Appears in the Proceedings of the 2021 IEEE International Conference on Software Testing, Verification, and Validation (ICST 2021)

MANDOLINE: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis

Khaled Ahmed
Univ. of British Columbia, Canada
khaledea@ece.ubc.ca

Mieszko Lis
Univ. of British Columbia, Canada
mieszko@ece.ubc.ca

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

***Abstract*—Dynamic program slicing is used in a variety of tasks, including program debugging and security analysis. Building an efficient and effective dynamic slicing tool is a challenging task, especially in an Android environment, where programs are event-driven, asynchronous, and interleave code written by a developer with the code of the underlying Android platform. The user-facing nature of Android applications further complicates matters as the slicing solution has to maintain a low overhead to avoid substantial application slowdown.**

In this paper, we propose an accurate and efficient dynamic slicing technique for Android applications and implement it in a tool named MANDOLINE. The core idea behind our technique is to use minimal, low-overhead instrumentation followed by sophisticated, on-demand execution trace analysis for constructing a dynamic slice. We also contribute a benchmark suite of Android applications with manually constructed dynamic slices that use a faulty line of code as a slicing criterion. We evaluate MANDOLINE on that benchmark suite and show that it is substantially more accurate and efficient than the state-of-the-art dynamic slicing technique named ANDROIDSLICER.

I. INTRODUCTION

Program slicing [1] computes the set of statements that affect a particular variable or statement of interest, often referred to as a *slicing criterion*. Slicing techniques are used in a variety of tasks, e.g., program debugging, to help locate the origin of an error more easily. Slicing can be performed either statically or dynamically [2]. While static slicing considers all possible program paths leading to the slicing criterion, dynamic slicing focuses on one concrete execution.

The main idea behind a dynamic slicing tool is to first collect an *execution trace* of a program, i.e., the set of all executed program statements, by instrumenting either the program itself or its underlying runtime environment. Then, the tool inspects control and data dependencies of the trace statements, identifying statements that affect the slicing criterion and omitting the rest. The produced dynamic slices are more compact than static ones, making them particularly suitable for debugging activities [3], [4].

Dynamic slicing techniques have been extensively studied in the literature, mostly for traditional, desktop/server applications, e.g., [5], [6], [7], [8]. Slicing mobile programs (often referred to as mobile apps) is a more challenging task. First, in mobile systems, instrumentation typically targets the app itself rather than the underlying mobile platform: the implementation of the platform changes rapidly, making modifications obsolete within less than a year [9]. Producing app-level instrumentation is challenging because the instrumentation has to balance low runtime overhead (mobile apps are user-facing and mobile platforms simply kill slow apps [10]) with the ability to collect sufficient information for performing accurate slicing. Furthermore, mobile apps rely heavily on the code of the underlying platform and thus a slicing solution has to consider control and data flowing through the framework. Finally, it has to consider numerous interleaved threads of the app, as mobile apps are mostly asynchronous and event-driven.

In this paper, we focus on addressing these challenges, contributing an efficient and effective solution for slicing Android mobile apps. Android slicing was already attempted before, in a tool called ANDROIDSLICER [11]. Yet, ANDROIDSLICER makes several decisions that sacrifice accuracy for low instrumentation overhead, e.g., it does not track data propagation via fields of objects used in different methods.

Consider, for example, the Planner app in Figure 1, which schedules calendar events for the user. The code of the app was borrowed from the `com.alexstyl.specialdates` app in our evaluation dataset and was simplified for illustration purposes. For simplicity, we also discuss the example and the slicing algorithm at the source-code level while our solution is able to process apps and their third-party libraries at the byte-code level, even when no source code is available.

In our example app, the method `onCreateDialog()` of the `PlanDialog` class (lines 4-15) is called by the platform when the user opens a dialog to pick a day for a recurrent event. This method first creates an instance of the `EditText` UI widget, where the user can enter the name of the event (line 5), and an instance of the `Picker` widget, where the user can select a day from a predefined range of numbers between 1 and 31 (lines 6-8). The method then associates the `Picker` object with a listener, which will be triggered when the user picks a day (lines 9-10). Finally, it creates the “Set” button and associates it with another listener, which will be triggered when the button is pressed (lines 11-14).

Both listeners are implemented as `PlanDialog`’s inner classes (lines 16-24 and lines 25-40, respectively), following the recommendation in the Android tutorial [12]. Such an implementation provides both inner classes access to the fields of its enclosing class, e.g., the field `d` in line 2. Within an inner class, this field is accessed via `PlanDialog.this.d` expression (lines 22 and 31). In our example, the constructors of the inner classes (lines 18-20 and lines 27-29, respectively) are

```
 1 class PlanDialog extends DialogFragment {
 2   int d = 1;
 3   EditText eventNameText;
 4   void onCreateDialog() {
 5     eventNameText = new EditText(...);
 6     Picker picker = new Picker(...);
 7     picker.setMinValue(1);
 8     picker.setMaxValue(31);
 9     PListener pListener = new PListener(this);
10     picker.setOnValueChangedListener(pListener);
11     Button button = new Button(...);
12     button.setText("Set");
13     CListener clistener = new CListener(this);
14     button.setListener(cListener);
15   }
16   class PListener extends OnValueChangeListener {
17     PlanDialog this$0;
18     PListener(PlanDialog planDialog){
19       this.this$0 = planDialog;
20     }
21     void onValueChange(Picker picker) {
22       PlanDialog.this.d = picker.getValue();
23     }
24   }
25   class CListener extends OnClickListener {
26     PlanDialog this$0;
27     CListener(PlanDialog planDialog){
28       this.this$0 = planDialog;
29     }
30     void onClick() {
31       int day = PlanDialog.this.d;
32       int year = Year.now().getValue();
33       int month = 0;
34       while (month < 12) {
35         Date date = new Date(day, month, year);
36         //... schedule the event in the calendar
37         month++;
38       }
39     }
40   }
41 }
```

Figure 1: Planner: a faulty app.

generated by the compiler automatically and are not written by the developer. We include the constructors in our example for completeness of the representation.

When the user selects a day in the `Picker` widget, the `onValueChange(...)` method of `PListener` is triggered (lines 21-23), storing the selected day in the `PlanDialog`'s field `d` (line 22). Then, when the user presses the "Set" button, the `onClick()` method of `CListener` is triggered (lines 30-39). This method retrieves the day selected by the user from field `d` (line 31), computes the current year (line 32), and iterates over all months of the year, placing the event in the user calendar for the selected date of each month (lines 33-37).

The app fails when the user selects a day that does not exist in a particular month, e.g., 31 in February, which leads to the `IllegalFieldValueException` exception when creating the `Date` object in the second loop iteration (line 35). A slice from the faulty line can help narrow down the program execution only to code relevant for the failure, e.g., omitting the code dealing with event name (lines 3 and 5) and its scheduling in the calendar (line 36).

To accurately identify the relevant program slice for the statement in line 35 as the slicing criteria, one needs to compute the data dependencies of the variable day used in line 35. This variable is defined as `PlanDialog.this.d` in line 31, which, in fact, is aliasing the field `d` of the `PlanDialog` class. However, the value of this field is not set in line 2 but rather in line 22, by the `onChangeValue(...)` method of `PListener`.

