Graph-Aided Directed Testing of Android Applications for Checking Runtime Privacy Behaviours

Joseph Chan Joo Keng, Lingxiao Jiang, Tan Kiat Wee, and Rajesh Krishna Balan
School of Information Systems, Singapore Management University
joseph.chan.2012@phdis.smu.edu.sg, {lxjiang, williamtan, rajesh}@smu.edu.sg

ABSTRACT

While automated testing of mobile applications is very useful for checking run-time behaviours and specifications, its capability in discovering issues in apps is often limited in practice due to long testing time. A common practice is to randomly and exhaustively explore the whole app test space, which takes a lot of time and resource to achieve good coverage and reach targeted parts of the apps.

In this paper, we present MAMBA[1], a directed testing system for checking privacy in Android apps. MAMBA performs path searches of user events in control-flow graphs of callbacks generated from static analysis of app bytecode. Based on the paths found, it builds test cases comprised of user events that can trigger the executions of the apps and quickly direct the apps' activity transitions from the starting activity towards target activities of interest, revealing potential accesses to privacy-sensitive data in the apps. MAMBA's backend testing engine then simulates the executions of the apps following the generated test cases to check actual runtime behavior of the apps that may leak users' private data. We evaluated MAMBA against another automated testing approach that exhaustively searches for target activities in 24 apps, and found that our graph-aided directed testing achieves the same coverage of target activities 6.1 times faster on average, including the time required for bytecode analysis and test case generation. By instrumenting privacy access/leak detectors during testing, we were able to verify from test logs that almost half of target activities accessed user privacy data, and 26.7% of target activities leaked privacy data to the network.

CCS Concepts

•**Security and privacy** → *Privacy protections;* •**Software and its engineering** → *Automated static analysis; Dynamic analysis;*

Keywords

Automated Mobile Application Testing, Mobile Privacy

[1] We name our system after the MAMBA, a fast-moving terrestrial snake, in reference to similarities with the multiple path chains our directed tester takes.

AST'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4151-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896921.2896930

1. INTRODUCTION

There has been significant interest in detecting and preventing privacy violations in mobile applications. Presently, various leak detectors exist in the research community that can flag privacy data accesses [5, 6], block the sending of private data [4, 9] or leverage crowdsourcing [1, 21] to provide guidance to users based on the opinions of the crowd. Mobile platforms such as Apple iOS and Android have also made privacy managers default tools on their systems [8].

While these tools have various degrees of effectiveness, they possess some disadvantages such as CPU overheads on the users' devices (up to 35% over baseline [26]), improper behaviours in up to 40% of applications due to data starvation [9], as well as requiring customized phone images or operating system modifications.

To avoid these disadvantages, detecting and preventing privacy violation may be done separately by some trusted parties (e.g., the app stores) before delivering the apps to users. Automated testing of apps is a commonly used approach for such a purpose. Various automated testing tools have been built for Android applications [2, 3, 7, 11, 14]. Without knowledge about potential locations of issues in the apps, the testing tools are often designed to cover as many portions of the apps as possible, including the application code, runtime states, activities and windows, GUI widgets, etc., and they need to ensure that certain hard-to-reach activities, states, or conditions are reached as well. In order to ensure high coverage for an app, a common practice is to produce more test cases and increase testing resources (e.g., machines and/or testing instances), trying to comprehensively test the app to completion [20]. Lee et al. [11] found that app testing could take anywhere from a few minutes to more than 10 hours to complete per application. Thus, limited resources can significantly hinder the effectiveness of the automated testing tools.

Scaling up automated app testing and improving its effectiveness within limited resources are a challenging problem. A scalable and effective method for automated app testing can really help app stores to protect users' privacy, and can also have benefits in other areas such as detection of bugs, security vulnerabilities, malware [12], checking application design and runtime behaviours with respect to specifications [11], and analysis of possible causes of privacy leaks [10].

To address the challenge, we built the MAMBA system for allowing fast and directed testing of Android applications towards privacy violation detection, based on analyzing control-flow graphs of Android callbacks obtained from static analysis of application bytecode. The control-flow graphs of callbacks are used to build test cases for Android application window transitions, consisting of user actions on clickable application GUI views and widgets. The test cases are then provided to MAMBA's automated testing backend, which stimulates user actions to trigger the execution of the app and direct its transition from its starting activity to the target activities of interest, which in this paper are the ones that call Android APIs to access users' private data, and we use *privacy-sensitive API calls* to refer to such calls. The main contributions of the paper are as follows:

- Present techniques to utilize the control-flow graphs of Android callbacks for test case generation;
- Tailor test case generation to reveal privacy-sensitive API calls in apps;
- Direct testing of apps to follow test cases that may potentially reveal privacy-sensitive API calls;
- Evaluate our graph-aided directed testing and compare its performance with another automated testing approach based on exhaustive search, and verify actual privacy leaks revealed through our approach.

