Dynamic Slicing for Android

Tanzirul Azim, Arash Alavi, Iulian Neamtiu and Rajiv Gupta

May 26, 2019

Dynamic Slicing for Android

Abstract—**Dynamic program slicing is useful for a variety of tasks, from testing to debugging to security. Prior slicing approaches have targeted traditional desktop/server platforms, rather than mobile platforms such as Android. Slicing mobile, event-based systems is challenging due to their asynchronous callback construction and the IPC (interprocess communication)-heavy, sensor-driven, timing-sensitive nature of the platform. To address these problems, we introduce AndroidSlicer, the first slicing approach for Android. AndroidSlicer combines a novel *asynchronous slicing* approach for modeling data and control dependences in the presence of callbacks with lightweight and precise instrumentation; this allows slicing for apps running on actual phones, and without requiring the app's source code. Our slicer is capable of handling a wide array of inputs that Android supports without adding any noticeable overhead. Experiments on 60 apps from Google Play show that AndroidSlicer is *effective* (reducing the number of instructions to be examined to 0.3% of executed instructions) and *efficient* (app instrumentation and post-processing combined takes 31 seconds); all while imposing a runtime overhead of just 4%. We present three applications of AndroidSlicer that are particularly relevant in the mobile domain: (1) finding and tracking input parts responsible for an error/crash, (2) fault localization, i.e., finding the instructions responsible for an error/crash, and (3) reducing the regression test suite. Experiments with these applications on an additional set of 18 popular apps indicate that AndroidSlicer is effective for Android testing and debugging.**

Index Terms—**Mobile apps, Android, Dynamic analysis**

I. Introduction

While mobile platforms have been very successful – Android alone runs on more than 2 billion devices [1] – they are prone to development, testing, and reliability issues that affect users, developers, and manufacturers [2], [3]. We propose an approach for tackling such issues via dynamic slicing, a technique that isolates relevant code and data dependences in an execution. Slicing has proven useful in many contexts, from security (e.g., taint analysis) to debugging (e.g., fault localization) and testing, but prior slicing approaches have not targeted mobile platforms [4]–[6].

The smartphone platform, compared to "traditional" desktop/server platforms, has unique challenges and constraints. This renders traditional slicing approaches inapplicable and has allowed us to design novel solutions. First, reconstructing control flow is difficult – Android apps employ callbacks which process events *asynchronously*. To cope with this, we introduce the concept of *asynchronous dependences* to capture control- and data-dependences between callbacks as *supernodes* in a *supergraph*. Second, Android apps are time-sensitive: even slight delays in sensor input can change input semantics, perturbing app execution. We use an on-demand static-after-dynamic analysis to permit low-overhead yet precise slicing. Third, Android has a wide range of inputs: multi-touch gestures, sensors, mic, camera, etc. Capturing this input correctly without losing precision is challenging, and introduces significant overhead in other tools, e.g., Pin [7]. We solve this challenge by combining lightweight tracking (AF wrappers) with a layered abstraction (supergraph). Fourth, Android uses IPC heavily for inter- and intra-app communication, which requires tracking dependences from apps to system stores and across address spaces corresponding to multiple apps.

Section III presents our model and dependence rules. Section IV describes our approach to handling Android-specific challenges, e.g., sensor input, low overhead, and IPC.

Our implementation, AndroidSlicer, is described in Section V; AndroidSlicer works on Android apps running on actual phones and does not require app source code.

Next, we show how AndroidSlicer serves as a building block for constructing three applications that facilitate bug finding, bug fixing, and testing. First, *Failure-inducing Input Analysis*, i.e., finding the input parts that are relevant to an error or a crash and tracking their propagation to the error/crash site (Section VI-A). Second, *Fault Localization*, i.e., finding the part of the code (a set of instructions) responsible for an error or crash (Section VI-B). Third, we show how slicing helps reduce the number of tests for *Regression Testing* (Section VI-C). Our analyses are robust and scalable: we applied them to isolate causes of real bugs in widely-popular apps such as SoundCloud, Etsy, K9-Mail, and NPR News.

In Section VII we evaluate AndroidSlicer on 60 popular apps from Google Play. Of these, we manually analyzed 10 apps to check AndroidSlicer's correctness. Experiments indicate that AndroidSlicer is efficient: it typically instruments an app in just 19.1 seconds and slice computation during post-processing typically takes just 11.9 seconds. At most, slice computation took 293.5 seconds for the substantial Twitter app, whose bytecode size is 50.6 MB. Moreover, AndroidSlicer has a low runtime overhead, typically 4%. Finally, AndroidSlicer is effective: it manages to reduce large executions (on average, 14491 instructions) down to small slices (on average, 44 instructions) that are much more manageable for developers. In summary, this work makes four contributions:

1) A novel asynchronous slicing model.
2) A precise approach for slicing Android apps that addresses the specific challenges and constraints of this platform and environment.
3) AndroidSlicer, a tool for slicing Android apps while they run on actual phones.
4) A novel approach to reducing runtime overhead and handling the large array of inputs supported by Android.
5) Three slicing applications that facilitate development, debugging, and testing for Android apps.

II. Background

We first present a brief overview of program dependence graphs and program slicing; next, we discuss the Android platform and its event-based model.

Program dependence graph (PDG). A PDG captures a program's data and control dependences. Each PDG edge represents a data or control dependence between nodes that can either represent an instruction or a basic block. A directed data dependence edge $n_j \leftarrow_d n_i$ means any computation performed in n_i depends on the computed value at node n_j. A control dependence edge $n_j \leftarrow_c n_i$ indicates that the decision to execute n_i is made by n_j, that is, n_j contains a predicate whose outcome controls the execution of n_i. A static PDG consists of all possible data and control dependence relations. A dynamic PDG is a subgraph that contains only those nodes and edges that are exercised during a particular run.

Program slicing. Dynamic program slicing, a class of dynamic analysis, was introduced by Korel and Laski [4]. The backward dynamic slice at instruction instance s with respect to *slicing criterion* $\langle t, s, value\rangle$ (where t is a timestamp) contains executed instructions that have a direct or indirect effect on $value$; more precisely, it is the transitive closure over dynamic data and control dependences in the PDG starting from the slicing criterion. The slicing criterion represents an analysis demand relevant to an application, e.g., for debugging, the criterion means the instruction execution that causes a crash.

Android platform. The Android software stack comprises of: apps; a middleware component named Android Framework (AF) which orchestrates control flow and mediates inter- and intra-app communication, as well as communication between apps and the lower layers; libraries and services; a run-time environment;[1] and the OS, an embedded version of Linux.

Android's event-based model. Android apps do not have a main() method but rather consist of components (e.g., an app with a GUI consists of screens, named Activities[2]) and one or more entry points. Unlike traditional programs, apps use an event-driven model that dictates control flow. An event can be a user action (e.g., touch), a lifecycle event (e.g., onPause() when the app is paused), arrival of sensor data (e.g., GPS), and inter- or intra-app messages. All these traits, from externally-orchestrated control flow to time-sensitive sensor input, render traditional slicing approaches inapplicable to Android.

III. Model

In this section we present the model underlying our approach. Slicing *precision* depends on accurately identifying control and data dependences – these dependences form the PDG. Our callback-centric design incurs low-overhead event capture without sacrificing precision (all input data is captured natively from the framework). Instead of considering a single instruction instance as a node, we collectively define callbacks as a node containing other nodes (instructions) or a supernode. Just like a regular node, callbacks or supernodes can invoke other events/callback directly (a control dependence), or by passing argument to the framework which in turn are passed to another callback (a data dependence). Our model captures both of these dependences for callbacks/supernodes, and instructions/single nodes. We use a "hierarchical" PDG, constructed as follows. High-level *supernodes* N represent callbacks and their dynamic context; *superedges* represent asynchronous control- or data-dependences between supernodes. Within supernodes, we use low-level instruction nodes S_{it}, and edges which capture sequential dependences.

