node V_s, it adds a new screen node $V_{s'}$ for the activity A, which contains a union of fragments in V_s and those identified in the chain. It also adds a transition between V_s and $V_{s'}$, with an empty label, as this transition is triggered automatically by the system.

Fragments in callback methods are handled similarly. Since the order of callbacks cannot be predicted, *Screen Builder* assumes the callbacks can occur in any order; it thus creates a new screen $V_{s''}$, if it does not exist yet, for any possible order of callback methods which modify fragments of the "base" screen. *Screen Builder* also creates transition edges between these screens and sets the transition label e_τ to be the action triggering the corresponding callback.

Finally, *Screen Builder* analyzes each screen V_s to identify its menus, drawers, and dialogs. To this end, it analyzes the call graphs of the screen's activity V_s^A and all its collected fragments V_s^F, to identify methods initializing menus, drawers, and dialogs. When such a method is found, *Screen Builder* copies the screen V_s for the activity A into $V_{\hat{s}}$, adds the found menu, drawer, or dialog to $V_{\hat{s}}$, adds $V_{\hat{s}}$ to the set of all screen nodes $\mathbb{V}_s$, and creates a transition edge between V_s and $V_{\hat{s}}$. The transition label e_τ is set to be the action that triggers the callback method.

Inter-Component Analyzer. After the previous step, we have collected all screen nodes and the transitions between screen nodes of the same activity but with different fragments, menus, drawers, and dialogs. The *Inter-Component Analyzer* collects the inter-component transitions between nodes corresponding to different Android components, e.g., screens of different activities, services, and broadcast receivers. These transitions are performed via inter-component communication (ICC) methods and we rely on FlowDroid's integration with IccTA [40] to identify ICC links between app components.

For each link where the source and target are services and/or broadcast receivers, *Inter-Component Analyzer* simply creates the corresponding transition edge e in STG, If the target of the transition is a broadcast receiver, the transition label e_τ is set to be the broadcast which triggers the event, as specified in the broadcast receiver intent-filter. If the target of the transition is a service, $e_\tau = \emptyset$, as this transition is triggered "automatically", without any user or system event.

Activities are represented by a number of screens and thus require a more nuanced treatment. If a source of an ICC link is an activity A, *Inter-Component Analyzer* identifies the activity entry point p from which the communication originates. It then finds all screen nodes in STG that correspond to A, which can be reached from the "base" screen node of A via transitions associated with the action that triggers p. It adds a transition from all these nodes to the nodes that represent the targets, labeling them with the action that triggers p. When the target is an activity as well, *Inter-Component Analyzer* finds only the "base" screen node of that activity and creates a transition to that node. That is because when a new activity is launched, it starts in the initial screen; transitions between different screens of the target activity are already handled by *Screen Builder*.

C. Target Detector

When the exploration target is specified in a form of an activity, *Target Detector* simply traverses and marks all screen nodes that correspond to that activity. Reaching any of these nodes will be considered reaching the target. For targets given as an API, *Target Detector* scans the app call graph to find all statements that invoke the given API. For targets given as a single statement, *Target Detector* simply finds the statement in the app call graph.

Next, it maps the identified statements to STG nodes as follows: if a statement is identified in a lifecycle event of a component, it marks all nodes that correspond to that component as targets. If a statement is in a callback method, simply reaching the component is insufficient as one also needs to invoke the callback method itself. Therefore, *Target Detector* identifies the STG transitions labeled by the action that triggers the callback and sets the destination of these transitions as the targets.

D. Dynamic Explorer

Given an STG of the app and a set of targets, *Dynamic Explorer* performs goal-driven exploration guided by the STG, to trigger at least one of these targets. It also records an executable script which navigates the app to the target; the script can be replayed without consulting with the static model or performing any further analysis.

Dynamic Explorer has three main components outlined in Fig. 4: *Action Picker*, *Input Generator*, and *Actuator*, executing in a loop until a target node is reached.

Action Picker. In each iteration, *Action Picker* first evaluates all possible actions enabled from the current screen. To this end, it retrieves the current activity and fragments using an adb [41] `shell dumpsys` command and reads the screen UI hierarchy using UIAutomator [16]. This information is used to verify that the exploration arrived at the intended STG node or map the current screen to the corresponding node in STG.

If *Action Picker* is already "locked" on a particular target, it performs a breadth-first search to find all paths from the current screen node V_s to that target and sorts the paths from the shortest to the longest. It picks the next action from the shortest path that leads to the new screen node $V_{s'}$ and checks if the action is available on the screen. If so, it proceeds to the next step: *Input Generator*.

