Automatic Construction of Callback Model for Android Application

Chenkai Guo*, Quanqi Ye‡, Naipeng Dong†, Guangdong Bai§, Jin Song Dong† and Jing Xu*
*College of Computer and Control Engineering, Nankai University
‡NUS Graduate School for Integrative Sciences and Engineering, National University of Singapore
†School of Computing, National University of Singapore
§Singapore Institute of Technology

***Abstract*—The heavy use of event-callback mechanism in frameworks like Android causes challenges for static analysis. Modelling of callback mechanisms for Android applications (app for short) is becoming a major method to address such challenges. In this work, we aim to construct a generic callback-related model that supports path-sensitive analysis. We consider three unresolved challenges in the existing modelling approaches: 1) building connections between different components; 2) identifying path-sensitive conditions; 3) handling the system-driven callbacks and fine-grained lifecycle callbacks. We propose algorithms for constructing a generic path-sensitive callback model and present a prototype model constructor, AndroChecker, to validate our approach. We evaluate 20 real-world apps using AndroChecker. The evaluation result shows that our method and tool have a strong capability in modelling path conditions and inter-component invocations.**

I. Introduction

Correctness and security of Android apps has become a critical concern and thus verification is needed. However, traditional flow analysis [1], [2] is hard to be directly adopted to Android apps, since Android apps do not have a fixed program entry. A promising way of static analysis is to create models of an Android app rather than directly analysing the app source code, due to its event-driven structures (e.g, listener interfaces, event callbacks, etc.). There are two ways of app modelling: *callback-directed* [3], [4] and *data-directed* [5], [6]. Callback-directed modelling captures event-callback sequences within the targeted app and represents them as a variation of control flow graph (*CFG*); whereas data-directed modelling identifies types of information related to the analysis goal and combines this information according to the flow sequences. The former way is more versatile and is able to provide logical structure over the entire app program. However, it can hardly provide concrete assist for analysis in practical scenarios, since most of the modelling targets, functionality testing or security verification, are closely combined with solid data input and output. Data-directed modelling has advantages in tackling specific analysis, but has a heavy cost in constructing a generic model. Neither of them is able to conduct a callback model in a generic and fine-grained manner.

In more details, prior app modelling suffers from incompleteness in three aspects. 1) Component types: only `activity` is involved in the modelling; other components like `service` and `broadcast receiver` are ignored. However, security compromises and logic bugs frequently exist in `service` and `broadcast receiver` components. 2) Callback types: only callbacks related to lifecycle and user interaction are taken into consideration. Actually, in Android, a variety of callbacks are driven by system events, like phone and location status. 3) Path-insensitive: in the existing callback modelling, the generated edges only present a *possible* flow from a start node to an end node, ignoring information such as how and when the flow is executed. There lacks a valid way to handle the conditions of the execution of possible flows.

In this work, we develop an approach for constructing generic path-sensitive callback (*GPC* for short) model for Android apps. There are three key insights underlying our approach: 1) components like `service` run in a parallel way with other components; 2) each of the non-lifecycle callbacks needs to register itself in their parent callbacks before execution; 3) the callback execution condition can be identified by analysing the register implementation along with possible paths. Our approach leverages the above insights to fully automate the model construction via static program analysis. We design and implement a proof-of-concept model builder AndroChecker, which receives Android *apk* files as input and outputs their corresponding *GPC* models. We apply AndroChecker to 20 real-world apps. The evaluation results show that our technique is both accurate and complete.

II. Construction Algorithm

In this section, we introduce the *GPC* construction algorithms, using an example shown in Figure 1 and Figure 2. The algorithm includes the following four stages.

A. Path-insensitive Inner Component Model (PII)

Taking the source code of a target component as input, the *PII* constructing process *ConsPII* outputs a path-insensitive model for the component, including valid nodes (set N) and edges between the nodes ($E_l \cup E_r$, E_l contains the edges connecting two lifecycle nodes, and E_r contains the edges produced by callback registering). For example, taking the source code in Figure 1 as input, *ConsPII* generates the black arrows and all the nodes in Figure 2.