Tracking such data dependencies dynamically requires an accurate analysis of object references, to establish that `PlanDialog.this.d` in both line 31 and 22 refer to the same object. A straightforward solution to this problem is to collect, via app instrumentation, memory addresses of all objects manipulated by the app, making it possible to uniquely identify and match variables referring to the same object [4], [13], [5], [14], [8]. The main drawback of this solution is its high runtime overhead because practically every line of code must be instrumented in order to record addresses of all used objects. We conjecture that for this reason, ANDROIDSLICER does not track such dependencies and thus cannot build the complete slice for the app in Figure 1, missing the assignment in line 22. Our work addresses this limitation, contributing a method for identifying variables referencing the same object via *alias analysis* [15], [16], [17], [18] conducted over the execution trace of the app, after the app execution terminates. Our method enables tracking data propagation via object fields without increasing the instrumentation cost.

Moreover, continuing slicing from the statement in line 22 requires tracing field definitions and usages inside Android framework methods, to establish that there is a data dependency between the `picker.getValue()` and `picker.setMaxValue(31)` statements in lines 22 and 8, respectively. This is because `picker.getValue()` uses an internal array structure that holds all possible `Picker` values shown to the user; it retrieves the selected value from that array while `picker.setMaxValue(31)` sets the boundaries of this array. Such dependencies implemented inside the framework pose challenges to existing slicing solutions: ANDROIDSLICER misses the statement in line 8, which, in fact, is the statement that causes the crash. Inspired by existing approaches for Android static analysis [19], our work addresses this limitation by modeling data propagation inside the framework methods. Specifically, our solution captures a set of defined and used variables for each framework method and uses it to augment "classical" data dependencies between Java statements.

We implement our proposed slicing approach in a tool named MANDOLINE (as the cooking utensil used for slicing vegetables rather than the musical instrument with a similar name). To evaluate MANDOLINE and compare it with the state-of-the-art Android slicing techniques, we built a benchmark suite containing 12 Android apps with known faults and their corresponding dynamic slices. We borrowed the apps from the existing benchmark suites containing faulty apps: ReCDroid [20] and DroixBench [21]. Then, two members of our research group independently analyzed each app, building the expected slice manually and cross-validating each other's results to ensure correctness; the overall effort of building the ground truth for slicing took more than 30 workdays. Using the produced benchmarks, we evaluate MANDOLINE and compared it with ANDROIDSLICER, showing that MANDOLINE outperforms the state-of-the-art Android slicing techniques w.r.t. both accuracy and runtime overhead due to its efficient instrumentation, dynamic field analysis, and Android modeling approaches. Our work is the first to assess the quality of the produced Android slice against a fixed ground truth rather than only calculating the code reduction rate achieved by slicing.

Contributions. The main contributions of the paper are: (1) An efficient dynamic slicing approach, implemented in a tool called MANDOLINE. Our approach does not require expensive app instrumentation and relies on efficient field alias analysis, modeling of Android framework methods, and modeling of threads and asynchronous Android callbacks. (2) The first benchmark suite containing manually produced slices for Android applications. (3) An evaluation of MANDOLINE on the proposed benchmark, which compares its accuracy and runtime efficiency with the state-of-the-art approach in Android slicing: ANDROIDSLICER. Our implementation of MANDOLINE and the experimental data is available online [22].

II. BACKGROUND AND NOTATIONS

We now introduce Android apps structure, notations used in this paper, and define slicing for Java programs.

A. Application Structure

Unlike a desktop program, an Android app does not have a single `main` method. Instead, an app defines multiple entry points, referred to as *callbacks*, which are triggered by the Android platform to react to user-generated and sensor-generated events, such as button clicks and location changes. For example, the method `onClick()` in line 30 of Figure 1 is a callback triggered when the user clicks the "Set" button; the callback is associated with the button in line 14, using the `setListener(...)` method. Android callbacks whose timing and order of execution is preset by the platform are called *lifecycle events*. For example, the method `onCreateDialog()` in line 4 of Figure 1 is a lifecycle event triggered by the platform when initializing the dialog.

Android relies on Java *multi-threading* mechanisms and extends them with additional, framework-specific mechanisms. For example, the `AsyncTask` class provides the method `doInBackground(...)`, which makes it possible to perform long-lasting operations in a background thread. Apps can also start asynchronous executions in response to a timer or to message events triggered by the platform. For example, the `Timer` class provides the method `schedule(...)`, allowing the application to schedule a thread to run at specific time intervals.

Finally, we refer to the APIs exposed to the application by the underlying Android platform as *framework methods.* For example, the methods `setMinValue(...)` and `setMaxValue(...)` of the class `Picker` (lines 7-8 in Figure 1) allow the developer to constraint the range of numbers presented to the user by this widget.

B. Control and Data Dependencies

In static program analysis, each node of a *Control-Flow Graph (CFG)* represents a statement; an edge between the nodes represents a potential flow of control between these statements [23]. For the example in Figure 1, the statement in line 5 is control-flow dependent on the statement in line 4 as control flows from line 4 to 5. For simplicity, we refer to the statement in line i of this example as s_i, i.e., saying that s_5 is control-flow dependent on s_4.

In dynamic analysis, each statement of an app can be triggered multiple times during the app execution, e.g., in multiple iterations of a loop or in different instances of a thread. We refer to each individual execution of a statement as a *statement instance* and denote the j^{th} execution of a statement s_i as t_i^j. We refer to the sequence of statement instances executed in a particular app run as an *execution trace*. As s_{35} was executed twice in the faulty scenario (for January and February), it has two statement instances: t_{35}^1 and t_{35}^2.

Unlike static control-flow dependencies, dynamic control-flow dependencies represent a concrete transfer of control recorded during the app execution. We say that a statement instance t_k^m is control-flow-dependent on t_i^j in a *Dynamic Control-Flow Graph (DCFG)* if t_k^m is executed immediately after t_i^j in the same execution thread and one of the following holds: (i) both statements are in the same method and s_k is control-flow-dependent on s_i in a static control-flow graph or (ii) the statements are in different methods and s_i triggers the method whose first statement is s_k.

A statement instance t_k^m is *control-dependent* on t_i^j if and only if s_i can alter the program's control and it determines whether s_k executes [24]. Examples of statements that can alter the control are `if` and `while`. In Figure 1, t_{35}^2 is control-dependent on t_{34}^2, as s_{34} is a `while` condition whose outcome affects whether the control reaches s_{35} or not.

We say that a statement instance t_i^j is a *dynamic reaching definition* of a variable v in t_k^m if and only if (a) t_i^j is control-flow-reachable from t_k^m, (b) there exist a variable v s.t. v is used in s_k and defined in s_i, and (c) there is no redefinition of v along the control-flow edges between t_i^j and t_k^m in the DCFG. In that case, we also say that statement instance t_k^m is *data-flow-dependent* on t_i^j w.r.t. the variable v [4]. For example, the dynamic reaching definition of the variable `day` in t_{35}^2 is t_{31}^1 and, thus, t_{35}^2 is data-flow-dependent on t_{31}^1 w.r.t. `day`.