Some actions only become enabled after interacting with a screen. For example, only after adding an item to the shopping

cart the "checkout" button will be enabled. As STG does not model the correct order of user events (i.e., clicks, scrolls, etc.) on a screen, *Action Picker* dynamically interacts with the screen in an attempt to change its state and activate the required action. Like other dynamic exploration tools [22], [27], it adopts a weighted UI exploration strategy, which picks the next event for a particular widget based on the event type, past execution frequency, and the number of new widgets the event unrolls. To this end, we adopt the weights used by Stoat [27] and list them online [33].

If $V_{s'}$ still cannot be reached after a certain number of attempts (currently set to 50 iterations), *Action Picker* backtracks and proceeds to the next path. If no action is available on any of the paths reaching to the current target or if there is no "working" target set yet, *Action Picker* performs a breadth-first search to find the next available target and repeats the search for that target. It returns "unreachable" if no action leading to any of the targets is found.

Input Generator. When an action is picked, *Input Generator* checks whether specific textual inputs are necessary to successfully trigger the transition. First, as many Google Play apps require the user to log in for accessing some app features, we equip GOALEXPLORER with the ability to handle logins, assuming that login credentials are supplied as an optional input. To this end, *Input Generator* analyzes the UI hierarchy of the current screen to search for textual input fields, such as `EditText`. It further evaluates whether the textual inputs requires login credentials. Similar to prior work [42], this is done by checking whether the widget ID, text, hint, label, and content description match the regular expressions associated with login and password strings, e.g., "username", "user id", etc. The exact list of expressions we used is online [33]. If a match is found, *Input Generator* enters the credentials into the corresponding text fields. Otherwise, it feeds random strings to all textual input fields on the current screen.

Actuator. *Actuator* fires the action selected by the *Action Picker*. Its implementation extends the navigation framework of Stoat. The original Stoat only supports Android API level 19 (Android 4.4), and we extend it to support newer Android platforms. *Actuator* also records all successfully triggered actions, producing an executable test scripts that contain all actions necessary for reaching the target. The script can be replayed without any further analysis and without reliance on the static model.

IV. EVALUATION

We now describe our experimental setup and discuss evaluation results. Our aim is to evaluate GOALEXPLORER both quantitatively and qualitatively, answering the following research questions:

RQ1 (Coverage): What is the fraction of targets GOALEXPLORER can reach and how does that compare with the baseline approaches?

RQ2 (Performance): How fast can GOALEXPLORER reach the target and how does that compare with the baseline approaches?

RQ3 (Accuracy-STG): What is the fraction of STG false negative and false positive transitions, i.e., existing transitions that are absent in STG and STG transitions that do not exist in reality, respectively? What are the reasons for such cases?

RQ4 (Accuracy-Dynamic): What is the fraction of true positive transitions, i.e., STG transitions that exist in reality, that GOALEXPLORER cannot trigger dynamically? What are the reasons for such cases?

A. Experimental Setup

Subject Apps. To answer our research questions, we use Stoat's benchmark [27] that consists of 93 open-source F-Droid [43] apps: 68 of these apps are from the survey of Choudhary et al. [44] and de facto became a standard benchmark in analyzing automated testing tools; the authors of Sapienz used these apps for their evaluation as well [26]. The additional 25 apps were introduced by the Stoat authors to further extend that benchmark.

To extend our evaluation to real applications commonly used in practice, we also downloaded the 100 most popular free apps from the Google Play store as of July 31, 2018. We excluded from our analysis five apps that failed to run on the emulator (the decision to use emulators for the experiments is discussed in the *Environment* sub-section below), arriving at the set of 93 benchmark and 95 Google Play apps. A detailed description of these apps is available online [33].

Exploration Targets. We experiment with two scenarios. In the first one, we set each activity of the subject apps as the target, one at a time, and repeat the experiment for all activities in an app. That allows us to align our results with those commonly reported in related work. Then, we experiment with setting a code statement as a target, investigating a more nuanced, goal-driven exploration scenario which motivates our work. To this end, we pick the *URL.openConnection* API that is used by many apps to send and receive data over the Internet; this API is also used by the Zedge app in our motivating example to send the Facebook authorization token. We identify all occurrences of this API in the subject apps and set them as targets, one at a time.

Methodology and Metrics. For the quantitative evaluation in **RQ1** and **RQ2**, we perform two main experiments. First, we compare the exploration that is based on STG with the exploration based on the WTG and WTG-E models described in Sect. III-A. As the original WTG model introduced by Gator [29] is not compatible with the current Android API level, Gator failed to run for most of the Google Play apps. We thus re-implemented the WTG extraction process for apps with API level 23. We then extended WTG to handle fragments, services, broadcast receivers, etc., producing the WTG-E representation, as described in Sect. III-A. That is, we compare STG with two static models: WTG – our re-implementation of Gator's model to handle modern apps, and WTG-E – our extension of WTG to handle fragments, drawers, services, and broadcast receivers.