Supernodes N, the core of our model, are defined as:

$$N := \langle c, t, a, \{[S_{it} \leftarrow_c S_{jt} \mid S_{it} \leftarrow_d S_{jt}]*\}\rangle$$

where callback variables c contain the address of a callback, t is the timestamp for node creation, a is the activity context (activity's runtime state), while S_{it} (a regular node) represents the instance of instruction S_i at time t. Data and control dependences are denoted $\leftarrow_d$ and $\leftarrow_c$, respectively. Superedges connect supernodes and regular edges connect regular nodes. Note that supernodes might contain sub-graphs (consisting of regular nodes $S_{it} \leftarrow_c S_{jt}$ or $S_{it} \leftarrow_d S_{jt}$); hence the hierarchical PDG notion. We now explain our dependence rules, shown in Figure 1.

A. Asynchronous Dependences

Asynchronous data dependence. Intuitively, asynchronous data dependences capture communication via message passing (IPC, objects, etc.). We denote the set of registers defined in callback c_1 as $Def(c_1)$; and the reference stored in register v_x at time t as $ref(v_x, t)$. Then, callback instance c_2 is data-dependent on callback instance c_1 (i.e., $N_1 \leftarrow_d N_2$) if at time t, c_1 defines an inter- or intra-process messaging object (intent) in a register v_1, and c_2 receives the same reference in register v_2 as a parameter. This also introduces a data dependence from the first instruction S_{2t} in c_2 that uses v_2, to the instruction S_{1t} in c_1 that defines v_1. Depending on the callback, data dependence can be inter-app or intra-app; we track both.

Asynchronous control dependence. Effective asynchronous slicing hinges on capturing dependences between asynchronous events precisely, via superedges $N \leftarrow_c N_2$ whenever N determines (initiates) the execution of N_2 via activity context transitions. Being an event-based model, Android's runtime system switches between different UI states ("Activity contexts") when asynchronous callbacks are invoked. Assuming the current activity context is a_2, the current callback is c_2 whose supernode is N_2, the previous activity context was a_1 and its 'exit' (i.e., callback that triggered the context transition) was c_1 whose supernode is N_1, we use the shorthand $initiator(N_2) = N_1$ to indicate that N_1 has triggered the transition to a_2. We define two rules that capture who has initiated N_2. The first rule captures a simple transition – no intent passed between the two events, i.e.,

[1]Dalvik VM for Android prior to 5.0; in Android version 5.0 and later, Android uses ART, a runtime system and ahead-of-time compilation. Our implementation (Section V) is on Android 5.1.1.

[2]The vast majority of Google Play apps consist of Activities. There are other component types such as Services, Content Providers, and Broadcast Receiver [8] but these are used much more sparsely.

$$\boxed{\text{Asynchronous data dependence}}$$

$$\frac{N_1 = \langle c_1, t_1, a_1, \ldots \rangle, N_2 = \langle c_2, t_2, a_2, \ldots \rangle \quad v_2 \in param(c_2), v_1 \in Def(c_1), ref(v_1, t_1) = ref(v_2, t_2)}{N_1 \leftarrow_d N_2 \qquad S_{1t} \leftarrow_d S_{2t}}$$

$$\boxed{\text{Sequential data dependence}}$$

$$\frac{v_1 \in S_{1t}, v_2 \in S_{2t}, v_1 \in dins(S_{1t}), v_2 \in uins(S_{2t}) \quad ref(V_{1t}) = ref(V_{2t})}{S_{1t} \leftarrow_d S_{2t}}$$

$$\boxed{\text{Asynchronous control dependence}}$$

$$\frac{N_1 = \langle c_1, t_1, a_1, \ldots \rangle, N_2 = \langle c_2, t_2, a_2, \ldots \rangle, a_1 \neq a_2, initiator(N_2) = N_1, \neg(N_1 \leftarrow_d N_2)}{N_1 \leftarrow_c N_2}$$

$$\frac{N_1 = \langle c_1, t_1, a_1, \ldots \rangle, N_2 = \langle c_2, t_2, a_2, \ldots \rangle, a_1 \neq a_2, initiator(N_2) = N_1, N_1 \leftarrow_d N_2, N_0 \leftarrow_c N_1}{N_0 \leftarrow_c N_2}$$

$$\boxed{\text{Sequential control dependence}}$$

$$\frac{isBool(S_{1t}) = true \quad S_{ipdt} = ipd(S_{1t}), isPost(S_{2t}, S_{ipdt}) = true}{S_{1t} \leftarrow_c S_{2t}}$$

Fig. 1: Dependence rules.

$\neg(N_1 \leftarrow_d N_2)$.[3] The second rule applies when N_2 is data dependent on N_1; in that case, the initiator is that super node N_0 that N_1 is control-dependent on. We then apply the same rule recursively on N_0. Put otherwise, these two rules help capture event *causality* by finding which event N caused event N_2; this is recorded as superedge $N \leftarrow_c N_2$.

B. Sequential Dependences

Sequential data dependence is captured by tracking the propagation of values through instructions *sequentially*, that is, control flow does not leave the current callback or its callees. Note that Android is a register-based machine, hence registers are used to hold values, pass values in and out of methods, perform computation, etc. We denote the set of registers defined in instruction s as $dins(s)$, and the set of registers used in instruction s as $uins(s)$. Then, instruction S_{2t} is data-dependent on instruction S_{1t} if at least one of the registers v_2 used in S_{2t} is defined in S_{1t} in a particular execution at time t. In other words, this register appears in both the set of definitions $dins(S_{1t})$ and the set of uses $uins(S_{2t})$; a condition also known as V_{2t} being "live" in S_{2t}.

Sequential control dependence. For instructions that are executed sequentially, certain predicates govern control flow, i.e., determine which instructions to execute next. Let $ipd(s_i)$ be the immediate post dominator of instruction s_i – always a unique instruction s_j. The check $isPost(s_i, s_j)$ returns true iff s_j is a post dominator of s_i. We define S_{2t} as being control-dependent on S_{1t} if S_{1t} is a conditional (e.g., if) and S_{2t} belongs to the set of instructions between S_{1t} and its immediate post-dominator S_{ipdt}.

IV. Android Slicing Challenges

We now show how the Android programming model/platform introduce challenges for constructing a dynamic slicer, and discuss how we have overcome these challenges.

A. Low Overhead

Dynamic slicing (as with any dynamic analysis) on mobile platforms must not *interfere* with the execution of the app that is being analyzed. Mobile apps do not tolerate delays gracefully: we illustrate this with three examples. First, even just attaching the standard dynamic analyzer Pin [7] to an Android app – a trivial operation on desktop/server – can have unacceptable overhead, or outright crash the app. To do the comparison with Pin, we instrumented 6 well-known apps using AndroidSlicer and Pin (for Pin we used a simple instrumenter that prints the number of dynamically executed instructions, basic blocks and threads in the app). Table I presents the results. We used Monkey with default settings to send the apps 5,000 UI events. Note that Pin instrumentation crashes all of the apps while AndroidSlicer instrumentation has a low overhead of 5%. Second, introducing delays in GUI event processing can alter the semantics of the event: an uninstrumented app running at full speed will interpret a sequence of GUI events as one long swipe, whereas its instrumented version running slower might interpret the sequence as two shorter swipes [9]. Third, harmful interference due to delays in GPS timing, or in event delivery and scheduling can easily derail an execution [10].

TABLE I: AndroidSlicer and Pin comparison.