2. GRAPH-AIDED DIRECTED TESTING

In this section, we describe how the MAMBA testing system works (Please refer to Figure. 1 for the system diagram). An Android app's binary .apk file is decompiled into its intermediate representations (smali/bytecode) of the virtual-machine code. To determine which are the target activities of interest, the system first analyzes which activities in the app are linked to privacy sensitive API calls in a call-chain (Privacy Specifications). Next, a control-flow graph of call-backs (CCFG) is generated from the bytecode using the GATOR/SOOT tool [27]. The CCFG is then analyzed and traversed to obtain test-cases of actionable user-actions that can guide automated testing from the root activity to any of the other activities within the app (*Automated Test Walks*). These test cases are then provided to an automated front-end tester, where only the test-cases leading to target activities of interest are executed to reduce testing time. The run-time privacy behaviours of the apps are logged and verified using leak/privacy-data reports generated from privacy leak detectors TaintDroid and PMP [6, 19].

The proceeding sub-sections details the implementation and techniques utilized in the directed testing system.

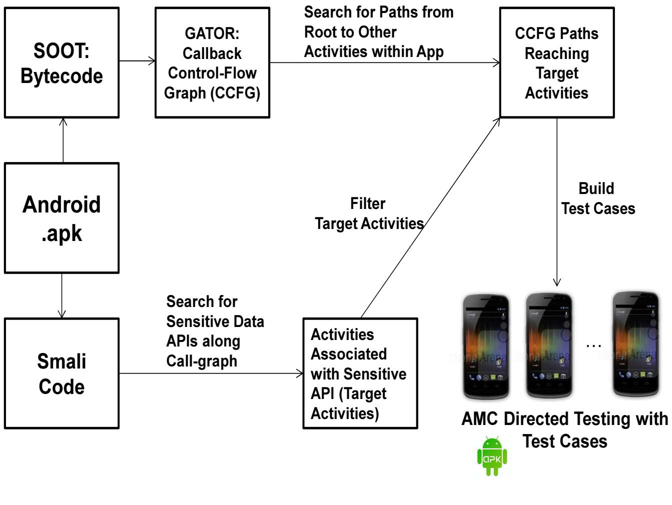
SOOT: Bytecode
GATOR: Callback Control-Flow Graph (CCFG)
Search for Paths from Root to Other Activities within App
CCFG Paths Reaching Target Activities
Android .apk
Filter Target Activities
Build Test Cases
Search for Sensitive Data APIs along Call-graph
Smali Code
Activities Associated with Sensitive API (Target Activities)
...
AMC Directed Testing with Test Cases
APK

Figure 1: System Diagram of MAMBA: Directed Testing of Android Applications for Checking Privacy Specifications

2.1 Privacy Sensitive API Call Association with App Activities

In order to determine where to direct the activity traversal and testing towards, our tool performs a process of finding links from sensitive API calls to available activities within the app. We obtained the API method and class names under the Android API documentation [15] which are involved in the access of users' privacy data. Such data accesses included, For e.g. GPS Location ('getLatitude'/'getLongtitude' etc.), MAC Address ('getMacAddress'), Contacts List ('ContactsContract') etc. We utilized analysis over Smali Code, an intermediate representation of the DEX format decomplied from Android apk files using APKTOOL [25].

Utilizing an analysis tool that we built using Java, we first checked for developer written methods and classes that contained any of the privacy sensitive Android API calls in its code body (Privacy Specifications). Our tool then performed method call-chain linking to build a control-flow from those initial methods with sensitive API calls. Along the call-chains, we identified whether any of the classes along the chain belonged to an Android activity class. If it did, we recorded the activity as a *target activity* in which automated directed testing should reach in Section 2.2.2.

2.2 Automated Testing of Mobile Applications

Automated testing involves exploring the app by effecting activity transitions and states, which is performed by stimulating user click events on clickable mobile buttons/widgets. Achieving good app coverage, reachability and efficiency are the main important goals in automated testing. Testing has to ensure that all the available activities within the app are explored, and that certain hard-to-reach states of activity windows are reached. It also has to do so in a practical amount of time. A random fuzzing strategy (for example Android Monkey [16]), might be able to provide sufficient coverage and reachability for checking privacy specifications, but only with a length of testing time that might not be practical.

We built our automated tester front-end over the Automated Model Checker (AMC) [11]. AMC determines the state of the current activity or window that an app is currently in by computing a hash over the application's DOM tree obtained from the Hierarchy Viewer tool [17]. It then lists and selects one of the available clickable UI component within the activity to send click-actions to using the Android Monkey. After sending click-action, it determines if the resulting newly arrived state is equivalent to the previous state by equivalence heuristics. It keeps the hashes of all visited states to track exploration, and ends testing when all states have been explored.