In a second line of experiments, we compare GOALEXPLORER with the state-of-the-art automated exploration tech-

niques: Sapienz [26] and Stoat [27]. As these tools cannot be configured to reach a particular target without dynamically exploring the app first, we run the tools on each app and record times when the tools reach each new target. We stop the execution when a tool does not discover any new targets for the period of one hour and proceed to the next app. We do not restart the tools every time they reach a new target as we believe such a comparison is more favorable for these tools.

We use configuration parameters specified by the papers describing the tools and kept default values for parameters not described in the papers. We also experimented with modifying the parameters, to ensure the baseline is most favorable for the goal-driven exploration task, e.g., by increasing the sequence length and population size for Sapienz and prioritizing coverage over test diversity for Stoat. As these adjustments did not affect the results, we kept the default configurations.

Moreover, we extended both tools to handle login screens, as we did for GOALEXPLORER (see Sect. III-D). Specifically, we added the same login logic that we used for our tool to that of Stoat. As the relevant code of Sapienz is not open-sourced (the motif part is only available as binary), we implemented a login detection component which runs in parallel to Sapienz and constantly checks if the execution arrived at the login screen. When it does, we pause Sapienz and handle the login flow as we do for GOALEXPLORER and Stoat. The execution time of this operation is negligible and is comparable to the handling of logins in GOALEXPLORER and Stoat.

To answer **RQ1**, we measure the fraction of targets reachable by each tool per app. Then, to answer **RQ2**, we measure the time it takes to trigger each *reachable* target. To ensure a fair comparison, we report the average time-to-target only for targets that are reachable by all tools.

For **RQ3** and **RQ4**, we manually establish the ground truth of all reachable `URL.openConnection` statements for 36 open-source F-Droid apps. To arrive at a representative and practically valuable subset of apps, we picked all F-Droid apps from our benchmark which are also available on Google Play. We focused on F-Droid apps as establishing ground truth manually is unrealistic for closed-source Google Play apps because these apps are largely obfuscated and manually tracking reachability of statements would not be reliable.

For the selected 36 apps, the first author of this paper collected the set of all `URL.openConnection` statements and manually identified those that are reachable by both running the app and inspecting its code; this manually established ground truth was cross-validated with another member of the research group. We then analyze and classify the reasons preventing GOALEXPLORER from triggering reachable targets and cases when GOALEXPLORER encounters spurious STG paths that cannot be followed due to the over-approximation made by our model construction. The set of the analyzed apps and the constructed ground truth is available online [33].

Environment. We run all tools on a 64-bit Ubuntu 16.04.4 physical machine with a 20-core CPU (Intel Xeon) and 512GB of RAM. As we run each app for up to one hour per target for each of the compared tools, the total evaluation time is in the thousands of hours only to answer RQ1 and RQ2. Therefore, we run the experiments on six Android emulators, to allow running multiple executions in parallel. We allocate 64GB of RAM for each emulator.

B. Results

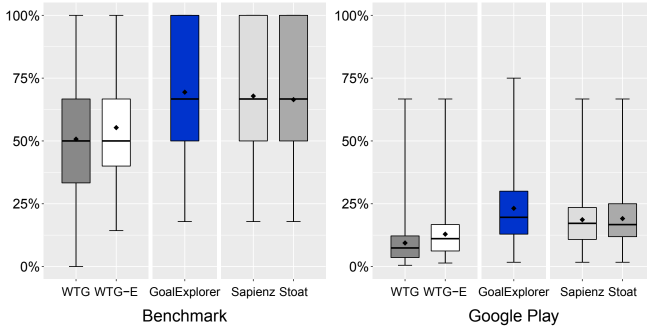
RQ1 (Coverage): Fig. 7(a) shows the box and whisker plots representing the minimum, median, mean, and maximum fraction of activities reached by goal-driven exploration under three different models: WTG, WTG-E, and STG (GOALEXPLORER). It also shows the fraction of activities reachable by the state-of-the-art dynamic exploration tools: Sapienz and Stoat. The results are plotted separately for the 93 benchmark F-Droid apps and the 95 Google Play apps.

For the benchmark apps, GOALEXPLORER reaches 69.4% of activities per app on average. For comparison, WTG and WTG-E reach 50.7% and 55.3% of activities on average per app, respectively. The differences between the UI models is much more substantial for the Google Play apps. WTG can only cover around 9% of app activities. WTG-E increases the activity-level coverage to 12.9% on average, as it is able to deal with fragments, services, and broadcast receivers. Finally, GOALEXPLORER reaches the highest activity-level coverage of 23.2% on average, due to its accurate representation of app screens. This result demonstrates that only extending existing static models with handling of fragments, services, and broadcast receivers, as we did in WTG-E, is less beneficial than the full set of solutions GOALEXPLORER applies.