App	**Original run** (s)	AndroidSlicer **run** (s)	**Pin run** (s)	AndroidSlicer **overhead** (%)
Indeed Job Search	15.8	17.1	Crashed at 14.7	8
Geek	29.4	32.2	Crashed at 17.4	9
Scanner Radio	29.5	30.9	Crashed at 15.1	5
Daily Bible	23.9	24.2	Crashed at 23.6	1
CheapOair	21.7	22.8	Crashed at 12.2	5
Kmart	24.5	25.2	Crashed at 14.6	3

Our approach. We address this challenge by optimizing register tracking at the AF/library boundary. First, in the runtime tracing phase, for a call into the AF/library we only track live registers, and only up to the boundary; upon exiting the AF/library we resume tracking registers. Second, in the static analysis phase we compute taint (source $\rightarrow$ sink) information to identify those methods that take values upward to the AF (sources) as well as those methods which return values downward to the app code (sinks). Finally, in the trace processing phase we instantiate the static taint information with the registers tracked into and out of the framework: this ensures sound yet efficient tracking.

B. High-throughput Wide-ranging Input

Android apps are touch- and sensor-oriented, receiving high-throughput, time-sensitive input from a wide range of sources. Typical per-second event rates are 70 for GPS, 54 for the camera, 386 for audio, and 250 for network [10]. A simple swipe gesture is 301 events per second [9]. Thus, we require low-overhead tracking of high-throughput multi-sourced input.

[3]Without loss of generality we ignore other callbacks c that precede c_2 in the current context a_2; c and c_2 are both available to dispatching hence cannot be control-dependent upon each other.

Our approach. Android employs AF-level event handlers for capturing external events. We achieve both scalability and precision by intercepting the registers at event processing boundary, as illustrated next. Swipes are series of touches, with the event handler onFling(MotionEvent e_1, MotionEvent e_2, float $velocityX$, float $velocityY$). We intercept the event by tracking the registers that hold the event handler parameters, i.e., e_1, e_2, $velocityX$, $velocityY$, and tagging them as *external inputs*. This approach has two advantages. First, register tracking is efficient, ensuring scalability. Second, being able to trace program behavior, e.g., an app crash, to a particular external input via a backward slice allows developers to "find the needle in the haystack" and allows us to perform efficient and effective fault localization (Sections VI-A and VI-B). Although our implementation targets Android, it is agnostic of the low-level OS layer.

C. Inter- and Intra-app Communication

Android relies heavily on IPC. The fundamental IPC mechanism is called Intent: using an intent, an activity can start another activity, or ask another app for a service. For example, the Facebook app can send an intent to the Camera app asking it to take a picture; the picture is returned via an intent as well. There are two types of intents: *implicit* and *explicit*. An implicit intent starts *any* component that can handle the intended action, potentially in another app; therefore, an implicit intent does not name a specific destination component. An explicit intent specifies the destination component (an Activity instance) by name. Explicit intents are used intra-app to start a component that handles an action.

Implicit intents and consequently, inter-app communications, complicate slicing. We illustrate this in Figure 2.[4] The example shows the GetContacts activity that allows the user to pick a contact. An intent can launch an activity via the startActivity or startActivity ForResult methods. Upon completion, Android calls the onActivityResult method with the request code that we have passed to the startActivity ForResult method (line 5 in the example). Without understanding the impact of inter-app intents, we would not be able to find complete slices. Assume we want to compute the slice with respect to variable name starting at statement 14. The resulting slice should contain statements {14, 9, 10, 12, 13, 8, 11, 5, 4}. However, traditional slicing would not find the complete slice because it only adds statements {14, 13, 12, 11, 10, 9} to the slice – it will miss statements 4 and 5 for two main reasons. First, traditional slicing fails to pair startActivityForResult with onActivityResult – which are similar to a caller-callee – and thus it fails to reconstruct control flow to account for IPC. Second, note how we cross memory spaces into the Contacts app, hence we need to account for Android's sandboxing to be able to trace the initial (request) and result intents.

Explicit intents also complicate slicing, as shown in Figure 3. The example shows ActivityOne starting ActivityTwo; the message "Some Value" is passed via IPC, the Bundle in this case. Consider computing the slice with respect to variable value starting at statement 8. The dynamic slice should contain statements $\{8, 7, 4, 3, 2\}$. However, traditional slicing cannot find the precise slice because it does not account for intra-app communication. Specifically, the example uses Bundle's putExtra and getExtra to pass data between the two activities; the Bundle is a system component, so in this case the dataflow is mediated by the system, and would elude a traditional slicer. Hence traditional slicing would not traverse statements $\{4, 3, 2\}$ due to the missing dependences between the two activities and would yield slice $\{8, 7\}$ which would be *incorrect*.

Our approach. To address challenges due to inter- and intra-app communication, we analyze app inputs and track callbacks and AF APIs to construct asynchronous data dependence edges accordingly (Section III-A). For example, if an activity calls another activity by sending an intent via startActivity or sendBroadcast, we trace the receiver callback and the parameter referencing the intent. For example we introduce data dependences $n \leftarrow_d m$, where n is the node associated with the instruction that sends the broadcast with the intent as parameter reference in a register v_n, while m is the node associated with the instruction that receives the intent, by reference, in register v_m.

Analyzing intra-app communication is complicated by several factors. Android allows developers to only receive intents within the app's context for certain internal activities. Intercepting these intents to construct data-flow facts requires further analysis of app bytecode. The task can be challenging, e.g., for intents that receive custom Parcelable objects (Android's form of serialization). To address this challenge, we add instrumentation routines to track the registers containing intent references. These intents are passed as parameters in different AF or API calls.

To summarize, by recording callbacks and intents, AndroidSlicer captures inter-app and intra-app communication precisely, with no under- or over-approximation.

V. Implementation

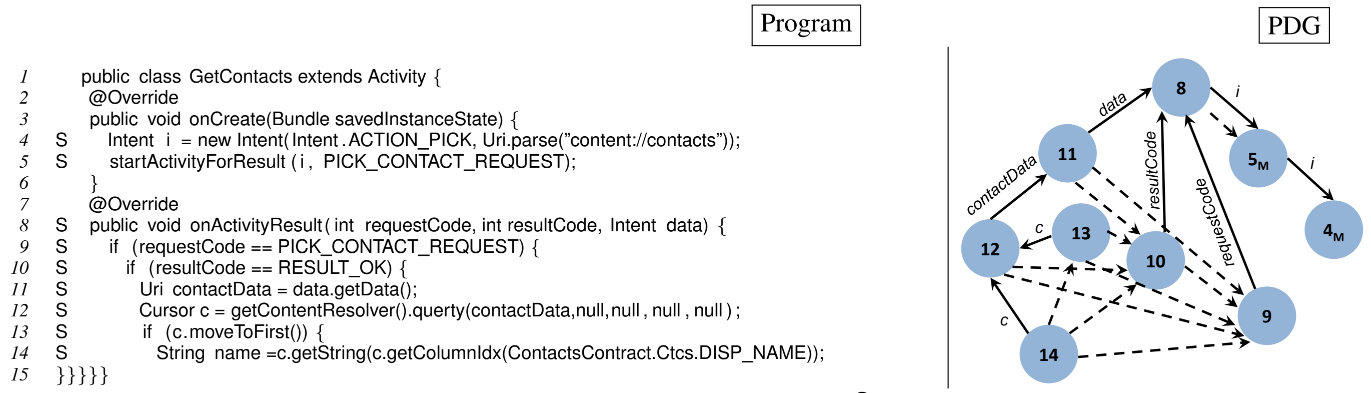
In this section, we describe AndroidSlicer's implementation. An overview is shown in Figure 4. In the first stage, the app is instrumented to allow instruction tracing. Next, as the app executes, runtime traces are collected. We perform an on-demand static analysis to optimize trace processing, and then compute the PDG. Finally, we calculate slices for a given slicing criterion. We now discuss each phase.

A. Instrumentation

The purpose of this stage is three-fold: identify app entry points; construct method summaries; and add instruction/metadata tracing capabilities to the app.