2.2.1 *Problems with Testing of Applications*

Like other automated testers, AMC in its basic form is still impractical for scaling up testing of privacy specifications. Based on our experiences with testing for over a thousand apps, testing times to reach completion of all activities can take up to 4-5 hours for a single app. Model Checking suffers from the well known problem of 'State-Explosion' [24], where some tests fails to terminate altogether due to the huge number of states. While AMC implements heuristics in computing layout structure hashes instead of content hashes, state-explosion problems could still invariably occur and cause testing to fail to terminate. In addition, AMC requires a stabilization period of waiting time for each app state to stabilize (typically 10-20 seconds) before computing the state hash, and this contributes to prolonged testing times.

To help the problems that occur, utilizing the outputs of a call-graph to build a transition model would greatly help in guiding AMC and improve testing efficiency. The transition model can prevent AMC from re-testing states that it has already visited, and guide AMC on known paths to target activities. It also removes the need for a stabilization wait time, and greatly reduces testing time required.

2.2.2 *Automated Walking Tool*

We built an automated walking tool on top of AMC, which utilizes a pre-computed transition model as inputs to guide its testing. The transition model consists of sets of test-cases (separate and distinct paths) for each activity that exist in the app. Each test-case is made up of the user-actions that are required to be sent to the UI to cause activity transitions, which are executed in sequence one after another. The testing thus proceeds by 'walking' the application from the initial root activity to a target activity obtained in Section 2.1, before terminating and restarting from the root activity again for the next test-case. Testing now proceeds in a more deterministic and guided fashion, which improves coverage and efficiency.

2.2.3 *Test Monitors & Event Recorders*

Automated testing was carried out on hardware Android devices and Emulators. We implemented Event Recorders for test instances, that automatically logged and saved the traces of user-interaction with the UI-components with the test apps. Leak reports from 2 privacy leak detectors, TaintDroid [6] and ProtectMyPrivacy (PMP) [19] were also recorded. These logs allowed us to mine privacy leak-causing views and widgets/buttons, as well as determine privacy leak characteristics using Frequent Item-set Mining [10].

We utilized parallelization with Android devices and emulators and scaled up emulator testing over multiple test instances on the Amazon EC2 cloud computing platforms. Problems with testing speed and stability on the Android Emulator have previously been highlighted [7], with some works specifically avoiding to run testing on the Emulator due to these issues. To ensure smooth testing and deal with instabilities, we implemented Test Monitors and fault handling to coordinate multiple test instances and perform process monitoring.

2.3 Generating Test-Cases from Callback Control-Flow Graphs (CCFG)

In order to generate test cases for automated transitions from root to target activities, we require an appropriate static representation of Android apps. Activity/state changes on the Android app are mostly enacted by event-driven *callbacks* [18], which are system calls centrally coordinated by the Android platform. Callbacks are defined for various user-driven and system defined events such as user-clicks/touches, UI component creation/destruction, event listeners, hardware/sensor state changes etc. Due to this event-drive framework of Android's, a normal control and data-flow analysis would not work directly for our purposes.

We thus leveraged on the user-driven callback static control-flow analysis framework in [27], which also includes the modelling of Android GUI-related objects and their flow through the application [22]. In this framework, Android program behaviour is modelled using a representation referred to as a Callback Control-Flow Graph (CCFG). The CCFG representation contains only interprocedural edges to and from callback functions, where a node represents the execution of a GUI event-handler or component lifecycle callback, and an edge represents the trigger sequence of callbacks. In addition, artificial helper nodes (branch & join nodes) exist in the CCFG to aid better in the representation of execution sequences of the graph.

To form our activity transition model, we perform an abstraction of the CCFG; In which the graph nodes in our model are activities and graph edges are the widget identifiers of the GUI components that will cause transitions between 2 activities. We utilized widget identifiers for edges because Android's Hierarchy Viewer [17] provides visibility on clickable Android UI components from these identifiers. By traversing the CCFG, we obtain all feasible paths from root to target activities of interest and build test cases for inputs to the automated tester.

2.3.1 *Traversal and Path Search of CCFG*

Using the GATOR/SOOT tool [27], we generate the CCFG from decompiled apps' byte-code. In Android, all activities within the app has to be declared within the app's manifest resource. We obtain a list of all activity names that the app contains. From the CCFG, we identify all nodes that represent the creation (onCreate event-handler) of these activities. Using these nodes as starting points, we traverse backwards on the CCFG using a Depth-First Search (DFS) algorithm until the root (Main) activity node is reached. We thus recorded a resulting list of paths of all activities within the app to the root (Main) activity. We then utilized these paths to build test-cases for the automated tester (Section 2.3.2).