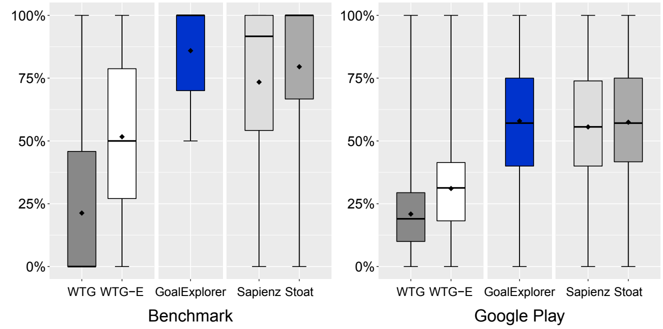
Notably, GOALEXPLORER achieves similar activity-level coverage as the dynamic exploration tools for both benchmark and the Google Play apps: for the benchmark apps, Sapienz and Stoat cover 67.9% and 66.5% of activities per app on average, respectively; for the Google Play apps, both tools reach around 19% of activities, on average, while GOALEXPLORER covers 23.2%. That is an encouraging result for GOALEXPLORER, showing that the STG model it builds statically is accurate and comparable with the models constructed during the dynamic exploration.

Fig. 7(b) shows similar data for the fraction of reachable `URL.openConnection` statements in an app. Here, GOALEXPLORER largely outperforms other static UI models, triggering 85.9% target API statements on benchmark apps, compared with 21.3% and 51.7% for WTG and WTG-E, respectively. With its 85.9% coverage, it also slightly outperforms the dynamic tools, Sapienz (73.4%) and Stoat (79.5%).

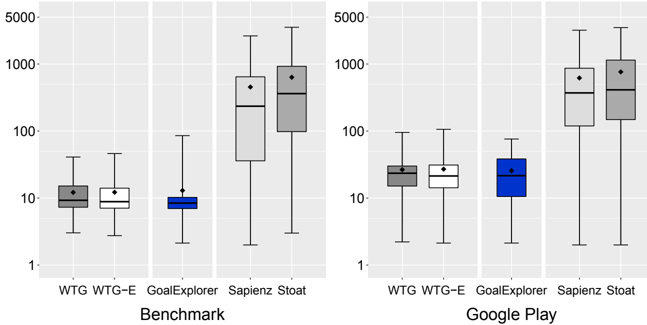
For GooglePlay apps, GOALEXPLORER also substantially outperforms other UI models. It performs comparable with dynamic tools and triggers 57.9% of the target statements on average, while Sapienz triggers 55.6%, and Stoat – 57.5%. Our inspection of the results shows that statement-level coverage is substantially higher than activity-level coverage because many of the `URL.openConnection` statements are located inside libraries or utility classes and thus have many possible paths that can reach them. As the target is triggered if any of the paths can reach it, this task appears to be “easier” than triggering all activities.



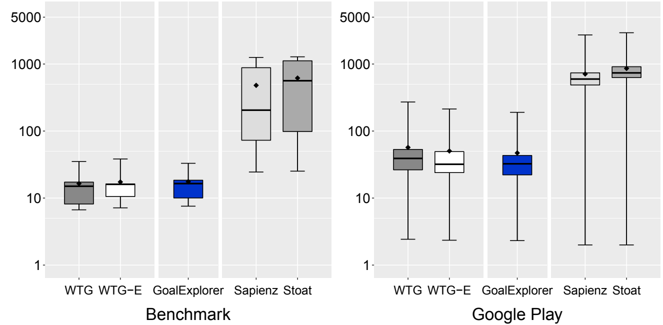
(a) Fraction of reachable activities in an app (higher is better).



(b) Fraction of reachable *openConnection* statements in an app (higher is better).



(c) Average time, in seconds, to reach an activity (lower is better).



(d) Average time, in seconds, to reach an *openConnection* (lower is better).

Fig. 7: Coverage and performance results for the benchmark and GooglePlay apps.

> To answer RQ1: Our experiments show that using the STG model allows GoalExplorer to trigger substantially more targets than with other statically-built Android UI models, namely, WTG (Gator) and WTG-E. Moreover, GoalExplorer is as effective as the dynamical tools in the ability to reach a certain target.

RQ2 (Performance): To evaluate performance, we measure the minimum, median, mean, and maximum time, in seconds, it takes GoalExplorer to reach a target activity or statement. We only present times for targets that are reachable by all of the compared tools.