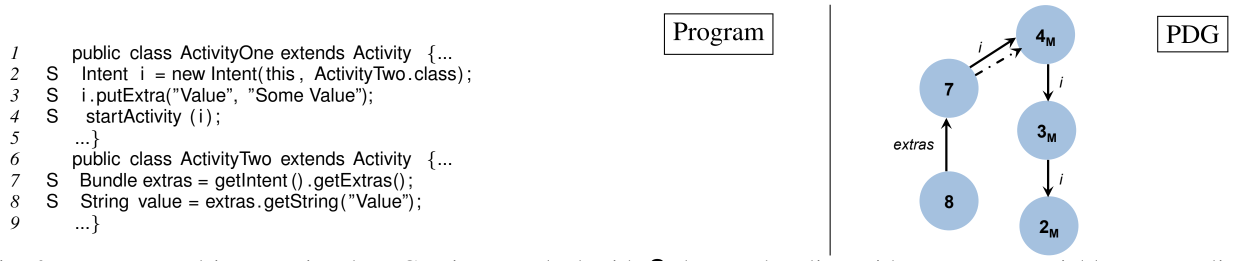
Constructing method summaries. Method summaries (which include in/out registers and method type) capture method information for the online trace collection phase (Section V-B). To compute summaries, we first build a callgraph for each app class from the analyzed app entry points. For

[4]The purpose of the PDGs in Figures 2 and 3 is to compare with traditional slicing, hence we only show the "regular" PDGs without supernodes. Figures 5 and 6 show supernodes and superedges.



```
1        public class GetContacts extends Activity {
2         @Override
3         public void onCreate(Bundle savedInstanceState) {
4    S      Intent i = new Intent(Intent.ACTION_PICK, Uri.parse("content://contacts"));
5    S      startActivityForResult (i, PICK_CONTACT_REQUEST);
6         }
7         @Override
8    S    public void onActivityResult( int requestCode, int resultCode, Intent data) {
9    S      if (requestCode == PICK_CONTACT_REQUEST) {
10   S        if (resultCode == RESULT_OK) {
11   S          Uri contactData = data.getData();
12   S          Cursor c = getContentResolver().querty(contactData,null,null, null, null);
13   S           if (c.moveToFirst()) {
14   S             String name =c.getString(c.getColumnIdx(ContactsContract.Ctcs.DISP_NAME));
15     }}}}}
```

Fig. 2: Program and its associated PDG. In the program: lines marked with an S denote the slice with respect to variable name on line 14. In the PDG: solid edges denote data dependences; dashed edges denote control dependences; graph nodes marked with an M denote nodes that would be missed by traditional slicing techniques. Labels on solid edges denote the variables which cause the data dependence.



```
1        public class ActivityOne extends Activity {...
2    S    Intent i = new Intent(this, ActivityTwo.class);
3    S    i.putExtra("Value", "Some Value");
4    S    startActivity (i);
5        ...}
6        public class ActivityTwo extends Activity {...
7    S    Bundle extras = getIntent().getExtras();
8    S    String value = extras.getString("Value");
9        ...}
```

Fig. 3: Program and its associated PDG. Lines marked with S denote the slice with respect to variable value on line 8.

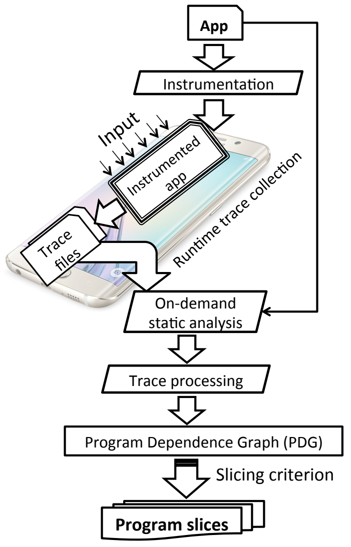


Fig. 4: AndroidSlicer overview.

each node in the callgraph (i.e., method) we add instrumentation tags that summarize that method. This instrumentation is an extended version of the method signature present in the Dexcode (Android bytecode); we save information for parameter registers and return value registers. We also detect callbacks at this time and add necessary information about input parameters. We identify intents referenced through registers used as callback parameters and construct metadata such as caller information (i.e., name of the callback-generating and broadcasting the intent), as well as string properties associated with the intent's action filter. This information helps reveal *callers* and their *callees* during offline trace analysis.

Adding tracing instructions. We add tracing capabilities via Soot [11]. AndroidSlicer's instrumenter takes the app binary as input; the output is the instrumented app, which we run on the phone. To support tracing, we inject a new Dexcode instruction for every app instruction or callback routine we encounter. The trace format is described next.

B. Runtime Trace Collection

We collect the traces while the instrumented app is running on the phone. Traces have the format:

```
trace_entry := <t, memory_offset, instruction, [summary]>
summary := <type, invoked_method, parameter_registers,
return_registers, callback_parameter_registers,
intent_source, intent_action_filters >
```

Trace entries have the following semantics: t is the actual time the instruction was executed (t in the model); memory_offset is the instruction's address; instruction is the opcode (op in the model); therefore we have the S_{it} from the model. Summary information is only used for method invocations; it contains the method's type, e.g., an IPC or a non-IPC call, in/out register values, caller information (i.e., current callback), and, where applicable, the action string (filter) associated with the intent.

C. On-demand Static Analysis

To build the PDG efficiently, we conduct a post-run on-demand static analysis that uses the collected runtime information to narrow down the scope of the static analysis to only those app parts that have been exercised during the run. The advantage of this on-demand approach is that, instead of statically analyzing the whole program (which for Android apps raises scalability and precision issues) we only analyze those methods encountered during execution. The on-demand analysis phase performs several tasks.

D. Trace Processing and PDG Construction

With the static analysis information at hand, we analyze the application traces to generate the PDG of that particular execution. The PDG is built gradually via backward exploration of dependences, adding nodes and edges as explained shortly. Our prior static analysis produces two sets: (1) $StaticData_{si}$ – the set of static data dependence nodes for an instruction s_i, and (2) $StaticControl_{si}$ – the set of static control dependence nodes for an instruction s_i. As mentioned in Section III, suffix t distinguishes between different occurrences of an instruction; the implementation uses a global counter for this purpose.

Sequential data dependence edges. For every occurrence of an instruction s_i, add a data dependence edge to the last executed occurrence of every instruction in its $StaticData_{si}$.

Sequential control dependence edges. For every occurrence of an instruction s_i, add a control dependence edge to the last executed occurrence of every instruction in its $StaticControl_{si}$.

Asynchronous data dependence superedges. For every occurrence of a callback $callee$ node, add a data dependence to the last occurrence of its $caller$. This information is revealed via static callback analysis. We also identify the instruction s_{ipct} that contains the actual IPC method call in the caller that passed the intent reference at time t. The callee receives the intent through one of its parameter registers v_{intent}. We then identify the last occurrence of the first instruction in $callee$ at time t that uses v_{intent}. Let us name this S_{int}. We then add a data dependence $S_{ipct} \leftarrow_d S_{int}$.

Asynchronous control dependence superedges. Based on our two asynchronous control dependence rules (Figure 1), if there is no data dependence between the corresponding supernodes from two consecutive activity contexts, i.e., callback $callee$ and its $caller$ (N_1 and N_2), we add a control dependence superedge $N_1 \leftarrow_c N_2$. Otherwise, we add a control dependence superedge $N_0 \leftarrow_c N_2$, where N_0 is the supernode N_1 is control-dependent on.