Multiple Paths & Path Feasibility: Our analysis encountered many cases whereby there were multiple paths from an activity back to the root activity. While our tool logged all possible paths, it does not identify which paths are actually feasible or can be reached during run-time. Infeasible paths might occur if some condition is required before the next resulting activity can be started. For example, an activity might only be accessible if the user is on WiFi. Another example could be in a shopping app, in which a checkout activity might only be accessible if the user had checked on certain items in a list.

To increase accessibility and improve testing time, we ordered multiple paths for each activity according to length of shortest to longest, and executed shortest paths first before trying out longer paths if the shorter paths were found to be infeasible. Our automated tester executes shorter paths before moving to try out the longer paths if it is unable to reach the resulting activity. In the future, it might be possible to consider modelling of the conditions under which infeasible paths occur, so that automated testing can identify and enable certain hard to reach conditions for accessibility of app activities.

Search Time of CCFG: Observations on testing on a small set of 50 apps indicated that times required ranged from 10 seconds to over 3 minutes on a quad-core desktop machine running Linux, depending on the size and branch structure of the CCFG. This indicated that the approach was reasonably scalable. Given that we only utilized a simple DFS, there are possibilities to further optimize the search time by considering other searches (e.g. Tabu search). It would also be possible to consider search strategies that consider the branching structure of the CCFG and adaptively modify the search. We leave such algorithmic designs of path search strategies for future work.

2.3.2 *Test Case Generation*

Based on the paths found from an app's root activity to all other activities contained within the app, we generate test cases of user-events on UI components for automated directed testing. As mentioned in Section 2.3, nodes on the CCFG represent callback functions of GUI event-handlers (e.g. onClick(), onTouch()) or component lifecycles (e.g. onCreate(), onDestroy()). Figure. 2 illustrates a node representing an 'onCreate' callback of an activity with name 'com.bitsmedia.android.quitpro.activities.PremiumActivity'. This activity is linked by an edge to an 'onClick' callback function of an app's button UI component, which exists on a separate activity called 'com.bitsmedia.android.quitpro.activities.InfoActivity'. The widget identifier of the button UI component ('premiumUpgradeButton') is visible within the 'onClick' callback in the edge.

Node (Callback of Activity):

[START] Node ⟨ Activity[com.bitsmedia.android.quitpro.activities.PremiumActivity], ⟨ com.bitsmedia.android.quitpro.activities.PremiumActivity: void **onCreate**(android.os.Bundle) ⟩ ,implicit_lifecycle_event,Activity[**com.bitsmedia.android.quitpro.activities.PremiumActivity**] ⟩

Edge (Callback of Button -> Callback of Activity):

Node ⟨ Inflate_Node[android.widget.Button,WID[**premiumUpgradeButton**]], ⟨ com.bitsmedia.android.quitpro.activities.InfoActivity$1: void **onClick** (android.view.View) ⟩ , click, ACT[**com.bitsmedia.android.quitpro.activities.InfoActivity**] ⟩
->

[START] Node ⟨ Activity[com.bitsmedia.android.quitpro.activities.PremiumActivity], ⟨ com.bitsmedia.android.quitpro.activities.PremiumActivity: void **onCreate** (android.os.Bundle) ⟩ ,implicit_lifecycle_event,Activity[**com.bitsmedia.android.quitpro.activities.PremiumActivity**] ⟨ [normal, start_activity, [window_must_start]]

Figure 2: Node & Edge Representations in CCFG

Activity: com.bitsmedia.android.quitpro.activities.PremiumActivity
Test Case #1 : [action_premium]
Test Case #2 : [action_info, premiumUpgrabdeButton]

Activity: com.interestingcoolerfreegoimbh.app.koisettings
Test Case #1 : [settings, button1unlock]
Test Case #2 : [settings, RelativeLayout04, gridview]
Test Case #3 : [settings, RelativeLayout05, gridview2]
Test Case #4 : [settings, RelativeLayout03, gridviewback]

Figure 3: Example Sets of Test Cases Built For Automated Tester

On tracking along the discovered paths, we look out for edges which indicate that an activity creation callback was due to a prior GUI-component event-handling callback (Figure. 2 illustrates such an edge). We obtained all the widget identifiers of the GUI-components of these edges. To facilitate Directed Testing, the widget identifiers are then relatable to DOM layout information provided by Android's Hierarchy Viewer, whereby determinations can be made on screen coordinates in which to send clicks to using Android Monkey.

Figure 3 illustrates examples of resulting test-cases for 2 different activities: 2 different paths were found for '.PremiumActivity', the 1st shorter path being a single user-action on the widget with identifier 'action_premium', and the 2nd longer path involving transition with 2 user-actions ('action_info' and 'premiumUpgradeButton'). The other example for the activity '.koisettings' found 4 different possible paths from root to the activity, the 1st path requiring 2 user-actions, and 2nd-4th paths requiring 3 user-actions. As mentioned in Section 2.3.1, not all of the paths might be feasible. As a heuristic to save testing time, we indicated to automated testers to test shorter paths first before moving to longer paths if initial transition attempts were unsuccessful.