The nodes in N can be simply identified by matching predefined regular expression. To compute E_l, *ConsPII* first initializes an entire lifecycle graph (*ELG*) in accordance with the

```
1   public class ShareMyPosition extends MapActivity implements LocationListener{
2     private LocationManager locationManager;
3
4     public void onCreate(Bundle savedInstanceState){
5       ...locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE); ...}
6
7     private void performLocation(boolean forceNetwork){
8       ...List<String> providers = locationManager.getProviders(true);   ...
9       boolean containsGPS = providers.contains(LocationManager.GPS_PROVIDER);
10      boolean containsNetwork = providers.contains(LocationManager.NETWORK_PROVIDER);
11      if ((containsGPS && !forceNetwork) || (containsGPS && !containsNetwork)) {
12        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 5, this);}
13      else {...finish();}}
14
15    protected void onResume(){
16      ...performLocation(false);}
17
18    protected void onPause(){
19      ...locationManager.removeUpdates(this);}
20
21    protected Dialog onCreateDialog(int id){
22      return new AlertDialog.Builder(this).setTitle(R.string.app_name)
23        .setView(sharedMapView)
24        .setNeutralButton(R.string.options, new OnClickListener() {
25          public void onClick(..., ...){
26            ...startActivity(option);}})
27        .setPositiveButton(R.string.share_it, new OnClickListener() {
28          public void onClick(..., ...){
29            ...startActivity(share);}})
30        .setNegativeButton(R.string.retry, new OnClickListener() {
31          public void onClick(DialogInterface arg0, int arg1){
32            ...performLocation(false);}}).create()}
33
34    public void onLocationChanged(final Location location){
35      ...locationManager.removeUpdates(this);
36        this.location = location;...}}
```

Fig. 1. Motivating example derived from ShareMyPosition [7].

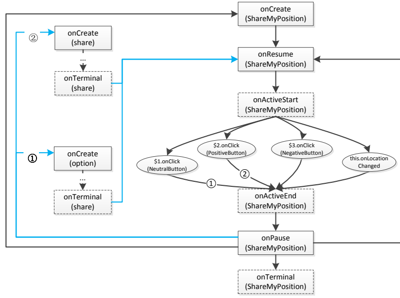

Fig. 2. GPC model illustration for the motivating example.

type of the given component. For the nodes in the *ELG* that are not implemented in target component (noted as *hiddenNodes*), *ConsPII* removes them and connects their children nodes to their corresponding parent nodes.

To identify the edges in E_r, *ConsPII* process first identify a set of register actions *FRA* (e.g., View.setOnClickListener, SensorManager.registerListener, etc.), each of which contains the invoker, invokee and the object conducting the (un)register action, then adopts an improved *DFS* (Deep First Search) algorithm over the entire program. The traversal process starts with `onCreate` callback in the launcher activity. For each identified register action, the invokee callback *ive* is inserted to the *active area* as a new node. The *active area* is a lifecycle interval between two auxiliary nodes (*onActiveStart* and *onActiveEnd*) where the non-lifecycle callbacks are normally invoked. There are two types of connections between lifecycle and non-lifecycle callbacks: *lifecycle* → *non-lifecycle* and *non-lifecycle* → *non-lifecycle*. The former generates edges *onActiveStart* → *ive* → *onActiveEnd*; and the latter generates edges *ivr* → *ive* → *onActiveEnd*, where *ivr* is the invoker in a register action.

B. Connections between Components (CBC)

Connections between components (noted as E_j) can be divided into two types: *sequential jumping* and *parallel jumping*. The former happens when the invokee occurs after the end of the invoker. It normally exists in the connections between activities, since the activities occupy the foreground (in running status) in the sequence of invocation. The latter occurs when services involve, because a service runs in the background in parallel with the foreground activity and other background services.

The CBC constructing process *ConsCBC* takes the sequential jumping as an edge between the invoker and invokee, similar as in *PII*. For parallel jumping, *ConsCBC* generates specialized *parallel edges* to represent the parallel relationship. Specifically, if the trigger component is an activity, we define *parallel slice* to represent the part of a triggering activity model that can run in parallel with the service. For example, service *S* is launched and stopped in activity *A* via the API `startService` and `stopService` respectively. Only the nodes that can run between `startService` and `stopService` are possible to run in parallel with *S*. The paths in *A* consisting of such nodes and their edges are the *parallel slice*. The parallel slice provides a fine-grained model for the parallel relation between the invoker and invokee. The algorithm traverses all the feasible paths that contain startNode (e.g., startService, bindService) and checks the paths that contain endNode (e.g., stopService, unbindService). In a path, the algorithm marks nodes between startNode and endNode as the parallel slice nodes. Note that in parallel jumping, if the service is terminated by itself, the parallel slice is the entire *PII* of the triggering activity after the startNode. The blue arrows in Figure 2 are generated by the *ConsCBC*.