E. Generating Program Slices from the PDG

We now discuss our approach for generating slices given the PDG and a slicing criterion. Algorithm 1 provides the high-level description. The slicing criterion $\langle t, s_t, v_s \rangle$ represents the register v_s in instruction s at a particular timestamp t. Since an instruction is a regular node in the PDG we will use both terms interchangeably, i.e., s_t refers to both the instruction and the PDG node. We maintain a workset T_s that holds the nodes yet-to-be-explored (akin to the working queue in Breadth-first search). The output OUT_{st} is the set of distinct nodes in the PDG we encounter while backward traversing from s_t to any of the app entry points affecting the value held in register v_s. We first traverse the edges in the PDG starting from s_t and create a dynamic data dependence table Def_n and a control dependence table $Ctrl_n$ for each node n on paths to entry points. For each regular node n' in the set $P_n = Def_n \cup Ctrl_n$

Algorithm 1 Dynamic program slicing

Input: PDG, slicing criterion $SliceCriterion = (t, s_t, v_s)$
Output: set of nodes OUT_{st}

```
1:  procedure SLICE($SliceCriterion$)
2:      $T_s \leftarrow \{s_t\}$ // initialize workset $T_s$
3:      $OUT_{st} \leftarrow \{s_t\}$
4:      for all nodes $n$ that are in $T_s$ do
5:          calculate set $P_n = Def_n \cup Ctrl_n$
6:          for all nodes $n'$ in $P_n$ do
7:              if $n'$ is a supernode then
8:                  Expand & extract the last regular node $n_r$
9:                  Add $n_r$ to $Def_n$
10:             else if $n' \equiv n$ & $P_{n'} \equiv P_n$ then
11:                 Merge $(n', n)$; remove $n'$ from $P_{n'}$
12:             else if previous occurrence of $n$ is in $P_{n'}$ &
                $P_{n'} \subset P_n$ then
13:                 Merge $(n', n)$; remove $n'$ from $P_{n'}$
14:             else
15:                 add $n'_i$ to $OUT_{st}$; add $n'$ to $T_s$; remove $n$
                from $T_s$
16:             end if
17:         end for
18:     end for
19: end procedure
```

we add n' to OUT_{st}. If n' is a supernode and $n' \in Def_n$ we expand n'. The expansion adds the last occurrence of the regular node n_r inside n' that broadcasts an intent to Def_n and recalculates P_n. Note that n_r passes the IPC reference (intents) in a register to the next supernode, and hence it should be included in the slice. Since the same instruction can appear multiple times because of different occurrences at different timestamps, this procedure adds nodes with the same instructions to the slice for each occurrence. This increases the size of the slice. To reduce the number of nodes in OUT_{st} we make two optimizations.

1. Node merging. Given different occurrences (i.e., at times t and t') of a regular node (i.e., $n \equiv s_t, n' \equiv s_{t'}$) if $P_n = P_{n'}$ we merge n and n' into n_{merged}. For two different occurrences N and N' of the same supernode, we also apply merging: if N and N' have incoming or outgoing data dependence edges we expand the nodes and merge the individual instructions, i.e., regular nodes inside them; if N and N' are connected by control dependence edges only, we merge them.

2. Loop folding. In loops, for every new occurrence of a loop body instruction s, we will add a new node in the slice. But these nodes may point to the same set of data and control dependence in the PDG – they are different occurrences of s. To reduce these duplications, we merge two distinct nodes n and n' in the loop if the following conditions are met: (a) current occurrence of n' depends on the previous execution of n; (b) current occurrence of n depends on the current occurrence of n'; and (c) $P_{n'} \subset P_n$.

Let us call the new node created after the merge n_{merged}.

Each time we find a different occurrence of the merged node we compute the set $P_{n_{merged}}$. Then we apply reduction rule 1 to further reduce it to a single node.

F. Limitation

Since AndroidSlicer's instrumenter is based on Soot, it inherits Soot's static analysis size limitations, e.g., we could not handle extremely large apps such as Facebook. Note that this is not a slicing limitation per se, but rather a static analysis one, and could be overcome with next-generation static analyzers.

VI. Applications

We describe three applications that leverage AndroidSlicer to facilitate debugging and testing Android apps.

A. Failure-inducing Input Analysis

This analysis finds the input parts responsible for a crash or error. Note that unlike traditional programs where input propagation and control flow largely depend on program logic, in event-driven systems propagation depends on the *particular ordering of the callbacks associated with asynchronous events*. Leveraging our PDG, we can reconstruct the $input \rightarrow \ldots \rightarrow failure$ propagation path.

Problem statement. Let I be the set of app inputs $I_1, I_2, ...$ (e.g., coming from GUI, network, or sensors) through registers $v_1, v_2, ...$ Let the faulty register be v_{err}, i.e., its value deviates from the expected value (including an incorrect numeric value, crash, or exception). Hence the analysis' input will be the tuple $\langle I, v_{err}, PDG \rangle$ while the output will be a sequence of registers $v_1, v_2, ..., v_n$ along with the callbacks $c_1, c_2, ..., c_m$ the registers are used in.

Tracking input propagation. In the PDG, for every asynchronous callback, we can create an input propagation path by tracking the data dependence for the value of any register v_i. We determine whether the values propagated through registers are *influenced* by any of the app inputs I. This is particularly useful for identifying faults due to corrupted files or large sensor inputs (e.g., a video stream).

Example. We illustrate our analysis on an actual bug, due to a malformed SQL query, in Olam, a translator app.[5] The app takes an input word from a text box and translates it. In Figure 5 (top) we show the relevant part of the code: the instruction number (left), the actual instruction (center) and the value propagation through registers $v_1, v_2, ..., v_n$ along the PDG edges (right). In the method getSimilarItems, the app attempts to query the SQLite database, which generates an exception, resulting in a crash. The exception trace from the Android event log indicates that the query is ill-formed. The PDG (bottom left) points out the callback in which the exception was thrown: the onClick event associated with the search button in the MainSearch activity. We analyze the event inputs by following the data dependence edges backwards and see that the registers' values are pointing towards the input text from the textbox editText. We compute the slice using the faulty register reference as slicing criterion.

The execution slice is shown in Figure 5: we see that the ill-formatted string was stored in register v_1. Our approach back-propagates the value of v_1 through the slices to determine whether it was impacted by any part of the input. Back-propagation starts from the error location, i.e., instruction number 29754. The value propagates to register v_5 which references the return value from getText invoked on an instance of v_4 that is pointing to the GUI control element EditTextBox. Our analysis ends by returning the register v_5 with the corresponding callback information. The second part of the figure shows the associated supernodes which reveal that the executed slices belong to the MainSearch:onClick callback. The failure-inducing input was thus essentially identified analyzing a much smaller set of instructions, and more importantly, in the presence of non-deterministic callback orders.

B. Fault Localization

This analysis helps detect and identify the location of a fault in an app. For sequential programs, fault localization is less challenging in the sense that it does not need to deal with the non-determinism imposed by asynchronous events. Android apps are not only event-driven but also can accept inputs at any point of the execution through sensors, files, and various forms of user interactions. For this reason, fault localization on Android can be particularly challenging for developers.

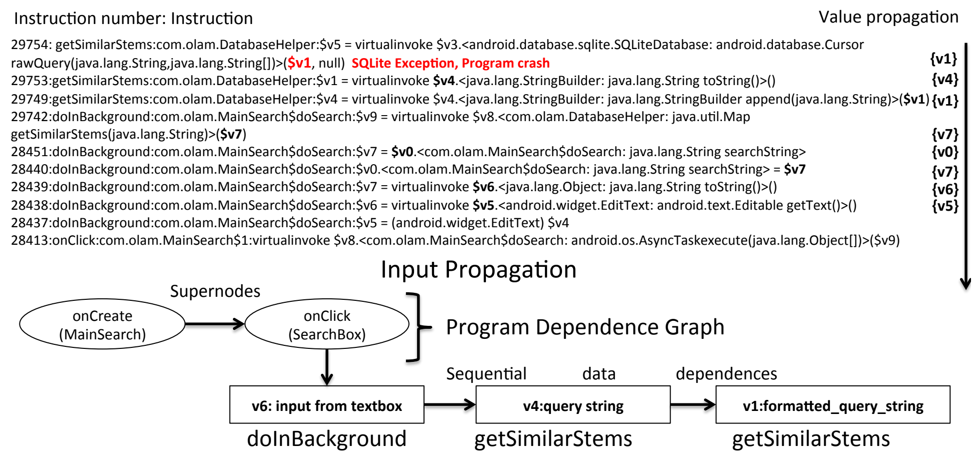
Problem statement. The input to the analysis will be the application trace, and the register v_{err} holding the faulty value in a specific occurrence of an instruction. The output this time will be the sequence of instructions $s_1, s_2, ..., s_n$ that define and propagate the value referenced in v_{err}.