C. Slicing

A *slicing criterion* for an execution trace is a tuple (c, V), where c is a statement instance and V is a set of all variables of interest used in this statement instance [2]. If V is omitted, it is assumed to include all variables used by c. For the example in Figure 1, the app crashes with `IllegalFieldValueException` in the second iteration of the loop in line 35, making t_{35}^2 with all its used variables a suitable slicing criterion for the developer interested in finding the cause of the crash.

A *backward dynamic slice* [2] is the set of statement instances whose execution affects the slicing criterion, i.e., the set instances on which the slicing criterion is control- or data-flow-dependent, either directly or transitively. For the example in Figure 1, the slice from t_{35}^2 contains t_{35}^2, t_{34}^2, t_{37}^1, t_{35}^1, t_{34}^1, t_{33}^1, t_{32}^1, t_{31}^1, t_{22}^1, t_8^1, t_7^1, and t_6^1.

Algorithm 1 outlines the classic backward slicing algorithm by Agrawal et al. [4]. The algorithm uses a working set W that holds pairs (t, v) of statement instance and the variable used for slicing. It initializes W with the slicing criteria (line

Algorithm 1: Dynamic slicing algorithm.

```
1  Input  : DCFG, c, $v_1$, ..., $v_n$
   Output: slice
2  begin
       ▷ Create working set of pairs of statement instances and used variables
3      W ← {(c, $v_1$) ... (c, $v_n$)}
4      while $W \neq \emptyset$ do
5          ($t$, $v$) ← pick and remove from W
6          slice ← slice ∪ {$t$}
7          $t'$ ← the instance $t$ is control-dependent on
8          if $t'$ not in slice then
               ▷ Ensure not processing statements more than once
9              W ← W ∪ {($t'$, $v'$) | $v'$ ∈ use($t'$)}
                   ▷ Add $t'$ to working set with all its used variables
10         $t'$ ← reaching definition of $v$ at $t$
11         if $t'$ not in slice then
12             W ← W ∪ {($t'$, $v'$) | $v'$ ∈ use($t$)}
13     return slice
14 Procedure use($t$)
15     return all variables used in t
```

3 in Algorithm 1) and then iterates over W (lines 4-12) (i) transitively adding all control and data-flow dependencies of each pair $(t, v) \in W$ to W (lines 7-12) and (ii) adding statement instances from each processed pair to the produced slice (line 6). The algorithm terminates when W is empty, i.e., it cannot discover new control and data-flow dependencies.

The main challenge of slicing is thus efficiently calculating data dependencies, as we discuss in Section IV.

III. Inter-Callback Dependency Graph (ICDG)

We represent the execution of an application as an *Inter-Callback Dependency Graph (ICDG)*, defined below. Figure 2 shows an example of such graph for the Planner app in Figure 1. The nodes of the graph are statement instances. It has four types of edges: control-flow and control-dependence edges correspond to the definitions in Section II. In Figure 2, control-flow edges are depicted with dotted lines, e.g., between t_5^1 and t_4^1; control dependencies are depicted with dashed lines, e.g., between t_{35}^2 and t_{34}^2. We only capture control and control-flow edges within a single callback/thread, e.g., `onCreateDialog`, `onValueChange`, or `onClick`. That is because callbacks/threads are triggered by the Android framework rather than call each other. Thus, there is no incoming control-flow edge for the nodes t_{22}^1 and t_{31}^1.

We depict data-flow edges with solid lines and annotate each line with the name of the corresponding variable, e.g., t_{35}^2 is data-flow-dependent on t_{31}^1, with the variable `day`. We extend the definition of data-flow edges to consider framework methods, e.g., t_{22}^1 and t_8^1. This is done to capture the data flows within the method and their effect on the method externally-visible variables: method parameters and their fields, fields of `this` reference and their sub-fields, and static fields and their sub-fields. Specifically, we model each framework method as a set of assignments with exactly one right-hand-side and one left-hand-side variable, denoted by r and l, respectively. r is an externally-visible variable modified in the method or the method return value; l is the variable affecting the value of r. For example, method t_{22}^1 is represented by a single assignment `return = this.values` because this method outputs the value selected from the picker internal `values` array. In our model, one variable can appear in more than one assignment, on both left- and right-hand sides (as a variable can affect and be affected by more than one other variable) and there is no significance to the order of assignments in the model.

We now extend the notion of data-flow to consider these sets of assignments: given two statement instances t and t', we say that t' is data-flow dependent on t w.r.t. the variable v if and only if t contains at least one dynamic reaching definition of a variable v in any of the expressions in t'. We build up on the taint analysis model provided by StubDroid [19] to approximate the set assignments for a framework method, as discussed in Section IV.

Finally, Android callbacks can specify which other callbacks respond to system/UI events. In Figure 1, `onCreateDialog()` uses `Button::setListener(...)` method in line 14 to associate the listener with the button, effectively specifying which method is called when the button is pressed: `onClick()` in

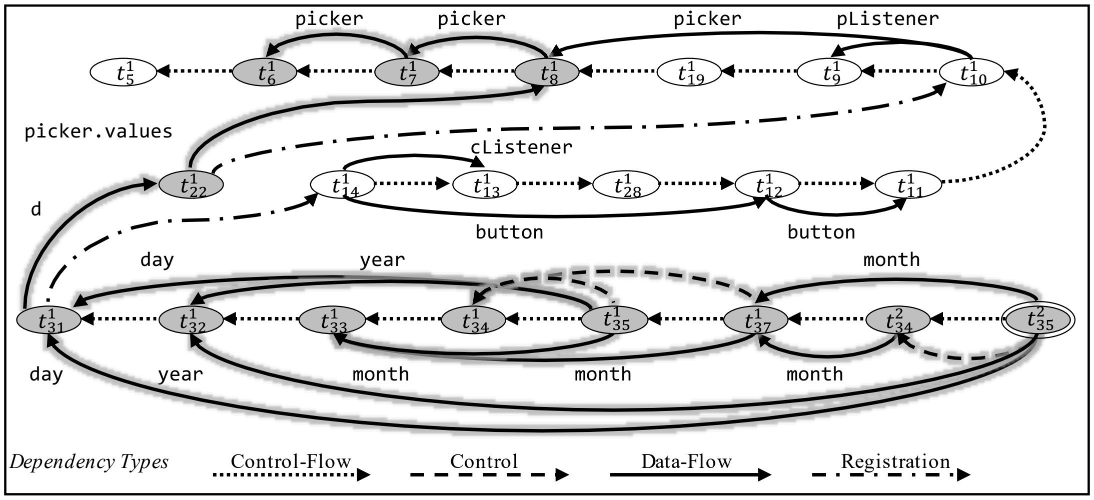

Figure 2: Inter-Callback Dependency Graph (ICDG) and the dynamic slice for the Planner app in Figure 1.