C. Path-sensitive Inter-component Model (PSI)

The *PII* and *CBC* do not include the triggering conditions, which heavily limit their applications. To address this issue, we propose a path-sensitive modelling approach called *ConsPSI* (constructing the *PSI*). Essentially, *ConsPSI* aims at handling the triggering conditions for register edges E_r and inter-component connections E_j. Lifecycle edges E_l need not to be handled since they are triggered without extra conditions. For each edge in E_r and E_j, the path-sensitive model represents the set of conditions under which the invokee callback is fired. To achieve this goal, *ConsPSI* engages a backward data dependency analysis for critical program slice, which involves two core analyses: *checkCV* (for checking the condition variables) and *backTraverse* (for tracking the condition values). Since *ConsPSI* uses the same methods to handle E_j as to handle E_r, we only describe how to handle E_r for simplicity.

checkCV checks the triggering conditions for callback invocation and records the set of variables in the conditions. The checked objects are recorded in the register actions in *FRA*, as the invocation of a non-lifecycle callback heavily relies on the execution of the register actions. For each $fra \in FRA$, *checkCV* identifies the register actions branch using backward matching, and then records the branch conditions. Finally, *checkCV* abstracts all the variables within the branch conditions and stores them in a list *CVList*.

The *backTraversal* for variable dependency analysis is based on the *CVList*. The goal for *backTraversal* is to clarify the values of the condition variables. To this end, *backTraversal* tracks the variable assignment backward to update the *CVList*. The atomic unit of *CVList* is a variable tuple $vt = (var, val)$,

where *var* and *val* refer to the name and value of the variable respectively, and the *val* can be either a *vt* set or a value. The tuple *vt* keeps evolving during variable tracking. Eventually, all the *val*s in *vt* are updated to values. When the tracking ends in a method, the *val* is possible to be an undetermined variable. In this case, *backTraversal* identifies the types of these variables and marks them as *PARA* or *GLOBAL*, which are used to represent the parameter type and global type of *var* respectively. Hence, *backTraversal* typically records three types of leaf *val* at the end of tracking: constant value, *PARA* and *GLOBAL*. Eventually, the final results are treated as an attribute of the invoker node in *fra*.

The path-sensitive condition determines whether a callback is active; and the triggering event determines whether it is invoked. *ConsPSI* gets the corresponding triggering event tuple ($\langle$*trigering objects*, *trigering actions*, *trigered object*$\rangle$) in accordance to the invokee type, and then assigns the tuple to the invoker node as its attribute.

Limitations. Currently, the path-sensitive model only supports intra-procedural condition tracking, which limits the preciseness of the condition values. In fact, inter-procedural data dependency analysis exploits similar method in tracking the variables as the intra-procedural one. The differences embody in the process of handling the parameters. It requires extra costs to handle the inter-procedural data tracking. How to improve the preciseness of inter-procedural condition tracking in an acceptable time and resource cost is an interesting direction of our future work.

D. Jumping Confusion

In activity jumping, multiple jumping invokers existing in the same activity may cause *jumping confusion* on the matching between the invoker and invokee. A naive solution is to generate a direct edge connecting the invoker *ivr* with invokee *ive*. However, it ignores the related lifecycle nodes (i.e., `onActiveEnd` and `onPause`) in real sequence. Our analysis adopts a marking algorithm to address the jumping confusion. For each jumping edge E_{j-a}, we use a unique ID to mark the edge from its *ivr* to the *onActiveEnd* node, and then the same ID is used to mark the edge from *onPause* node to its *ive*. If two jumping actions point to the same *ive*, they share the same ID. An illustration example is shown in Figure 2. The edges \$1.*onClick* → *shareMyPosition.onActiveEnd* and *shareMyPosition.onPause* → *option.onCreate* are marked as ID ①, and other two similar edges are marked as ID ②.