Tracking fault propagation. Our slicing approach aids fault localization as follows. Given a fault during an execution, we determine the faulty value reference inside a register v_{err} by mapping the Android event log to our execution trace. Then we compute the execution slice for v_{err} by back propagating through the execution slice. While we traverse the PDG backwards, we consider asynchronous callbacks and their input parameters if they have a direct data or control dependence to the final value of v_{err}. This way, we can both handle the non-determinism of the events and also support the random inputs from internal and external sources.

Example. We illustrate our approach on a real bug in the comic book viewing app ACV.[6] Figure 6 shows the generated sequential and asynchronous dependences for the faulty execution. The bug causes a crash when the user opens the file explorer to choose a comic book. If the user long-taps on an inaccessible directory, the app crashes with a null pointer exception. From Figure 6 we can see that the object reference stored in register v_6 at instruction 7153 was the primary cause of the error. The corresponding callback is revealed to be onItemLongClick in activity SDBrowserActivity. Our analysis tracks back the object reference in v_6 through the slices, reaching instruction 6803. Here we can see a file system API invocation (java.io.File.getName()) that attempts to return a filename, but fails

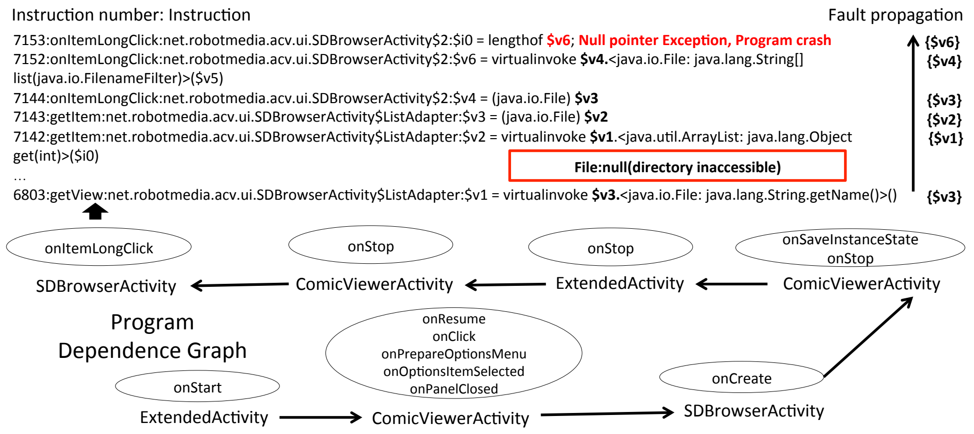
[5]https://play.google.com/store/apps/details?id=com.olam

[6]https://play.google.com/store/apps/details?id=net.androidcomics.acv



```
Instruction number: Instruction                                                                                      Value propagation
29754: getSimilarStems:com.olam.DatabaseHelper:$v5 = virtualinvoke $v3.<android.database.sqlite.SQLiteDatabase: android.database.Cursor
rawQuery(java.lang.String,java.lang.String[])>($v1, null)  SQLite Exception, Program crash                                    {v1}
29753:getSimilarStems:com.olam.DatabaseHelper:$v1 = virtualinvoke $v4.<java.lang.StringBuilder: java.lang.String toString()>()  {v4}
29749:getSimilarStems:com.olam.DatabaseHelper:$v4 = virtualinvoke $v4.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($v1) {v1}
29742:doInBackground:com.olam.MainSearch$doSearch:$v9 = virtualinvoke $v8.<com.olam.DatabaseHelper: java.util.Map
getSimilarStems(java.lang.String)>($v7)                                                                                      {v7}
28451:doInBackground:com.olam.MainSearch$doSearch:$v7 = $v0.<com.olam.MainSearch$doSearch: java.lang.String searchString>     {v0}
28440:doInBackground:com.olam.MainSearch$doSearch:$v0.<com.olam.MainSearch$doSearch: java.lang.String searchString> = $v7     {v7}
28439:doInBackground:com.olam.MainSearch$doSearch:$v7 = virtualinvoke $v6.<java.lang.Object: java.lang.String toString()>()   {v6}
28438:doInBackground:com.olam.MainSearch$doSearch:$v6 = virtualinvoke $v5.<android.widget.EditText: android.text.Editable getText()>()  {v5}
28437:doInBackground:com.olam.MainSearch$doSearch:$v5 = (android.widget.EditText) $v4
28413:onClick:com.olam.MainSearch$1:virtualinvoke $v8.<com.olam.MainSearch$doSearch: android.os.AsyncTaskexecute(java.lang.Object[])>($v9)
```

Fig. 5: Failure-inducing input analysis.



```
Instruction number: Instruction                                                                                      Fault propagation
7153:onItemLongClick:net.robotmedia.acv.ui.SDBrowserActivity$2:$i0 = lengthof $v6; Null pointer Exception, Program crash    {$v6}
7152:onItemLongClick:net.robotmedia.acv.ui.SDBrowserActivity$2:$v6 = virtualinvoke $v4.<java.io.File: java.lang.String[]    {$v4}
list(java.io.FilenameFilter)>($v5)
7144:onItemLongClick:net.robotmedia.acv.ui.SDBrowserActivity$2:$v4 = (java.io.File) $v3                                     {$v3}
7143:getItem:net.robotmedia.acv.ui.SDBrowserActivity$ListAdapter:$v3 = (java.io.File) $v2                                   {$v2}
7142:getItem:net.robotmedia.acv.ui.SDBrowserActivity$ListAdapter:$v2 = virtualinvoke $v1.<java.util.ArrayList: java.lang.Object  {$v1}
get(int)>($i0)
                                                                     File:null(directory inaccessible)
...
6803:getView:net.robotmedia.acv.ui.SDBrowserActivity$ListAdapter:$v1 = virtualinvoke $v3.<java.io.File: java.lang.String.getName()>()  {$v3}
```

Fig. 6: Fault localization.

because the file's directory is inaccessible. Our value propagation ends here, revealing the source of the error. We return the set of instructions $\{6803, ..., 7142, 7143, 7144, 7152, 7153\}$, and the traversed PDG nodes. For simplicity, we only show the data dependence edges and relevant parts of the slices. Our approach then back-propagates through the PDG according to the execution slice to localize the fault (for presentation simplicity we have combined consecutive supernodes in the same activity into a single node).

C. Regression Test Suite Reduction

Regression testing validates that changes introduced in a new app version do not "break" features that worked in the previous version. However, re-running the previous version's entire test suite on the new version is time-consuming and inefficient. Prior work [12], [13] has shown that slicing reduces the number of test cases that have to be rerun during regression testing (though for traditional apps).

Problem statement. Given two app versions (V_1 and V_2), and a test suite T_1 (set of test cases) that has been run on V_1, find T_2, the minimal subset of T_1, that needs to be rerun on V_2 to ensure that V_2 preserves V_1's functionality.

Test case selection. Agrawal et al. [12] used dynamic slicing to find T_2 as follows: given a program, its test cases, and slices for test cases, after the program is modified, rerun only those test cases whose slices contain a modified statement. This reduces the test suite because only a subset of program statements (the statements in the slice) have an effect on the slicing start point (program output, in their approach [12]). However, this technique can be unsound, because it only considers whether a statement has been modified, not *how it has been modified*. When the changed instructions affect predicates leading to an asynchronous control dependence (see Section III-A), missed control dependences will lead to potentially missing some test cases. Our approach considers such dependences to maintain soundness.