Figs. 7(c) and 7(d) present the results of the comparison for a target activity and statement, respectively. The results are presented on a logarithmic scale and clearly show that GoalExplorer can reach a target much faster than dynamic tools: for activities, it takes GoalExplorer 13 seconds on average on the benchmark apps and 25.6 seconds on the Google Play apps, compared with 453.7 and 619.5 seconds for Sapienz and 634.3 and 763.5 seconds for Stoat, respectively. GoalExplorer reaches target statements in 17.6 seconds on average on the benchmark apps and 47.1 seconds on the Google Play apps, compared with 480.7 and 710.3 seconds for Sapienz and 617.7 and 860.9 seconds for Stoat, respectively. Not surprisingly, GoalExplorer performs comparable to other static models – WTG and WTG-E, as we only consider targets that are reachable given the tools' model. Our experiments also show that it takes GoalExplorer around 93 seconds per app to build the STG, which contains 116.7 nodes and 179 transitions on average. Even given this time, it outperforms dynamic tools and reaches substantially more targets than other static tools.

> To answer RQ2: GoalExplorer can trigger exploration targets substantially faster than dynamic exploration tools and on par with other tools based on a static model.

RQ3 (Accuracy-STG): To evaluate the STG accuracy, we manually identified all paths leading to `URL.openConnection` statement in 36 apps. The analyzed apps contain 127 such statements in total (3.53 per app). Out of those, 102 are reachable (2.83 per app), 21 are in library methods that apps do not use (which occurs in 8 apps), and the remaining 4 are in unused app methods (all in the same app).

Below, we classify the reasons for (1) transitions that STG misses (false negatives) and (2) STG transitions that do not exist in reality (false positives). We also list the implication of false negative transitions on the ability of GoalExplorer to reach the desired target.

(1) STG False Negatives. When a correct transition does not exist in the model, GoalExplorer cannot construct a complete path to the target and thus will not attempt to follow that path. In our sample, that happens in eight cases in total, in five apps. The reasons for missing a transition are:

Reflection: 2 transitions (in one app) are missing due to the limitation in handling complex reflective calls. GoalExplorer relies on call graph construction implemented by FlowDroid, which only handles reflective call targets for constant string objects.

Unmodeled Component: 2 transitions (in two apps) are missing due to unmodeled components, such as `App Widgets`, which are miniature app views that can be embedded in other apps, e.g., in the home screen. GOALEXPLORER, as well as all other tools, does not model these components as they are placed outside of the main app.

Intent Resolution: 4 transitions (in two apps) are missing due to failures in resolving the targets of intents, i.e., which component is to be launched by the intent. GOALEXPLORER relies on a constant propagation technique from IC3 [45] to resolve the intents and identify the targets. When the constant propagation fails, the correct transitions are missing in STG.

GOALEXPLORER finds alternative paths to avoid missing the targets for 4 out of the 8 missing transitions in STG; the remaining ones lead to four missed targets – two are due to reflection and two as the consequence of unmodeled components. Sapienz and Stoat cannot trigger the latter two missed targets either.

(2) STG False Positives. GOALEXPLORER identifies 42 spurious transitions, in 27 apps (1.55 transitions per app that contain false positive transitions, and 1.17 transitions per app overall), that do not exist in reality:

Callback Sequence: in 28 cases (17 apps, 1.64 transitions per app), incorrect transitions in STG are caused by over-approximation in modeling the sequence of the callbacks. Since the order of callbacks cannot be predicted statically, GOALEXPLORER assumes that the callbacks can happen in any order (see *Screen Builder* in Section III-B), producing screens in STG that are not feasible in practice.

Fragment Properties: in 14 cases (7 apps, 2.33 transitions per app), incorrect transitions are because STG only models the set of fragments but does not track the properties of each fragment. For example, apps can hide the fragments by setting its visibility properties, i.e., changing the visibility level to `View.GONE`. Our model thus over-approximates the set of visible fragments on a screen.

While STG over-approximates the set of possible transitions, all these transitions are "ignored" by GOALEXPLORER during exploration – by backtracking and selecting another path. None of the transitions analyzed above impacted the GOALEXPLORER's ability to triggering the targets. However, such spurious transitions, as well as transitions that cannot be triggered (RQ4), prolong the exploration, as it takes time to resolve the false positives dynamically. Our results demonstrate that the execution time of the tool is not substantially affected by that and GOALEXPLORER still able to reach the target substantially faster than dynamic tools.

> To answer RQ3: The main reasons for STG's false negative and false positive transitions are imprecisions of the underlying static analysis tools. GOALEXPLORER succeeds to find alternative paths in four out of eight false negative cases.

RQ4 (Accuracy-Dynamic): GOALEXPLORER encounters 39 correct transitions in 21 apps that it fails to trigger dynamically (1.86 transitions per app that contain such transitions, and 1.08 transitions per app overall). The main causes for such failures (false positives of dynamic exploration) are:

Semantic Inputs: in 17 cases (11 apps, 1.54 transitions per app), meaningful inputs (other than login credentials) are required to explore the path to the target. For example, a valid zip code is required to start the search in the Mileage app and an mp3 file has to be selected for upload in AnyMemo. Neither GOALEXPLORER nor the contemporary dynamic exploration techniques can generate such semantic inputs.