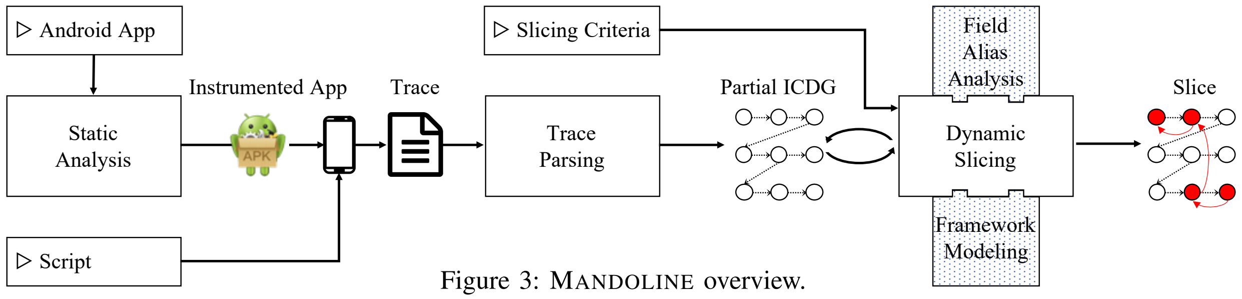

Figure 3: MANDOLINE overview.

line 30. In that case, we say that one callback *registers* another and depict the *registration dependency* between callbacks with dashed-dotted lines, e.g., t^1_{31} is registration-dependent on t^1_{14}. We utilize registration dependencies to track the flow of data between the callbacks and apply similar treatment to define registration dependencies between threads.

We say that $(t, t') \in$ ICDG if there is at least one dependency from t' to t (i.e., t' happens at a later time). We discuss our slicing approach and our approach to building the ICDG next.

IV. Slicing with Trace-Based Alias Analysis

A. Main Workflow

Figure 3 gives a high-level overview of MANDOLINE. Our tool receives as input an Android app, an execution script that triggers the fault in the app, and a slicing criterion, e.g., a statement that throws an exception. It produces as output a backward dynamic slice for the provided slicing criterion.

As the first step, MANDOLINE runs a lightweight static analysis to instrument the app and to calculate the control and control-flow dependencies for each statement. Our analysis and instrumentation relies on Soot [25] and works on its Jimple intermediate code representation. We follow the basic-block-level instrumentation by Kamkar et al. [26], which records a unique identifier of each execution thread (using `Thread.getId()`) and for each basic-block within the thread.

The instrumented app is then installed on the device and the tool runs the pre-recorded execution script (alternatively, the user can trigger the app manually to reproduce its faulty behavior). The output of this process is the execution trace, i.e., a sequence of the executed basic blocks. MANDOLINE parses the trace and builds a partial ICDG whose nodes are statement instances for each executed basic-block, as in the example in Figure 2. It then separates the nodes by their thread identifiers and adds control-flow edges between statement instances with the same thread identifier, in their order of execution.

To add registration dependencies, MANDOLINE relies on the callback identification logic from FlowDroid [27]. Specifically, it uses callback classes from FlowDroid and identifies statements that register these callbacks, e.g., line 10 in Figure 1. It then extracts the type of the object used for callback registration, e.g., class `PListener`, and uses this type to match the registration site with (possibly multiple) registered callbacks, e.g., `onValueChanged(...)` callback in line 21 of class `PListener`. We add a registration dependency edge for each such registration, e.g., the edge from t^1_{22} to t^1_{10} in Figure 2. We apply similar analysis, also borrowed from FlowDroid, for the thread identification logic, adding a registration dependency edge between the statement that creates the thread and the first statement instance of its corresponding new thread.

The produced partial ICDG, which now contains control-flow and registration edges, together with the user-defined slicing criterion, is used as the input to the dynamic slicing step. We follow the backward slicing algorithm by Agrawal et al. [4] described in Section II and outlined in Algorithm 1. The main objective of the algorithm is to identify control dependencies (line 7) and reaching definitions (line 10).

For identifying the control dependency of a statement instance t^m_k, we find its corresponding statement s_k in the static control dependency graph produced by Soot. We then identify the statement s_i that s_k is control-dependent on and traverse the ICDG to find the first statement instance of s_i, t^j_i, creating the corresponding dynamic control relationship between t^m_k and t^j_i. For example, t^2_{35} is control-dependent on t^2_{34} because s_{35} is control-dependent on the `while` condition in s_{34}.

Identifying reaching definitions aims to find statement instances t^j_i that define a variable v used at statement instance t^m_k. We distinguish between three types of v: local variables, fields, and framework methods. For local variables, we simply follow the work of Agrawal et al. [4]: we traverse the ICDG backward and stop at the first encountered statement instance that defines v. For example, the definition of the local variable `day` used at t^2_{35} is t^1_{31}. The treatment of fields and framework methods is the main contribution of our approach, which we describe in the following two sub-sections.

After control dependencies and reaching definitions are identified, these statements, together with all the variables that they use are added to the working set W (lines 9, 2) and the slicing process continues until no further relevant statements can be added. MANDOLINE then returns the produced slice: a set of statement instances that affect the slicing criterion through transitive control and data dependencies.

B. Data Flows Through Fields

For a field variable v in statement instance t, we use field alias analysis to find its *alias statement set* S: all statement instances that define variables aliasing v. We then find definitions of all variables in S and, from those, pick the last one that reaches t (without redefinitions). Unlike static approaches, we perform the alias analysis directly on the execution trace.

Our analysis starts from the statement instance t that uses v and goes backward, in the inverse execution order, to find each statement instance t' that defines a variable v' as an alias to v. Then, from each identified statement t', it spawns a forward analysis to find other possible variables that transitively alias v between t' and t. For the example in Figure 2, the backward analysis starts from t^2_{35}, looking for the definition of the local variable `day`. This variable is defined in t^1_{31} as the field `PlanDialog.this.d`. The name of the field is, in fact, "syntactic sugar" translated by the compiler to the variable `this.this$0.d`. By traversing the control-flow and registration dependencies, the analysis finds aliases to this field in t^1_{28} and t^1_{19}. Spawning the forward analysis from each of these two instances further finds an alias in t^1_{22} (via a registration dependency from t^1_{22} and t^1_{10} followed by two control-flow dependencies to t^1_9 and t^1_{19}). Thus, the alias statement set S for `PlanDialog.this.d` in this example is t^1_{28}, t^1_{22}, and t^1_{19}. According to the trace order, t^1_{22} is the last out of those three statements that was executed before t^1_{31}; thus, it is selected as a definition of `this.this$0.d`. The main slicing loop (Algorithm 1) then continues from t^1_{22}.