VII. Evaluation

We first evaluate AndroidSlicer's core slicing approach on 60 apps from Google Play; next, we evaluate it on the three applications from Section VI.

Environment. An LG Nexus 5 phone (Android version 5.1.1, Linux kernel version 3.4.0, 2.3 GHz Qualcomm Snapdragon 800 quad-core chip) for online and an Intel Core i7-4770 CPU (3.4 GHz, 24 GB RAM, 64-bit Ubuntu 14.04 kernel version 4.4.0) for offline processing.

A. Core Slicing

App dataset. We ran AndroidSlicer on 60 apps selected from Google Play. The apps were selected from a range of

TABLE II: AndroidSlicer evaluation: core slicing results.

App		**Dex code size (KB)**	**Installs (thousands)**	**Instructions**		**CD+DD**	**Callback events**	**Time (seconds)**				**Over-head (%)**
				Executed	**In slice**			**Stage 1 Instrumentation**	**Original run**	**Stage 2 Instrumented run**	**Stage 3 Slicing**	
Twitter		50688	500,000-1,000,000	107847	559	790	557	293.5	233.4	245.2	40.3	5
Evernote		7219	100,000-500,000	191304	22	34	17	81.3	221.7	228.6	45.9	3
Du Recorder		62338	50,000-100,000	36672	316	409	31	41.0	245.1	254.8	53.7	4
Indeed Job Search		2458	50,000-100,000	21752	246	356	235	24.0	222.2	230.3	12.1	4
Twitch		30106	50,000-100,000	2025505	5969	9429	5965	144.2	260.3	281.9	103.0	8
Across	*min*	*47*	*10-50*	*17*	*2*	*3*	*1*	*2.3*	*205.4*	*212.8*	*2.0*	*0*
all 60	*median*	*1,485*	*500–1000*	*14491*	*44*	*63*	*23*	*19.1*	*229.85*	*239.7*	*11.9*	*4*
apps	*max*	*78684*	*500,000-1,000,000*	*2025505*	*5969*	*9429*	*5965*	*293.5*	*260.7*	*281.9*	*103.0*	*14*

categories (from shopping to entertainment to communication) and with various bytecode sizes to ensure diversity in tested apps. In Table II we present detailed results for the top-5 apps sorted by number of installs. For brevity, we summarize the findings (min/median/max) in the last three rows. The second column shows the app's bytecode size. Note that the apps were substantial, with a median size of 1,485 KB (second-to-last row). The third column shows app popularity (number of installs, *in thousands*, per Google Play as of August 2018). While the median popularity is between 500,000 and 1,000,000 installs, 28 apps had more than one million installs.

Generating inputs and slicing criteria. To drive app execution, we used Monkey [14] to send the app 1,000 UI events and then collected traces for offline analysis. To measure AndroidSlicer's runtime overhead, same event sequence was used in instrumented and uninstrumented runs. As slicing criteria, variables were selected to cover all types of registers (local variables, parameters, fields) from a variety of instructions (static invokes, virtual invokes, conditions, method returns). This allows us to draw meaningful conclusions about slicing effectiveness and efficiency.

Correctness. We manually analyzed 10 out of the 60 apps to evaluate AndroidSlicer's correctness. The manual analysis effort in some apps can be too high, because of the large number of instructions and dependences (e.g., in the Twitch app, there are 5,969 instructions in the slice and 9,429 dependences). Therefore, we picked 10 apps whose traces were smaller so we could verify them manually with a reasonable amount of effort. We decompiled each app to get the Java bytecode, and manually created the corresponding PDG from the slicing criterion. Then we manually computed the slices based on the execution trace. The manually-computed slices were then compared with AndroidSlicer's; we confirmed that slice computation is correct, with no instruction being incorrectly added or omitted.

Effectiveness. Table II demonstrates that AndroidSlicer is effective. The "Instructions Executed" column shows the total number of instructions executed during the entire run. The median number of instructions is 14,491. If the programmer has to analyze these, the analysis task will be challenging. AndroidSlicer reduces the number of instructions to be analyzed to *44, i.e., 0.3%* (column "Instructions In slice"). The median number of dependences to be analyzed, data and control, is not much larger, 63, (column "CD+DD"). The next column shows the number of callback events fired during the run: the median was 23 across all apps.

Efficiency. The remaining columns ("Time" and "Overhead") show that AndroidSlicer is efficient. Stage 1 (instrumentation), typically takes just 19.1 seconds, and at most 293.5 seconds for the 50.6 MB Twitter app. The "Original run" column shows the time it took to run the original app, without our instrumentation – typically 229.85 seconds, and at most 260.7 seconds. Column "Stage 2 Instrumented run" shows the time it took to run the instrumented app, while collecting traces. The typical run time increases to 239.7 seconds. The "Overhead" column shows the percentage overhead between the instrumented and uninstrumented runs; the typical figure is 4% which is very low not only for dependence tracking, but for any dynamic analysis in general. Furthermore, our instrumentation strategy does not require monitoring the app or attaching the app to a third-party module – this allows the app to run at its native speed. We emphasize that AndroidSlicer's *low overhead is key* to its usability, because Android apps are timing-sensitive (Section IV-A). Finally, the "Stage 3 Slicing" column shows post-processing time, i.e., computing slices from traces, including on-demand static analysis; this time is low, typically just 11.9 seconds, and at most 103 seconds.

B. Failure-inducing Input Analysis

We evaluated this application on real bugs in 6 sizable apps (Table III) by reproducing the bug traces. Our failure-inducing input analysis is very effective at isolating instructions and dependences of interest – the number of executed instructions varies from 320 to 182527, while slices contain just 16–57 instructions. The CD and DD numbers are also low: 18–73.

C. Fault Localization

We evaluated our approach on 7 apps, including Notepad and SoundCloud that have in excess of 10 million and 100 million installs, respectively. Table IV shows the results. Note how fault localization is effective at reducing the number of instructions to be examined from thousands down to several dozen. SoundCloud and NPR News have large slices due to intense network activity and background services (audio playback), which increase the callback count substantially.

D. Regression Test Suite Reduction

We evaluated our reduction technique on 5 apps. For each app, we considered two versions V_1 and V_2 and ran a test suite T_1 that consisted of 200 test cases; on average, the suite achieved 62% method coverage. Next, we used AndroidSlicer to compute the reduced test suite as described in Section VI-C.

TABLE III: AndroidSlicer evaluation: Failure-inducing input analysis.

App	Dex code size (KB)	Installs (thousands)	Instructions		CD+ DD	Call-back events	Time (seconds)				Over-head (%)
			Executed	In slice			Stage 1 Instrumentation	Original run	Stage 2 Instrumented run	Stage 3 Slicing	
Etsy	5400	10,000–50,000	182527	19	24	9	94	8.7	10.4	129.2	19
K-9 Mail	1700	5,000–10,000	13042	30	34	16	89.1	107.4	125.3	58.8	16
AnyPlayer Music Player	780	100–500	26936	16	18	11	21.9	7.6	7.8	17.2	2
Olam Malay. Dictionary	651	100–500	31599	57	73	22	17.3	46.7	50.1	19.4	3
VuDroid	475.5	100–500	320	21	27	20	8.7	6.2	6.7	6.4	8
Slideshow	3700	10–50	68013	43	52	22	52.6	7.2	8.1	28.9	12

TABLE IV: AndroidSlicer evaluation: Fault localization.