Remote Triggers: in 13 cases (8 apps, 1.63 transitions per app), the transitions can only be triggered given a certain response from the remote server. For example, in SyncToPix, a target can only be triggered when the app receives a certain reply from the server, to syncing data. During our testing period, such reply was never received and none of the tools were able to trigger these transitions.

Event Order: in 9 cases (5 apps, 1.8 transitions per app), a transition is only possible under a certain order of events in a screen. For example, in the BookCatalogue app, the target can be reached only after the user marks certain electronic books as an anthology. While dynamic explorer is able to resolve many cases that require such interactions (17 cases got successfully resolved in our data set), the correct resolution is not always guaranteed.

Overall, the 39 transitions that GOALEXPLORER could not follow resulted in missing 12 targets in 9 apps; GOALEXPLORER finds alternative paths to targets for the remaining 27 transitions. Sapienz and Stoat cannot trigger 10 of the missing targets either.

> To answer RQ4: The main reasons GOALEXPLORER cannot trigger a valid transition are the lack of semantic inputs, e.g., files of a certain type, triggered that are external to the app, e.g., certain response from the server, and pre-defined order of input events that cannot be determined statically or dynamically.

Evaluation Summary: Overall, our experiments demonstrate that the combination of the static STG model and the run-time exploration techniques applied by GOALEXPLORER is effective for the goal-driven exploration task. By using these techniques, GOALEXPLORER is able to reach substantially more targets than techniques based on other static UI models. It is able to reach a comparable number of targets as the dynamic app exploration techniques, only substantially faster.

C. Limitations and Threats to Validity

The **external validity** of our results might be affected by the selection of subject apps that we used and our results may not necessarily generalize beyond our subjects. We attempted to mitigate this threat by using a "standardized" set of benchmark apps and by extending this set to include the 95 top Google Play apps. As we used most-popular real-world apps of considerable size, we believe our results will generalize for other apps. For **internal validity**, our static analysis relies on FlowDroid to construct the callgraph and collect the callbacks. The validity of our results thus directly depends on the accuracy of that tool.

The main **limitation** of our approach is the lack of knowledge of the correct combinations of UI events on a screen required to issue a transition. Also, even though we extended our implementation to handle login screens, GOALEXPLORER cannot handle screens that require other types of semantic inputs, such as a zip code or a specific type of file. This limitation is common to many other automated exploration approaches, and we intend to look at it as part of future work. Moreover, GOALEXPLORER inherits weaknesses of its underlying static analysis tools, such as handling of reflection, inter-component communication, and callback order resolution.

V. RELATED WORK

In this section, we discuss existing static Android UI models, automated app exploration techniques that rely on building UI models dynamically, and prior work on targeted app exploration.

Static Android UI Models. Approaches that statically construct an Android UI model, such as A3E [19] and Gator [29], are the closest to our work; the models they build – ATG and WTG – are extensively discussed in Sects. II and III-A and compared with our approach in Sect. IV. A number of approaches [46], [47], [48] extend these models, e.g., by adding information about specific API invocations and using that to estimate the execution time and energy consumption of possible app execution paths. Similar to GOALEXPLORER, StoryDroid [49] enhances WTG with information about fragments and then proposes an approach to statically render the UI of each activity. Yet, the resulting representation is similar to our WTG-E model, not the STG model used by GOALEXPLORER. That is, all these approaches do not change the underlying representation of UI elements, thus sharing all the discussed limitations of the ATG and WTG models.

Automated App Exploration. Instead of relying on a static model, some approaches [18], [20], [21], [22], [50], [23], [51], [26], [27], [52] build the model dynamically during app exploration. For example, Stoat [27] is based on a high-level, statically-constructed model that captures the core application activities and input events. The model is further refined during the dynamic analysis, adding missing states and events. Sapienz [26] seeks to maximize code coverage and minimize the length of the generated test sequences, to reveal more faults in shorter time. It uses pre-defined patterns of lower-level events that capture complicated interactions with the application in order to achieve higher coverage. Ape [52], which was published after our paper was submitted, represents the dynamic model as a decision tree and continuously tunes using the feedback obtained during testing. It constructs a finer-grained model than that of Sapienz and Stoat by, e.g., including additional widget attributes that indicate whether a widget is clickable or not. SwiftHand [50] focuses on minimizing app restarts during testing while still achieving maximal possible coverage.