3. EXPERIMENTAL EVALUATION

In this section, we present evaluation results of MAMBA on a set of 24 Android apps crawled from the Google Play Store. A comparison with Exhaustive Testing was also perform for the time required to reach and verify activities that involve the use of privacy sensitive APIs (Target Activities).

3.1 Path Search & Test-Case Generation

From Table 1, the Privacy Sensitive API associator uncovered that 21 apps accessed users' GPS data, 2 of the apps accessed the Contact List, 3 apps Device Identifiers, and 1 app each accessed users' Microphone and SMSes. 1 app (Arsenal Fan Club) did not access any privacy data. The total no. of activities each app contained as well as the no. of activities found to be accessing user privacy data are illustrated in the table.

3.1.1 Target Activities of Interest

The apps were found to each contain between 2 to 18 activities in total (Table 1). 10 of the 24 apps had 100% of their activities accessing privacy data, with varying numbers (0-100%) for the rest. These privacy accessing activities were flagged out to be reached by the directed tester. During automated testing, leak reports were obtained from leak detectors, as mentioned in Section 2.2.3. Mining of the test logs allowed us to infer data usage characteristics [10].

3.1.2 Generating Test-Cases for Directed Test

The processing times required for generating test cases for target activities of each app are tabulated in Table 1 (seconds). Test Case Generation time includes bytecode and CCFG generation, CCFG traversal and search for paths as well, Privacy Sensitive API association as the generation of user-action sequences for the automated tester. Test cases were generated on an AMD Opteron 4386, 3.1GHz CPU and running 64-bit Debian Linux.

Test Case Generation took between 26 to 94 seconds for each app (Under column 'Graph-Directed Search- Test Case Generation (s)'). The results indicated that generation times required were not dependent on the total number of activities in the app or the number of target activities, but were dependent on app size and complexity. An example was the No. 12 app: 'Fat Burning Foods', which had 15 activities in total and 15 target activities, but required 42 seconds for processing. This is in comparison to No. 2 app: 'A+ Certification Lite' that required a longer generation time of 45 seconds but had smaller number of only 4 activities and 1 target activity.

3.1.3 Automated App Testing

The testing times required for running the resulting test cases on an automated tester are tabulated under the column 'Graph-Directed Search-Test Case Running (s)'. Automated testing were performed on 3 x hardware Galaxy Nexus devices running Android OS 4.1.1, and connected to a testing program running on a Windows 7 machine. Testing can be carried out on emulators as well, but we only utilized hardware devices for the sake of consistency in this evaluation.

Running a single test case took about over a minute on average. Because this step involved automated activity transitions on the app, these timings required were significantly larger compared to the times required for test case generation. Automated app testing took between 69 seconds (1 minute 9 seconds) to 900 seconds (15 minutes) for each app, and were dependent on the number of target activities as well as the number of test cases.

3.1.4 Graph-Aided Directed Testing Times

As mentioned in Section 2.2.2, graph-aided directed testing involved automated activity transition of apps from the root activity to each of its target activities. The total testing time required for graph-aided directed testing is a combination of the time required for static test case generation as well as the automated running of the test cases on the automated tester. Total times required for Directed Testing times ranged from 2 mins 4 sec to 15 mins 40 sec (Column: 'Graph-Directed Testing-Total [hh:mm:ss]').