Algorithm 2: Alias analysis.

```
1   Input: ICDG, $t$, $v$
    Output: $T'$
2   begin
3       $V \leftarrow \{v\}$                                   ▹ Aliases of the variable $v$
4       $S \leftarrow$ BackwardAnalysis(ICDG, $V$, $t$)          ▹ Alias statements
5       return LastDefined($S$, $t$)                 ▹ Definitions of $v$ and its fields
6   Procedure BackwardAnalysis(ICDG, $V$, $t$)
7       begin
8           $S \leftarrow \emptyset$
            ▹ Traverse the ICDG in inverse execution order
9           foreach $t'$ s.t. $(t', t) \in ICDG$ do
10              if $method(t) \neq method(t')$ then
11                  $V \leftarrow$ ChangeScope($V$, $method(t)$, $method(t')$)
12              if $\exists v \in V$ s.t. $v$ and LHS($t'$) have a common prefix then
                    ▹ $t'$ is a definition of a variable in $V$
13                  $S \leftarrow S \cup \{t'\}$               ▹ Add the definition statement
14              $<V_b,V_f> \leftarrow$ AliasAnalysis($\uparrow$, $V$, $t'$)
15              $S \leftarrow S \cup$ BackwardAnalysis(ICDG, $V_b$, $t'$)
16              $S \leftarrow S \cup$ ForwardAnalysis(ICDG, $V_f$, $t'$)
17          return $S$
18  Procedure ForwardAnalysis(ICDG, $V$, $t$)
19      begin
20          $S \leftarrow \emptyset$
            ▹ Traverse the ICDG in execution order
21          foreach $t'$ s.t. $(t, t') \in ICDG$ do
22              if $method(t) \neq method(t')$ then
23                  $V \leftarrow$ ChangeScope($V$, $method(t)$, $method(t')$)
24              if $\exists v \in V$ s.t. $v$ is a prefix of LHS($t'$) then
                    ▹ $t'$ is a definition of a field of a variable in $V$
25                  $S \leftarrow S \cup \{t'\}$               ▹ Add the definition statement
26              $<V_b,V_f> \leftarrow$ AliasAnalysis($\downarrow$, $V$, $t'$)
27              $S \leftarrow S \cup$ BackwardAnalysis(ICDG, $V_b$, $t'$)
28              $S \leftarrow S \cup$ ForwardAnalysis(ICDG, $V_f$, $t'$)
29          return $S$
30  Procedure AliasAnalysis($d$, $V$, $t'$)
        ▹ Initialize with the original set of variables
31      if $d = \uparrow$ then
32          $V_b \leftarrow V$; $V_f \leftarrow \emptyset$
33      else if $d = \downarrow$ then
34          $V_b \leftarrow \emptyset$; $V_f \leftarrow V$
35      foreach $v \in V$ s.t. $v$ and RHS($t'$) have a common prefix do
            ▹ LHS($t'$) is a new alias for $v$
36          $V_f \leftarrow V_f \cup$ ExtendFields(LHS($t'$))        ▹ Follow it forward
37      foreach $v \in V$ s.t. $v$ is a prefix of LHS($t'$) do
            ▹ $t'$ is a re-definition of a field of $v$
            ▹ Follow the assigned variable both backward and forward
38          $V_b \leftarrow V_b \cup$ RHS($t'$)
39          $V_f \leftarrow V_f \cup$ RHS($t'$)
40      foreach $v \in V$ s.t. LHS($t'$) is a prefix or equal to $v$ do
            ▹ $t'$ is a full re-definition of $v$
41          if $d = \uparrow$ then
42              $V_b \leftarrow V_b \setminus \{v\}$        ▹ Do not search before the definition
                ▹ Follow the assigned variable both backward and forward
43              $V_b \leftarrow V_b \cup$ ExtendFields(RHS($t'$))
44              $V_f \leftarrow V_f \cup$ ExtendFields(RHS($t'$))
45          else if $d = \downarrow$ then
46              $V_f \leftarrow V_f \setminus \{v\}$           ▹ Do not search for new variables
47      return $<V_b,V_f>$
```

Algorithm 2 gives a more formal definition of the alias analysis approach described above. The algorithm accepts as input the ICDG, statement instance t, and variable v (initialized separately for each variable in the slicing criterion). It outputs a set of statement instances T' containing the definition of v and, possibly, also the definitions of its individual fields, if such definitions exist in the trace. The set V contains all found aliases of v and is initialized with v itself (line 3). The algorithm then traverses the control-flow and registration dependencies in the ICDG, looking for additional aliases of v and their corresponding definition statement instances, which are collected in S (line 4). Once S is built, the algorithm identifies the latest statement instance in S (according to the trace order) that defines v and each of its fields; it outputs this set of definitions (line 5).

The algorithm starts from the backward traversal (lines 6-17), processing one statement instance at a time and spawning additional backward and forward steps. When inspecting the next statement t' in the ICDG (line 9), the algorithm first checks if t and t' are in the same method (line 10). If not, the `ChangeScope` operator translates the namespace in V (line 11) to the new method, accounting for local variables that reference same objects due to parameter passing and returns. For example, `this` in t^1_{31} is translated to its corresponding variable `cListener` when crossing the method boundaries to t^1_{14}.

Then, the algorithm checks whether the inspected statement t' is an assignment and whether its *defined* variable (left-hand side of t', which we denote by LHS(t')) overlaps with a variable that is already in V (line 12). If so, t' is a definition of this variable in V and is added to the set of definitions S (line 13). The algorithm further examines t' to decide on the set of variables for the next steps in backward and forward analysis (line 14). To do so, it uses the helper function `AliasAnalysis` (lines 30-47), which obtains three parameters: the current direction of the analysis, the set V, and the statement instance t'. It returns a tuple $<V_b,V_f>$, where V_b defines a set of variables for the next step of the backward analysis (line 15) and V_f defines a set of variables for the forward analysis (line 16). These subsequent analysis steps further traverse the graph, collecting more definitions of v that are stored in S (lines 15-16). After these steps terminate, the backward analysis returns the collected set of definitions (line 17).

The forward analysis is similar (lines 18-29), except that it only adds definitions of the fields in v (lines 24-25), as a definition of the entire variable from v when processing the trace forward would kill the original variable of interest.

To decide how to build V_b and V_f, `AliasAnalysis` starts from the assumption that backward analysis (denoted by $\uparrow$) should process backwards with the original set of variables, i.e., V_b is initialized to V, and with no elements for forwards analysis, i.e., V_f is empty (line 32). Likewise, forward analysis (denoted by $\downarrow$) assumes proceeding forward with the original set V and with an empty backward set by default (line 34).

The algorithm then considers three possible types of assignment in t'. First, t' could *specify a new alias* for a variable $v \in V$, e.g., if V includes a variable $x.y.z$ and t' is `a.b = x.y` (lines 35-38). The algorithm then needs to search for the new alias (i.e., the left-hand side of the assignment) forward, as the backward analysis could have missed definitions of this newly discovered field that occurred before the slicing criteria, e.g., `a.b.z = w`. We thus add the corresponding left-hand-side variable to V_f, extending its fields to match those of v (line 36), if needed. I.e., we extend $a.b$ to $a.b.z$, as that is an alias to the original variable $x.y.z$.

If the variable defined in t' is a field of a variable $v \in V$, e.g., if V includes a variable $x.y.z$ and t' is `x.y.z.w = a.b`, we say that t' *redefines a field* of v (lines 37-38). In such cases, the algorithm searches for the assigned variable, e.g., $a.b$, both backwards and forward, to find all possible definitions that occurred before the slicing criteria.