Although the models used in these techniques are different, the underlying idea is similar: dynamically build a model of the app and mutate it so that the generated tests achieve higher coverage and fault detection. Since these tools focus on systematically executing the whole app to detect faults rather than swiftly navigating to a particular functionality of interest, they are inefficient for goal-driven exploration. That is because without building an accurate static model and identifying the target upfront, it is difficult to determine which part of the app to ignore/explore first.

Another line of work explores approaches for covering the full code and state space of an app [53], [54], [55]. Such approaches build symbolic execution engines that either model the Android framework or build a customized Dalvik virtual machine so that the app can be exercised in a controlled environment. They only work for a specific version of Android and have to be rebuilt for each new version. In contrast, our approach does not require such modifications.

Targeted App Exploration. Symbolic and concolic execution is also used for generating semantically meaningful inputs for directing app exploration towards a particular, usually malicious behavior. For example, Jensen et al. [56] use concolic execution to generate input event sequences that force the execution towards the target line of code. Yet, this approach only resolves the path constraints within one component and requires an UI model of the app to handle the transitions between components. Schütte et al. [57] also use concolic execution to force app executing into a security-sensitive API. Unlike GOALEXPLORER, this approach relies on modifying the app to “shortcut” the path to the target. Similar, SmartDroid [58] modifies the Android framework to prevent the creation of activities that do not lead to the desired sensitive target. Finally, Wong and Lie [59] and Rasthofer et al. [60] generate Android execution environments to direct the app towards a given target. Instead, our approach identifies a path to the target in the original, unmodified version of the app, which guarantees the existence and correctness of that path.

VI. CONCLUSIONS

This paper introduced GOALEXPLORER – a tool for goal-driven exploration of Android applications. Such a tool is useful for scenarios when an analyst needs to automatically trigger a functionality of interest in an app. The main contributions of GOALEXPLORER are (a) the static technique for constructing STG – a model that represents UI screens and transition between the screens, and (b) an engine that uses STG to build an executable script that triggers the functionality of interest. Our empirical evaluation shows that the STG model introduced in GOALEXPLORER is more accurate than other existing static UI models of Android applications and can reach the target substantially faster than the state-of-the-art dynamic approaches.

Acknowledgments. We would like to thank Junbin Zhang for his insightful comments on this work and, in particular, for helping us to manually establish the ground truth of reachable app target. We also thank the anonymous reviewers for their insightful comments that helped us further improve the work.

REFERENCES

[1] O. Hou. A Look at Google Bouncer. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/

[2] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective Real-time Android Application Auditing," in *Proc. of SP*, 2015, pp. 899–914.

[3] IBM. Identify and Remediate Application Security Vulnerabilities with IBM Application Security. [Online]. Available: https://www.ibm.com/security/application-security/appscan

[4] Tencent. Tencent Kingkong App Scan. [Online]. Available: https://service.security.tencent.com/kingkong

[5] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond, "Detecting Display Energy Hotspots in Android Apps," in *Proc. of ICST*, 2015, pp. 1–10.

[6] D. Ferreira, V. Kostakos, A. R. Beresford, J. Lindqvist, and A. K. Dey, "Securacy: An Empirical Investigation of Android Applications' Network Usage, Privacy and Security," in *Proc. of WiSec*, 2015, pp. 11:1–11:11.

[7] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson, "Haystack: In Situ Mobile Traffic Analysis in User Space," *CoRR*, 2015.

[8] Y. Liu, H. H. Song, I. Bermudez, A. Mislove, M. Baldi, and A. Tongaonkar, "Identifying Personal Information in Internet Traffic," in *Proc. of COSN*, 2015, pp. 59–70.

[9] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic," in *Proc. of MobiSys*, 2016, pp. 361–374.

[10] M. Henze, J. Pennekamp, D. Hellmanns, E. Mühmer, J. H. Ziegeldorf, A. Drichel, and K. Wehrle, "CloudAnalyzer: Uncovering the Cloud Usage of Mobile Apps," in *Proc. of the MobiQuitous*, 2017, pp. 262–271.

[11] Zedge, Inc. ZEDGE. [Online]. Available: https://play.google.com/store/apps/details?id=net.zedge.android

[12] Facebook. Facebook SDK for Android. [Online]. Available: https://developers.facebook.com/docs/android/

[13] ——. Facebook Login API. [Online]. Available: https://developers.facebook.com/docs/facebook-login/android/

[14] Privacy International. How Apps on Android Share Data with Facebook. [Online]. Available: https://privacyinternational.org/report/2647/how-apps-android-share-data-facebook-report

[15] Google. Espresso. [Online]. Available: https://developer.android.com/training/testing/espresso/

[16] ——. UI Automator. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[17] RobotiumTech. Robotium. [Online]. Available: http://www.robotium.org

[18] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps," in *Proc. of MobiSys*, 2014, pp. 204–217.