App	Dex code size (KB)	Installs (thousands)	Instructions		CD+ DD	Call-back events	Time (seconds)				Over-head (%)
			Executed	In slice			Stage 1 Instrumentation	Original run	Stage 2 Instrumented run	Stage 3 Slicing	
SoundCloud	516.3	100,000–500,000	9590	128	173	62	41.7	63.5	71.6	32.7	9
Notepad	44.2	10,000–50,000	2366	15	17	6	4.5	36.5	41	9.1	12
A Comic Viewer	569.1	1,000–5,000	12679	18	24	13	26.7	52.6	61.3	18.7	16
AnkiDroid Flashcards	804.6	1,000–5,000	27164	32	38	27	87.6	17.3	19.5	28.7	12
APV PDF Viewer	52.9	1,000–5,000	24672	67	79	45	11	10.3	11.2	27.2	8
NPR News	285.1	1,000–5,000	45298	239	327	107	28.7	49.3	52.7	42.5	6
Document Viewer	3900	500–1000	5451	8	11	2	11.2	34	36.1	9	6

TABLE V: AndroidSlicer evaluation: Regression testing.

App	Dex code size V_1–V_2 (KB)	Installs	Test suite size	Covered methods (%)	Instructions		Reduced test suite
					V_1	V_2	
Mileage	443.8-471.3	500-1,000	200	66	28252	36531	22
Book Catalogue	444.9-445.4	100-500	200	69	28352	28450	7
Diary	125.5-129.8	100-500	200	53	4591	4842	47
Root Verifier	462.9-1700	100-500	200	58	23482	83170	5
Traccar Client	49.4-51.6	50-100	200	66	1833	1937	8

Table V shows the results: the bytecode sizes for V_1 and V_2, the number of installs, the coverage attained by T_1 on V_1, and the instructions executed when testing V_1 and V_2, respectively. The last column shows the size of T_2. Notice how our approach is very effective at reducing the test suite size from 200 test cases down to 5–47 test cases.

VIII. Related Work

Slicing event-based programs has been investigated for Web applications [6], [15], [16] written in HTML, PHP, and JavaScript. These approaches record execution traces through a browser plugin [6] and construct the UI model to generate the event nodes. While both Web and Android apps are event-based, their slicing approaches differ significantly. First, Android apps have a sophisticated life-cycle with multiple entry points that cause apps to run in different scopes (i.e., activity, app, system), and handle different sets of requests (launch another activity, respond to an action, or just pass data). In contrast, Web apps have different build phases, such as UI building phase (HTML nodes) and event-handling phase (JavaScript nodes). Second, Android app slicing demands low overhead due to time sensitivity of events, whereas a Web app slicing tool (e.g., as a browser plugin) does not require low-overhead. Third, Android's inter- and intra-app communication requires IPC tracking; that is not the case for Web apps.

Traditional program slicing of Java bytecode has only targeted single-entry sequential Java programs [17]–[20]. Zhou et al. [21] and Zeng et al. [22] have used bytecode slicing for Android apps, but to achieve entirely different goals: mining sensitive credentials inside the app and generating low-level equivalent C code. They create slices at bytecode level and consider only data dependences making them imprecise: there is no tracking of code dependences or accounting for many Android features (e.g., callbacks, IPC, input from sensors).

Compared to Agrawal and Horgan's slicing for traditional programs [23], we add support for Android's intricacies, node merging for control dependence edges, dealing with slicing in the presence of restarts as well as asynchronous callback invocation. We support loop folding for regular nodes inside the supernodes. Slicing multithreaded programs is tangentially related work, where slicing was used to debug multithreaded C programs [24]–[28] — this setup differs greatly from ours.

Dynamic race detectors (e.g., CAFA [29], ASYNC-CLOCK [30], Droidracer [31]) have focused on discovering and maintaining strict event order for Android apps. We think it would be possible to write a race detector on top of AndroidSlicer because we capture dependences that are involved in races. However, checking the correct order of asynchronous events (done by race detectors) is orthogonal to our work.

Hoffmann et. al. developed SAAF [32], a static slicing framework for Android apps. A static slicing framework such as SAAF would not be sufficient to achieve our goals as it does not consider myriad aspects, from late binding to the highly dynamic event order in real-world Android apps.

IX. Conclusions

We presented AndroidSlicer, a novel slicing approach and tool for Android that addresses challenges of event-based model and unique traits of the platform. Our asynchronous slicing approach that is precise yet low-overhead, overcomes the challenges. Experiments on real Android apps show that AndroidSlicer is effective and efficient. We evaluated three slicing applications that: reveal crashing program inputs, help locate faults, and reduce the regression test suite.

References

[1] B. Popper, "Google announces over 2 billion monthly active devices on android," https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users, accessed: August 23, 2018.

[2] B. Zhou, I. Neamtiu, and R. Gupta, "Experience report: How do bug characteristics differ across severity classes: A multi-platform study," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, Nov 2015, pp. 507–517.

[3] ——, "A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios," in *19th International Conference on Evaluation and Assessment in Software Engineering, EASE 2015*, April 2015, p. 10.

[4] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155–163, 1988.

[5] N. Sasirekha, A. E. Robert, and M. Hemalatha, "Program slicing techniques and its applications," *CoRR*, vol. abs/1108.1352, 2011. [Online]. Available: http://arxiv.org/abs/1108.1352

[6] J. Maras, J. Carlson, and I. Crnkovic, "Client-side web application slicing," in *ASE'11*.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[8] Android Developers, "App Components," 2017, https://developer.android.com/guide/components/index.html.

[9] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *ICSE '13*, 2013.

[10] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. ACM, 2015, pp. 349–366.

[11] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[12] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *Proceedings of the Conference on Software Maintenance*, ser. ICSM '93. Washington, DC, USA: IEEE Computer Society, 1993, pp. 348–357. [Online]. Available: http://dl.acm.org/citation.cfm?id=645542.658149

[13] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proceedings Conference on Software Maintenance 1992*, Nov 1992, pp. 299–308.

[14] Android Developers, "UI/Application Exerciser Monkey," November 2017, http://developer.android.com/tools/help/monkey.html.

[22] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS '13, 2013, pp. 487–498.

[15] P. Tonella and F. Ricca, "Web application slicing in presence of dynamic code generation," *Automated Software Engg.*, pp. 259–288.

[16] F. Ricca and P. Tonella, "Construction of the system dependence graph for web application slicing," in *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, ser. SCAM '02, 2002, pp. 123–.

[17] Wang, T. and Roychoudhury, A., "Using compressed bytecode traces for slicing java programs," 2004.

[18] T. Wang and A. Roychoudhury, "Dynamic slicing on java bytecode traces," *ACM Trans. Program. Lang. Syst.*, pp. 10:1–10:49, 2008.

[19] "jslice," November 2017, http://jslice.sourceforge.net/.

[20] A. Szegedi and T. Gyimothy, "Dynamic slicing of java bytecode programs," *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 35–44, 2005.

[21] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15, 2015, pp. 23:1–23:12.

[23] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90, 1990, pp. 246–256.

[24] X. Zhang, S. Tallam, and R. Gupta, "Dynamic slicing long running programs through execution fast forwarding," ser. SIGSOFT '06/FSE-14, 2006, pp. 81–91.

[25] S. Tallam, C. Tian, R. Gupta, and X. Zhang, "Enabling tracing of long-running multithreaded programs via dynamic execution reduction," ser. ISSTA '07, 2007, pp. 207–218.

[26] S. Tallam, C. Tian, and R. Gupta, "Dynamic slicing of multithreaded programs for race detection," in *ICSM'08*, 2008, pp. 97–106.

[27] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, "Analyzing concurrency bugs using dual slicing," ser. ISSTA'10, 2010, pp. 253–264.

[28] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu, "Drdebug: Deterministic replay based cyclic debugging with dynamic slicing," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 98:98–98:108. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544152

[29] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," *SIGPLAN Not.*, pp. 326–336, 2014.

[30] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "Asyncclock: Scalable inference of asynchronous event causality," *SIGARCH Comput. Archit. News*, pp. 193–205, Apr. 2017.

[31] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 316–325. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594311

[32] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13, New York, NY, USA, 2013, pp. 1844–1851.