Finally, if the variable defined in t' is a *full redefinition* of a variable $v \in V$, e.g., if V includes a variable $x.y.z$ and t' is `x.y = a.b` or `x.y.z = a.b`, we distinguish between the backward and forward searches (lines 40-46). For the backward search ($\uparrow$), we remove the redefined variable v from subsequent backward search (line 42) because the definition in t' kills all prior definitions of the variable with the same name. Instead, we search for the assigned variable, e.g., $a.b$, both backwards and forward, like in the previous case (lines 43-44). In the forward search ($\uparrow$), we simply remove the variable defined in t' from further analysis (line 46), as t' redefined the original variable of interest with the same name (if it existed) and this new definition is not relevant to our search.

Our implementation also provides several optimizations improving the efficiency of this conceptual algorithm, e.g., we do not traverse the graph past the slicing criteria and add additional handling for registrations dependencies dealing with callbacks and threads. Details are available online [22].

C. Data Flows Through Framework Methods

We extend our treatment of data dependencies by modeling Android framework methods as a set of assignments that capture the aliasing effect of a method, as described in Section III. When traversing the ICDG using Algorithm 2, we treat a framework method as an unordered set of assignments, processing each independently of others using `AliasAnalysis`.

To produce the framework model, we rely on StubDroid [19] – a tool originally designed to produce accurate framework method summaries for the taint-analysis problem. Specifically, StubDroid defines how a taint originated from each method inputs (i.e., fields and parameters) propagates to its output (i.e., fields, parameters, and return value). We recast the taint-flow summaries of StubDroid as assignment statements which our analysis can use to propagate alias information, converting each taint-flow to an assignment. For example, the StubDroid summary for the method `Picker::setMaxValue(int)` will state that a taint from its first parameter `P1` reaches the field `this.max`. We use this information to represent a method as an assignment `this.max = P1`, expressing the fact that the value of this parameter is assigned to the `Picker`'s field `max` (directly or indirectly).

To deal with the situation that tainted parameters and fields might change within the method (without causing taint propagation), we augment all summaries with self-assignments for these objects and their fields. E.g., for the method call `X = Y.foo(Z)`, we add `Z.* = Z.*` and `Y.* = Y.*` to the summaries, where `.*` represents an object or any of its sub-fields.

StubDroid summaries exclude some methods, i.e., those containing native code. In such cases, we use FlowDroid's original taint wrappers [27], which provide a coarser-grained model of framework methods. Taint wrappers divide the framework methods into four types: *generation*, *exclude*, *kill*, and *default*. We use the statement `X = Y.foo(Z)` to explain the four strategies.

The generation strategy assumes that taints always flow from all arguments (`Z`) and the receiver object (`Y`) to the receiver and the return value (`X`), as well as all their fields (`Y.*` and `X.*`). Thus, there are four assignments, `Y* = Z.*`, `Y.* = Y.*`, `X.* = Z.*`, and `X.* = Y.*`. We also include the identity assignment to approximate the situation that tainted parameters change within the method: `Z.* = Z.*` In the exclude strategy, taint only propagates from the tainted variable to itself, which is equivalent to the identity assignment `Y.* = Y.*` and `Z.* = Z.*`. The kill strategy removes all taints: none of the variables is tainted after the method is called. Finally, the default strategy assumes that all taints propagate from the parameters to the return variable and its fields (but not to the receiver object): `X.* = Y.*`, `X.* = Z.*`, `Z.* = Z.*`.

The list of framework methods summaries that we used is available online [22]. While these summaries might over-approximate the set of possible behaviors, they make it possible for MANDOLINE to deal with the flow of data within the framework methods.

V. EVALUATION

We now describe our experimental setup and discuss the evaluation results. To evaluate MANDOLINE, we answer the following research questions:

RQ1 (Accuracy): How does the accuracy of MANDOLINE compare to the state-of-the-art?

RQ2 (Performance): How does the performance of MANDOLINE compare to the state-of-the-art?

A. Experimental Setup

1) Subject Applications: To create a benchmark of manual slices, we started from DroixBench [21] and ReCDroid [20] benchmarks, which consist of 24 and 51 faulty apps, respectively. All faults in these apps manifest in crashes. We chose these benchmarks because each app has an associated GitHub issue describing steps for reproducing the crash and because the statement in which an app crashes, together with all variables it uses, is a realistic slicing criterion.

We omitted apps for which we could not reproduce the crash and (1) apps that do not specify a fix for the crash, (2) apps where the fix is not in the source code of the app but rather in the resources or build files, (3) apps where the fix is not related to the existing code of the buggy app, and (4) apps that could not be instrumented to obtain a dynamic trace, most probably because they use language constructs that are not supported by the underlying Soot framework. For the remaining 14 apps, we manually reproduced the crash using the description in the GitHub issue and further filtered out two apps which required physical interaction with the device to trigger the crash, e.g., changing the screen orientation.

Our final dataset consists of 12 apps. We recorded execution scripts reproducing the crash in each app using the android-touch-record-replay tool [28]. We manually analyzed each app, applying the slicing algorithm described in Algorithm 1 to establish the "ground truth" dynamic slice. The manual slices were produced independently and in parallel by two members of our research group, in an effort that took more than 30 work-days. Any observed differences were discussed in a meeting with all the authors towards reaching a joint resolution.

Table I summarizes the subjects of our experiment. The table columns show the app name, app size, trace size, and the manual slice size. We report all sizes in the number of Jimple statements of the underlying app representation (#JS).

2) Methods and Metrics: We compare MANDOLINE to ANDROIDSLICER, the state-of-the-art dynamic slicer for Android. The available implementation of ANDROIDSLICER [29] suffers from several shortcomings, such as missing control dependency analysis and missing data-flow analysis through method parameters and returns. ANDROIDSLICER also uses statement-level instrumentation, which causes a high overhead when compared to basic-block level instrumentation. We thus contribute an enhanced version of ANDROIDSLICER, called ANDROIDSLICER++, which addresses these ANDROIDSLICER implementation issues.

To answer **RQ1**, we compute slices using ANDROIDSLICER, ANDROIDSLICER++, and MANDOLINE. We compare the manual slices with the slices produced by each tool and calculate the recall (R), precision (P), and F-Measure (F) achieved by each tool. The recall is the fraction of statement instances in both the computed slice and the manual slice out of the number of statement instances in the manual slice. It measures the tools' ability to correctly identify statement instances that are in the manual slice. The precision is the fraction of statement instances in both the computed slice and the manual slice out of the number of statement instances in the computed slice. It measures the tools' ability to exclude statement instances that are not in the manual slice. The F-measure is a harmonic mean that balances precision and recall. It reflects how well the tool can balance identifying relevant statement instances (in the manual slice) while rejecting irrelevant statements (not in the manual slice), $F\text{-}measure = 2\frac{R*P}{R+P}$.

Table I: Evaluation subject apps.