III. Empirical Evaluation

We implemented the method in a prototype tool – AndroChecker, which is built based on AndroGuard [8] framework, and consists of 4,400 lines of Python code. The input of our tool is the *apk* file. The output is the set of nodes and edges, path condition set, the nodes of parallel slice and related statistical results. We perform evaluation on 20 real-world apps downloaded from in Google Play, which are used in prior works [4], [9]. Then, we study three typical cases to illustrate results of *PSI* in details.

A. Construction GPC model

Table I shows the evaluation results for each stage of analysis. Column "Application" shows the component sizes, where "A", "S" and "B" represent the number of activity, service and broadcast receiver, respectively. Column "PII" shows the results of the inner component model, including: lifecycle nodes N_l, lifecycle edges E_l, hidden nodes hN, non-lifecycle nodes N_n and register edges E_r. Column "CBC" shows the results of the inter component connections, including: the number of inter-component edges between activities E_{j-a}, the number of inter-component edges involving service E_{j-s} and the sum of parallel slice nodes $NSlc$. Column "PSI" shows the result of path-sensitive conditions. We present the total nodes N and total edges E in this stage. From Table I, two formulas can be observed, which are $N = N_l + N_n$ and $E = E_l + E_r + E_{j-a} + E_{j-s}$. Column "Cons" represents the path sensitive condition number of callback flow in each app. As mentioned earlier, currently AndroChecker only supports inner-procedural conditions traversal. So the conditions residing out of the callback holder method would not be analysed.

We omit the results of marked edges (mentioned in Section II-D) since they are the same as E_{j-a}. Column "Time" shows the time cost of our approach. It includes two parts: parsing time (*PT*) and modelling time (*MT*). Since AndroChecker directly handles *apk* file, the process of reverse engineering consumes significant time cost. This part of work is done by AndroGuard, which is not counted. On the whole, the number of *PT* and *MT* are related to the components' size.

B. Path Sensitive Conditions (PSC) Discussions

We further manually study the *PSC* results on three typical apps shown in Table II, where the column "PSC" represents the simplified *PSC* results; column "N" shows the times a *PSC* appears; and column "F" shows the false positive (FP) in *N*.

1) Beem: Beem provides a full featured and easy to use XMPP client on Android. As shown in Table I, app *Beem* contains four *PSC*s. By manual analysis, the real conditions contains: ① `if ( SharedPreferences paramSharedPreferences.getBoolean("use_auto_away", false) && "use_auto_away".equals(paramString) )`; ② `if(LoginAnim.this.mTask.getStatus()==AsyncTask.Status.PENDING)`; ③ *FP*. For ① and ②, the conditions are abstracted successfully. As for ③, the jumping action `bindService` is directly contained by `onResume`, which indicates ③ is an *FP*. The reason is that the back traversal of AndroChecker can not correctly identify the loop structure. In details, AndroChecker treats `while(){} bindService;` as `while(){bindService;}`. Currently, in the codes generated by AndroGuard, distinguishing the two structures is intractable. This provides a direction for future improvement.

2) MyTracks: MyTracks helps travellers to look for a way to keep tracking the places that they have been to. By manual analysis, the real *PSC*s can be abstracted as ① `if(Utilities.isGpsOn(getApplicationContext()))`, ② *FP*, ③ `if (GooglePlayServicesUtil.a(localContext, i).str1!=null)`. For ①, the real *PSC* is successfully

TABLE I
EVALUATION RESULTS FOR GPC MODEL CONSTRUCTION ALGORITHMS.