[19] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *Proc. of OOPSLA*, 2013, pp. 641–660.

[20] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *Proc. of ASE*, 2012, pp. 258–261.

[21] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proc. of OOPSLA*, 2013, pp. 623–640.

[22] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *Proc. of ESEC/FSE*, 2013, pp. 224–234.

[23] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *Proc. of FSE*, 2014, pp. 599–609.

[24] K. Moran, L.-V. Mario, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *Proc. of ICST*, 2016, pp. 33–44.

[25] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor," in *Proc. of EuroSys*, 2014, pp. 18:1–18:15.

[26] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in *Proc. of ISSTA*, 2016, pp. 94–105.

[27] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, Stochastic Model-based GUI Testing of Android Apps," in *Proc. of FSE*, 2017, pp. 245–256.

[28] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated Testing of Android Apps: A Systematic Literature Review," *IEEE Transactions on Reliability*, pp. 1–22, 2018.

[29] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static Window Transition Graphs for Android," in *Proc. of ASE*, 2015, pp. 658–668.

[30] T. Bray. Fragments For All. [Online]. Available: https://android-developers.googleblog.com/2011/03/fragments-for-all.html

[31] An Open Protocol to Allow Secure Authorization in a Simple and Standard Method From Web, Mobile and Desktop Applications. [Online]. Available: https://oauth.net/

[32] The OAuth 2.0 Authorization Framework: Bearer Token Usage. [Online]. Available: https://tools.ietf.org/id/draft-ietf-oauth-v2-bearer-23.xml

[33] D. Lai and J. Rubin. (2019) Onlile Appendix. [Online]. Available: https://resess.github.io/PaperAppendices/GoalExplorer/

[34] Google. App Components. [Online]. Available: https://developer.android.com/guide/components/fundamentals

[35] ——. Activity-lifecycle Concepts. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle

[36] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proc. of PLDI*, 2014, pp. 259–269.

[37] Google. Webview. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView

[38] AXML Printer. [Online]. Available: https://github.com/rednaga/axmlprinter

[39] Google. Fragment. [Online]. Available: https://developer.android.com/reference/android/app/Fragment

[40] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting Inter-component Privacy Leaks in Android Apps," in *Proc. of ICSE*, 2015, pp. 280–291.

[41] Android Debug Bridge (adb). [Online]. Available: https://developer.android.com/studio/command-line/adb

[42] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps," in *Proc. of USENIX Security*, 2015, pp. 977–992.

[43] F-Droid Limited. F-Droid. [Online]. Available: https://f-droid.org/en/

[44] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *Proc. of ASE*, 2015, pp. 429–440.

[45] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis," in *Proc. of ICSE*, 2015, pp. 77–88.

[46] Y. Wang and A. Rountev, "Profiling the Responsiveness of Android Applications via Automated Resource Amplification," in *Proc. of MOBILESoft*, 2016, pp. 48–58.

[47] H. Wu, S. Yang, and A. Rountev, "Static Detection of Energy Defect Patterns in Android Applications," in *Proc. of CC'2016*, 2016, pp. 185–195.

[48] Y. Zhang, Y. Sui, and J. Xue, "Launch-mode-aware Context-sensitive Activity Transition Analysis," in *Proc. of ICSE*, 2018, pp. 598–608.

[49] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "StoryDroid: Automated Generation of Storyboard for Android Apps," in *Proc. of ICSE*, 2019, pp. 596–607.

[50] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proc. of OOPSLA*, 2013, pp. 623–640.

[51] Y.-M. Baek and D.-H. Bae, "Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria," in *Proc. of ASE*, 2016, pp. 238–249.

[52] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI Testing of Android Applications via Model Abstraction and Refinement," in *Proc. of ICSE*, 2019, pp. 269–280.

[53] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android Apps Through Symbolic Execution," *SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[54] R. Johnson and A. Stavrou, "Forced-Path Execution for Android Applications on x86 Platforms," in *Proc. of SERE-C*, 2013, pp. 188–197.

[55] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated System Input Generation for Android Applications," in *Proc. of ISSRE*, 2015, pp. 461–471.

[56] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated Testing with Targeted Event Sequence Generation," in *Proc. of ISSTA*, 2013, pp. 67–77.

[57] J. Schütte, R. Fedler, and D. Titze, "ConDroid: Targeted Dynamic Analysis of Android Applications," in *Proc. of AINA*, 2015, pp. 571–578.

[58] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proc. of SPSM*, 2012, pp. 93–104.

[59] M. Wong and D. Lie, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware," in *Proc. of NDSS*, 2016, pp. 21–24.

[60] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments," in *Proc. of ICSE*, 2017, pp. 300–311.