						Exhaustive Testing (Testing Time)	Graph-Directed Testing (Testing Time)		
No.	App Name	Package	Privacy Data	Total No. of Activities	No. of Target Activities	Total [hh:mm:ss]	Test Case Generation [s]	Test Case Running [s]	Total [hh:mm:ss]
1	Album Cover Finder	com.ftpcafe.coverart.trial	GPS	8	6	1:18:25	59	337	0:06:36
2	A+ Certification Lite	com.mhazzm.APlusCertification702Lite	GPS	4	1	0:03:10	45	101	0:02:26
3	Advanced Task Killer	com.rechild.advancedtaskkiller	GPS	6	1	1:04:11	47	140	0:03:07
4	African American Quotes	com.hmobile.africanamericanquote	GPS	11	11	1:18:18	37	610	0:10:47
5	Android Book App	com.appmk.book.main	GPS	5	3	0:19:41	56	146	0:03:22
6	Arsenal Fan Club	com.arsenalfanclub	Nil	9	0	0:46:55	57	N.A.	Nil
7	Blood Alcohol Tracker	com.promille	GPS	7	7	0:27:00	32	411	0:07:23
8	Call and SMS Easy Blocker	com.ekaisar.android.eb	GPS, Contact-List ,SMSes	9	2	0:17:17	94	168	0:04:22
9	Car Performance Free	com.unnull.apps.carperformancefree	GPS	7	2	0:12:51	26	224	0:04:10
10	CLT vs. PJ	com.neocode.cltxpj	GPS	3	3	0:03:35	30	185	0:03:35
11	Driving Skill Monitor	com.drismo	GPS	14	3	0:34:09	50	216	0:04:26
12	Fat Burning Foods	com.v1_4.fatburningfoods.com	GPS, IMEI	15	15	0:11:27	42	535	0:09:37
13	Font for Galaxy	com.hongik.fontomizerSP	GPS	2	2	0:03:09	38	114	0:02:32
14	Interesting Cooler	com.interestingcoolerfreegoimbh.app	GPS	10	1	0:17:56	55	69	0:02:04
15	Interpret Your Dream	com.dreamforth.iyd	GPS	6	6	0:06:01	34	296	0:05:30
16	JaquecApp	com.terranology.jaquecapp	GPS	18	1	0:37:35	49	109	0:02:38
17	My Display Check	com.jensu.screenchecker	GPS	7	7	0:15:40	30	357	0:06:27
18	Shikoku Railway	com.appspot.noritsubushi.shikoku	GPS	6	6	0:20:49	38	422	0:07:40
19	Solar Battery Charger	com.solar.charger.battery	GPS	3	2	0:07:12	39	123	0:04:42
20	Space War APK	com.space_war_free_10	GPS, IMEI Phone No.	13	10	0:19:26	40	900	0:15:40
21	Speak Mandarin Free	com.chineseskill.lan_tool.sc	Microphone, Device IDs	3	2	3:23:10	61	148	0:03:29
22	Super Runner Boy	com.pack.Super-RunnerBoyTrial	GPS	4	1	0:02:32	68	80	0:02:28
23	TooLate Lateness	afw.allforweb.toolate	Contacts	6	1	0:08:34	42	95	0:02:17
24	Trios Labs Reader	com.triosLabs.hadithreader	GPS	8	8	0:26:47	47	313	0:06:00

Table 1: Results of Automated Testing - Exhaustive vs. Graph-Directed Test (MAMBA)

3.2 Comparison With Exhaustive Testing

We carried out automated testing on the same set of apps, but with Exhaustive Testing as the approach instead using Automated Model Checker (AMC). In Exhaustive Testing, no target activities were specified, but available user-actions for each clickable widget/button on each activity encountered were ordered sequentially in a top-to-bottom fashion based on the layout and each tried in-turn. On reaching a next activity state, the app was backtracked to the previous activity before selection of another clickable widget/button for tapping. In this fashion, the tester tries to reach all activities in the app. The total time required for exhaustively testing each of the apps are displayed in Table 1 under 'Exhaustive Testing-Total [hh:mm:ss]'.

The total testing times required for Exhaustive testing ranged from over 2 minutes (0:02:32) to more than 3 hours (3:23:10). This is comparison to total testing times required for graph-directed testing ('Graph-Directed Testing-Total [hh:mm:ss]'), which ranged from 2 minutes (0:02:04) to just over 15 minutes (0:15:40). These testing total times required for graph-directed testing were therefore significantly shorter than that required for Exhaustive Testing. Graph-directed testing was 6.1 times faster across all the 24 apps (Average 0:31:15 compared to 0:05:03), and was up to 58 times faster (No. 21 app: 'Speak Mandarin Free').

Why is Graph-Directed Testing Faster?: Testing was faster due to the immediate and expedited transitions to target activities via app buttons that were already known beforehand from test-cases built from CCFGs. This was in contrast to Exhaustive Tests, which had to spend time exploring all clickable UI components in each app activity, many of which might cause state changes but not activity transitions. Examples of such occurrences are Apps #1, #4 and #21, which required over 1-3+ hours respectively.

The branching structure of activity transitions also played a part to increase test times in Exhaustive Tests. From inspection, apps with more highly cascaded activity transitions required more time in Exhaustive Testing, due to the time required for back-tracking to