Name	Application			PII					CBC			PSI			Time(sec)	
	A	S	B	N_l	E_l	hN	N_n	E_r	E_{j-a}	E_{j-s}	NSlc	N	E	Cons	PT	MT
APV	4	0	0	26	41	18	31	62	9	0	0	57	112	6	9	14
Astrid	57	16	27	304	400	329	118	236	8	0	0	422	725	15	117	125
BarcodeScanner	9	0	1	50	84	44	16	32	4	0	0	66	120	3	18	38
Beem	13	1	7	97	193	49	28	56	18	2	-	125	269	4	43	63
ConnectBot	11	1	1	76	114	51	49	98	9	1	-	125	222	17	47	62
FBReader	9	9	9	86	110	52	29	58	1	2	-	115	171	12	128	141
K9	32	9	8	196	246	177	154	308	24	4	-	350	582	11	66	97
KeePassDroid	24	1	2	115	144	138	36	72	6	2	4	151	224	16	11	18
Mileage	19	13	0	176	209	98	79	158	3	8	40	255	378	11	332	236
MyTracks	7	1	1	42	64	37	41	82	5	2	9	83	153	3	23	35
NotePad	1	0	0	6	10	4	1	2	0	0	0	6	12	0	19	23
NPR	31	13	51	235	370	153	217	434	10	19	-	452	827	32	124	139
OpenManager	53	16	5	321	596	272	272	544	75	8	-	593	1223	1	119	121
OpenSudoku	10	0	0	62	88	49	39	78	14	0	0	101	180	6	3	5
SipDroid	21	3	16	127	203	106	48	96	14	7	16	175	312	10	15	26
SuperGenPass	2	0	1	11	19	10	18	36	1	0	0	29	56	0	5	6
TippyTipper	83	11	102	536	848	359	373	746	362	127	817	909	2083	58	251	376
VLC	24	11	11	189	248	112	211	422	23	6	15	400	699	10	99	113
VuDroid	5	0	0	22	30	30	4	8	0	0	0	26	38	0	1	2
XBMC	3	0	0	11	15	19	0	0	2	0	0	11	17	0	0.1	0.1

TABLE II
EXAMPLE PATH SENSITIVE CONDITIONS IN STUDIED CASES.

Name	PSC(smali)	N	F
Beem	if-eq{SharedPreferences;→getBoolean (String;Z)Z, String;→equals(PARA;)Z}	2	0
	if-eq{AsyncTask;→getStatus()AsyncTask $Status;,GLOBAL}	1	0
	if-eq{ChangeStatus;→access$1400(ChangeStatus;) Landroid/widget/Button;}	1	1
My-Tracks	if-eq{Utilities;→isGpsOn(Context;)Z}	2	1
	if-ne{GooglePlayServicesUtil;→a(Context; I)String;}	1	0
Open-Sudoku	if-ne{InputMethod;→isInputMethodViewCreated()Z'}	1	0
	if-eq {File;→isDirectory()Z, File;→isFile()Z}	1	1
	if-ne {Iterator;→hasNext()Z}	4	0

matched as shown in Table II. For ②, the register action is directly contained by *onCreate*, and the reason of *FP* is the same as in *Beem*. For ③, the *PSC* abstracted by our approach lacks the variable *str1*. This is because *str1* is an attribute of *GooglePlayServicesUtil.a*, which can not be recognized by AndroChecker.

3) OpenSudoku: OpenSudoku is an open source sudoku game. AndroChecker detects six *PSC*s in *OpenSudoku*. Through manual analysis, the real *PSC*s can be abstracted as ① `if(!(InputMethod).isInputMethodViewCreated())`, ② *FP*, ③ `if(!Iterator.hasNext())`. The *PSC* ① and ③ are successfully detected by AndroChecker. Similarly, the *FP* occurs because of the loop obstacle.

IV. CONCLUSION

Prior work targeting at constructing control-flow based model in Android, provides limited benefit when applied to real systems. We proposed GPC, a generic representation with fine-grained path information, and developed algorithms for its construction and traversal. We described our proof-of-concept builder AndroChecker and presented the evaluation results on real apps, which evidenced the efficiency of AndroChecker.

V. ACKNOWLEDGEMENT

This research was supported by the National Natural Science Foundation of China (Grant No. 61402264), the Science and Technology Planning Project of Tianjin (Grant No. 13ZCZDGX01098), and the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No.NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.
[2] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.
[3] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs." in *NDSS*, 2013.
[4] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 89–99.
[5] 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 *ACM SIGPLAN Notices*, vol. 49. ACM, 2014, pp. 259–269.
[6] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, 2015.
[7] "Sharemyposition," https://f-droid.org/repository/browse/?fdfilter=shareMyposition&fdid=net.sylvek.sharem- yposition.
[8] A. Desnos, "Androguard: Reverse engineering, malware and goodware analysis of android applications," 2013.
[9] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 658–668.