App	Size (#JS)	Trace (#JS)	Manual Slice (#JS)
anki	250,966	291,910	82
birthdroid	24,999	1,060	23
fastadapter	202,815	488,010	226
fdroid	412,380	1,197,098	282
gnucash (2.0.5)	328,603	389,827	46
gnucash (2.1.4)	443,431	1,843,698	80
habdroid	327,250	842,874	4
k9 (5.111)	249,858	80,007	7
k9 (5.403)	259,033	10,602	99
micromath	325,292	2,381,831	27
newsblur	68,103	16,985	51
specialdates	146,360	275,504	353

To answer **RQ2**, we compare the instrumentation overhead of the tools as well as their slicing time. To reduce execution time fluctuation due to external causes, we run each experiment five times and average the execution times for each experiment. For the instrumentation overhead, we perform all runs on the same Pixel 2 device running Android API 28. We follow the work of Li et al. [30] in measuring the on-device CPU execution time of the app, *CPUt*. Specifically, we add additional lightweight instrumentation in all apps to record the execution time of each thread and callback and compute the app's CPU time as the sum of the execution time of all threads and callbacks. Such method eliminates input-dependent wait time, e.g., when the app waits for sensory inputs. We then compute the instrumentation overhead (O) for each instrumentation method compared with the CPU time of the original app, $O = \frac{CPUt_{tool} - CPUt_{original}}{CPUt_{original}} * 100\%$. For the slicing time, we run the slicing part of each tool on its corresponding trace and measure the execution time (Ht). We run these experiments on an Intel Xeon 2.6 GHz machine, allocating one core and 124GB of RAM to each run.

B. Results

RQ1: Table II shows the slice size (#JS) for the manual slice (column 2) and then for the slice produced by each of the tools (columns 3, 7, and 11). It also shows the accuracy results for each tool: recall, precision, and F-measure. Our results show that the accuracy of MANDOLINE, in terms of F-measure, is higher than that of ANDROIDSLICER in all 12 apps. Furthermore, the average F-measure for MANDOLINE is 81%, compared with only 48% for ANDROIDSLICER++ and 22% for ANDROIDSLICER.

In two of the apps, `anki` and `birthdroid`, ANDROIDSLICER achieves higher precision than MANDOLINE. That is because the slice computed by ANDROIDSLICER is relatively small; it thus contains only a few irrelevant statements. Yet, such behavior leads to a low recall and low overall F-measure.

Table II: Accuracy.

App	Manual	AndroidSlicer				AndroidSlicer++				Mandoline			
	#JS	#JS	R%	P%	F%	#JS	R%	P%	F%	#JS	R%	P%	F%
anki	82	12	14	92	24	3	4	100	7	83	92	91	92
birthdroid	23	6	33	100	49	8	34	100	51	18	73	94	82
fastadapter	226	43	7	33	11	28	12	100	22	226	73	73	73
fdroid	282	20	2	30	4	54	15	80	25	270	84	88	86
gnucash (2.0.5)	46	10	13	60	21	48	43	41	42	46	73	73	73
gnucash (2.1.4)	80	6	2	83	4	18	17	77	28	59	62	84	72
habdroid	4	4	75	75	75	4	50	100	66	4	100	100	100
k9 (5.111)	7	11	28	18	22	7	100	100	100	7	100	100	100
k9 (5.403)	99	7	5	41	9	82	55	77	60	90	57	63	60
micromath	27	43	40	25	31	11	40	100	57	33	92	75	83
newsblur	51	10	5	30	10	45	82	93	87	51	96	96	96
specialdates	353	44	3	18	5	59	15	94	27	226	47	73	57
Mean	107	18	19	50	22	30	39	87	48	92	79	84	81

Table III: Instrumentation overhead and slicing time (in seconds).

App	Exec. time(s)	AndroidSlicer			AndroidSlicer++			Mandoline		
		CPU$_t$	*O%*	Ht(s)	CPU$_t$	*O%*	Ht(s)	CPU$_t$	*O%*	Ht(s)
anki	0.4	7.8	1765.5	59.2	-	-	70.9	2.3	462.1	230.6
birthdroid	0.1	0.1	6.8	3.77	-	-	5.98	0.1	1.4	52.6
fastadapter	0.7	11	1498.9	118.8	-	-	75.2	5.4	685	284.5
fdroid	6.4	44.9	601.9	63.8	-	-	103.7	6.7	5.7	638.6
gnucash (2.0.5)	0.6	12.5	1805.8	42.7	-	-	103.3	3	360.3	302.2
gnucash (2.1.4)	1.7	34.9	1850.7	40.5	-	-	264.1	8.3	368.8	964.9
habdroid	0.8	56.5	7067.5	37.6	-	-	88.6	6.5	720.6	403.5
k9 (5.111)	0.2	2.9	1445.2	49.5	-	-	86.41	0.2	25.9	212.2
k9 (5.403)	0.1	0.3	364.9	57.5	-	-	81.25	0.7	13.2	227
micromath	0.4	102	27654.3	29.3	-	-	131.3	2.7	654.1	572.8
newsblur	0.1	0.9	671.1	17.2	-	-	15.2	0.4	253.4	150
specialdates	0.2	6	2492.9	27.4	-	-	38.3	2.3	893.9	208.1
Mean	0.9	23.3	3935.5	45.6	-	-	88.7	3.2	370.9	353.9

Mandoline outperforms AndroidSlicer++ in terms of F-measure in 10 apps and performs equivalently in the remaining two: *k9 (5.111)* and *k9 (5.403)*. For the first app, both tools achieve a perfect F-measure of 100% because the slice does not contain any field references: it has only seven statements all within two methods calling each other in a single callback. AndroidSlicer cannot identify the desired slice because it misses control flows from method return statements. Moreover, it identifies irrelevant statements in the callee method due to overapproximation in dealing with method parameters.

For the second app, while the recall of Mandoline is slightly higher than that of AndroidSlicer++ (57% vs. 55%), its precision is lower (63% instead of 67%). In this app, Mandoline adds unnecessary statements to the slice because of its approximate handling of arrays: when looking for a definition of an element in a certain position in an array, Mandoline includes definitions of all array elements.

In addition, we observed two main reasons Mandoline cannot achieve an even higher accuracy. The first is due to inaccurate handling of registration dependencies between callbacks. When connecting one callback to another, Mandoline relies on FlowDroid's point-to analysis to identify the type of the registered callback listener. Inaccurate analysis leads to overapproximations in possible listener types, e.g., all listeners implementing the `View.OnClickListener` interface. This results in unnecessary registration dependencies among callbacks, causing Mandoline to traverse unfeasible paths in the ICDG, identify incorrect variable definitions (reduced precision), and miss the correct ones (reduced recall).

Another reason for the drop in accuracy is the absence of certain framework methods, e.g., native methods, in the StubDroid and FlowDroid taint-wrapper models. Such methods are treated with the default strategy, which assumes that the method only modifies the return value. As a result, the data-flow propagation cannot reach the desired definitions.

Answer to RQ1: Our experiments show that using field alias analysis and framework modeling with dynamic slicing is effective and achieves more than 81% accuracy on average, almost four times higher than that of AndroidSlicer.