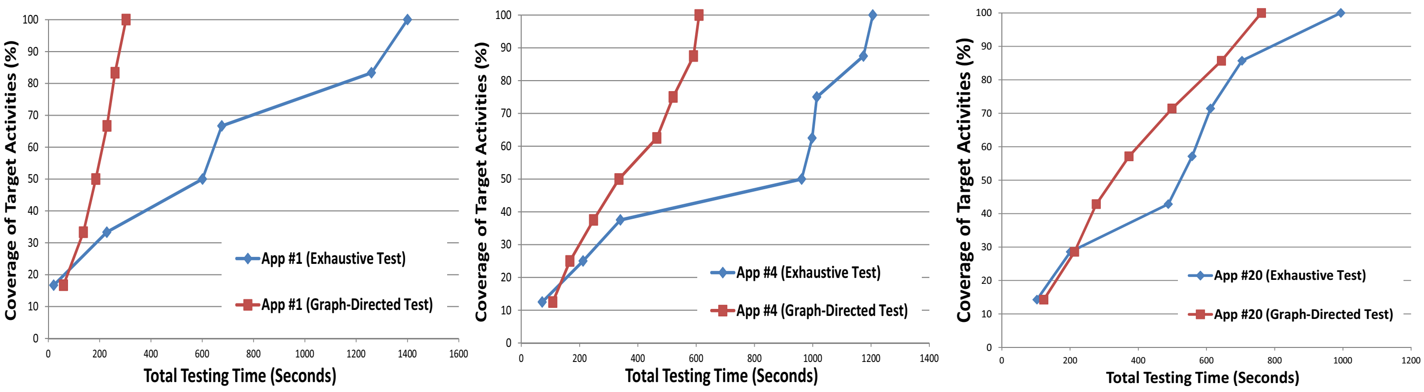

Figure 4: Coverage of Target Activities (%) over Total Testing Time (Seconds) for 3 Example Apps - (Left- App #1: "Album Cover Finder"; Middle- App #4: "African American Quotes" ; Right- App #20: "Space War APK")

previous activities, which graph-directed testing did not require.

Figure 4 displays the Coverage of Target Activities (%) over the Total Testing Time (seconds) for 3 example apps (App #1, #4 and #20). Although Exhaustive Testing was also able to cover target activities, it required significantly higher testing times compared to graph-directed testing. It can be seen that the Coverage of Target Activities in graph-directed testing converges faster compared to Exhaustive Testing. Coverage of activities in directed testing proceeded at a steady pace until completion, compared to the prolonged times required and stagnation at certain activity points for Exhaustive Testing.

3.2.1 Coverage of Target Activities

Both graph-directed and Exhaustive Testing were each able to cover 86 of the 101 target activities in Table 1 (85.1% coverage). From inspection, the remaining activities were inaccessible by our testers due to a few reasons. The first was that there were some defunct activities which were defined by developers and in app code, but did not contain start intents. There were also a few which required some hard to fulfil conditions that normal user-events could not meet. For example, App #11: Driving Skill Monitor had an activity that required input from GPS sensors for a certain distance before it could be accessed. App #20: Space War APK had 3 activities that required completion of game levels before it could be accessed. Allowing coverage for such activities falls outside the scope of our paper.

3.3 Verification of Privacy Specifications

As described in Section 2.2.3, leak reports from 2 privacy detectors, PMP and TaintDroid were logged during the automated test runs. PMP detected run-time privacy data accesses, while TaintDroid was able to detect privacy leaks over the network. Based on the logs captured from graph-directed testing, we verified that 45 of the 101 target activities (44.6%, over 13 apps) had privacy data accesses, and 27 of these activities (26.7%, over 8 apps) leaked private data to the network. Such privacy behaviours can be recorded in off-line databases for future notification to users [10]. The logs from Exhaustive Testing also indicated the same findings, which showed that MAMBA was able to perform verification of privacy specifications in apps just as well as exhaustively testing the app, but in a significantly reduced time.

4. DISCUSSION

Our evaluation demonstrated that graph-aided directed testing has significant advantages compared to Exhaustive Testing, mainly in much shorter testing times required to achieve coverage of target activities. On average across the test apps, Directed Testing was 6.1 times faster than Exhaustive Testing. This demonstrates the impact of building test-cases of activity transition models from CCFGs, prior to automated testing.

We observed that there were situations where graph-aided testing was only slightly faster than Exhaustive Testing. (For example for App No. 15: ’Interpret Your Dream’ and App No. 22: ’Super Runner Boy’.) From manual inspection of the apps, these arose in circumstances where activity transitions were only branched around the root (main) activity, such that the automated tester only needed to explore all buttons on the root activity page to transit to other activities within the app.

We highlight that the CCFGs generated from GATOR tool only contains explicit activity starting (’startActivity’) intents and do not include implicit intents. Explicit intents consists of activity type classes that have been defined within developer code of the app, while implicit intents consists of transitions to external window states (e.g. Transit from app to a web browser or email client).

5. FUTURE WORK

In the future, graph-aided directed testing can be applied to dynamically verify other types of run-time behavioural specifications. Examples of such specifications are software bugs, power testing & energy profiling of phone sensor behaviours and performance. The specifications might first be identified via static analysis, followed by fast and directed app activity transitions to reach the specific app activities.

There are also possibilities for improving strategies in app exploration, such as for example, analysis of an app’s transitional branching structure in the CCFG prior to automated testing. Currently, every directed test path re-starts from the root activity. There are cases where 2 or more test cases might follow some common paths. We might consider some backtracking strategies to further reduce the time required for directed testing.

Work is also available in modeling the conditions in which infeasible paths of app transitions might occur. Currently, automated testing prioritizes transition of shorter paths, only trying longer paths if a shorter path is found to be infeasible or inaccessible. Future work might consist of modelling and uncovering the conditions under which infeasible paths might occur (For e.g. paths that require WiFi, specific geographic conditions, login status etc.). There are also possibilities in cataloging and modeling of implicit intents, such as for example, transition to external browser, email client or other app activities etc.

6. RELATED WORK

Automated App Testing. Automated testing of Android apps has received quite a bit of attention in recent years. The earliest and most frequently used Monkey [16] performs black-box testing by generating random UI events. Intent Fuzzer [23] randomly generates intents for fuzz testing Android apps for crash fault detection. Dynodroid [14] and VanarSena [20] are both also based on random exploration, but have improved techniques in event selection and mapping to increase testing efficiency. AMC [11] and DECAF [13] both utilize similar concepts in terms of extracting the DOM trees from the apps at run-time, and performing systematic exploration along running UI state information.

While these testers are effective for automated app testing, they do not perform guided and directed explorations to specific app activities or states. This is because they do not have access to path information obtained from static analysis of CCFGs, prior to testing. MAMBA is unique in this aspect. Our work leverages on recent work and advances in static control-flow analysis of user-driven callbacks and the flow of Android GUI-related objects, which provides a through and more accurate model for guiding app exploration. Perhaps most related to our work is A^3E [3], that has a targeted exploration strategy based on following a Static Activity Transition Graph (SATG). While this concept is similar, the SATG is based on a coarser-level analysis of sources and sinks in start-activity intents, and does not utilize static modelling of GUI-objects and flows in the app. It is also unclear how event handlers are mapped to the GUI-objects in their targeted exploration mode.

Privacy Protection Platforms. Various tools, such as TaintDroid, PiOS and PMP [1, 5, 6], have been developed for privacy leak detection in mobile apps. These tools are useful in detecting leaked private data, but are less useful in providing users with a summarized analysis of app privacy behaviours. The users usually have to undergo a certain period of app usage before app privacy behaviours can be uncovered. Our prior work [10] advocates a framework of utilizing dynamic analysis of apps for pre-deployment discovery of app privacy behaviours and characteristics, as well as discovering culprit leak widgets/buttons. This is the motivation for our work on efficient and accurate directed app exploration.

7. CONCLUSION

In this paper, we present MAMBA, a graph-aided directed testing system for automated checking of privacy behaviours in Android apps. MAMBA utilizes static analysis of control-flow graph of callbacks and path searches from starting activities to target activities of interest, so that test cases of transitions can be build for automated testing. Evaluating MAMBA with the Exhaustive Test approach showed that directed testing can achieve coverage of target activities 6.1 times faster on average, while still allowing proper verification of activities that access private data.

8. ACKNOWLEDGEMENTS

This research is supported by the National Research Foundation Singapore under its Interactive Digital Media (IDM) Strategic Research Programme. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the granting agency, or Singapore Management University.

9. REFERENCES

[1] Y. Agarwal and M. Hall. Protectmyprivacy: detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 97–110. ACM, 2013.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[3] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 48(10):641–660, 2013.

[4] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54. ACM, 2011.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[6] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 1–6, 2010.

[7] 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 *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.

[8] R. Holly. Using app permissions in android m, June 2015.

[9] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[10] J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan. The case for mobile forensics of private data leaks: towards large-scale user-oriented privacy protection. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 6. ACM, 2013.

[11] K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin. Amc: Verifying user interface properties for vehicular applications. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 1–12. ACM, 2013.

[12] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[13] B. Liu, S. Nath, R. Govindan, and J. Liu. Decaf: detecting and characterizing ad fraud in mobile apps. In *Proc. of NSDI*, 2014.

[14] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[15] [Online]. Android developers api class reference, http://developer.android.com/reference/classes.html, Oct. 2015.

[16] [Online]. Android monkey, http://developer.android.com/tools/help/monkey.html, Oct. 2015.

[17] [Online]. Hierarchy viewer android, http://developer.android.com/tools/help/hierarchy-viewer.html, Jan. 2016.

[18] [Online]. Managing the activity lifecycle, http://developer.android.com/training/basics/activity-lifecycle/index.html, Jan. 2016.

[19] [Online]. Protectmyprivacy (pmp) for android, http://www.android.protectmyprivacy.org/, Jan. 2016.

[20] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 190–203. ACM, 2014.

[21] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[22] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 143. ACM, 2014.

[23] R. Sasnauskas and J. Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM, 2014.

[24] A. Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1991.

[25] R. Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.

[26] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, pages 539–552, 2012.

[27] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *International Conference on Software Engineering (ICSE)*, 2015.