RQ2: Table III shows the CPU time, in seconds, of each app without any instrumentation (column 2) and the corresponding numbers for AndroidSlicer's statement-level (column 3) and Mandoline's basic-block-level instrumentation (column 9). AndroidSlicer++ relies on the same instrumentation as Mandoline and we thus omit these numbers in the table. The table also shows the overhead of both instrumentations (column 4 for AndroidSlicer and column 10 for Mandoline), when compared with the app without any instrumentation. Our results confirm that the instrumentation overhead of Mandoline is consistently lower than that of AndroidSlicer, with the difference being more noticeable as the runtime of an app increases, e.g., *fdroid* and *micromath*.

At the same time, the slicing time for ANDROIDSLICER, ANDROIDSLICER++ and MANDOLINE (columns 5, 8, and 11, respectively) shows that MANDOLINE is slower than the other two. That is because it needs to perform the on-demand field alias analysis and also processes bigger slices. We still find the slicing execution time acceptable given the improved accuracy and run-time overhead: it takes MANDOLINE around 6 minutes on average to compute a slice while ANDROIDSLICER++ takes less than 2 minutes on average.

Answer to RQ2: Our experiments show that, on average, the runtime overhead of MANDOLINE is 10 times lower than the state-of-the-art. Its increased accuracy comes at the expense of almost eight-fold increase in slicing time.

C. Limitations and Threats to Validity

For **external validity**, our results may be affected by the subject apps' selection and may not necessarily generalize beyond our subjects. We attempted to mitigate this threat by using a set of apps available from related work without introducing investigator bias into the selection process. As we used different apps of considerable size and complexity, we believe our results are reliable. Furthermore, we had to create the ground truth for dynamic slicing manually, as such a benchmark was not available from prior research. To mitigate possible bias related to this manual effort, two members of our research group performed the analysis independently and cross-checked each other's results. We make our implementation and evaluation setup publicly available [22] to encourage validation and replication of our results.

For **internal validity**, deficiencies of the underlying tools our approach uses, such as Soot, FlowDroid, and StubDroid, might affect the accuracy of MANDOLINE. We controlled for this threat by manually analyzing the cases that we considered, to identify reasons for inaccuracies.

The main **limitation of our approach** is in relying on static information for building ICDG, such as the list of threads and callbacks from FlowDroid, object classes to identify targets for the callback registration statements, and statically generated method summaries for framework methods. We plan to investigate ways to address these limitations, e.g., by obtaining more accurate information dynamically, as part of future work.

VI. RELATED WORK

Our discussion of related work focuses on efficient dynamic slicing techniques and alias analysis.

Dynamic slicing. Agrawal et al. [3] presented dynamic slicing using graphs of statement instances. Duesterwald et al. [31] extended dynamic slicing for multi-threaded programs by using inter-thread data dependencies. Agrawal et al. [4] produced dynamic slices for programs with pointers by memory overlapping to find reaching definitions of pointers. It also adapted the slicing algorithm to work across methods.

Gupta et al. [32] proposed a hybrid approach to reduce the overhead of dynamic slicing. Instrumentation is done at a limited number of statements instead of the whole code, this instrumentation serves to eliminate unfeasible paths from the static control flow graph before performing static slicing. Tallam et al. [33] proposed a technique to record execution traces for threads relevant to the fault while ignoring irrelevant threads. Wang et al. [5] proposed compressing the execution trace at runtime to reduce the trace size. Zhang et al. [13] proposed an efficient method for traversing the trace to find reaching definitions by splitting the trace into chunks and summarize defined variables within a chunk. Our work is inspired by these approaches to reduce runtime overhead; however, it focuses specifically on computing data flows from lightweight execution traces and can be combined with these efficient trace collection mechanisms in future work.

ANDROIDSLICER [11] is the closest to our work as it is the only other dynamic slicer for Android apps. ANDROIDSLICER solves the problem of slicing across multiple components by modeling component transitions. Yet, it keeps the runtime overhead low by sacrificing field dependencies tracking. Our work deals with this limitation via field alias analysis and also contributes modeling of Android framework methods to increase slicing accuracy.

SAAF [34] is the first static slicer for Android apps as it can slice Android bytecode. Harvester [35] is also a static slicer for Android that is capable of producing executable slices. Static slicers suffer from a common drawback: a larger slice, when compared to their dynamic counterparts.

Alias analysis. Zheng et al. [16] proposed an alias analysis using a graph representation of the program. Yan et al. [17] modeled aliases using a graph representation of the heap. Boomerang [36] utilizes the IFDS framework [37] to find aliases of fields. Andromeda [18] is the closest to our work since it uses alias sets of access paths for the alias analysis. All static alias analysis techniques face challenges such as dealing with context, path, and flow sensitivities while keeping the analysis scalable. Such challenges do not exist for a dynamic trace: the calling context, the execution path, and control flows are all known from the trace. Our technique is the first to propose alias analysis directly on the trace and thus does not suffer from the challenges of the static tools.

VII. CONCLUSION

This paper introduced an accurate and efficient approach for dynamic slicing of Android applications and implemented it in a tool named MANDOLINE. The main contribution of MANDOLINE is the lightweight application instrumentation that produces the minimally-necessary app execution trace, followed by a sophisticated analysis performed on the obtained trace to recover data dependencies via alias analysis and modeling of the Android framework methods. Our experimental evaluation shows that MANDOLINE outperforms the state-of-the-art Android slicing tool, ANDROIDSLICER, in both accuracy and efficiency.

Acknowledgments. We thank our research group member, Yingying Wang, for the help in creating and cross-validating the manual slices. We also thank the authors of ANDROIDSLICER for answering our questions about the tool.

References

[1] M. Weiser, "Program Slicing," in *Proc. of the International Conference on Software Engineering (ICSE)*, 1981, p. 439–449.

[2] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[3] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic Slicing in the Presence of Unconstrained Pointers," in *Proc. of the Symposium on Testing, Analysis, and Verification (TAV)*, 1991, pp. 60–73.

[5] T. Wang and A. Roychoudhury, "Using Compressed Bytecode Traces for Slicing Java Programs," in *Proc. the International Conference on Software Engineering (ICSE)*, 2004, pp. 512—521.

[6] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-Localization Using Dynamic Slicing and Change Impact Analysis," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2011, pp. 520–523.

[7] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based Statistical Fault Localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[8] X. Li and A. Orso, "More Accurate Dynamic Slicing for Better Supporting Software Debugging," in *Proc. of the International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 28–38.

[9] Wikipedia, "Android version history," https://en.wikipedia.org/wiki/Android_version_history, (Last accessed: January 2021).

[10] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet Lightweight Record-and-Replay for Android," in *Proc. of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 349—366.

[11] T. Azim, A. Alavi, I. Neamtiu, and R. Gupta, "Dynamic Slicing for Android," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2019, pp. 1154–1164.

[12] Google, "Buttons," https://developer.android.com/guide/topics/ui/controls/button, (Last accessed: January 2021).

[13] X. Zhang, R. Gupta, and Y. Zhang, "Precise Dynamic Slicing Algorithms," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2003, pp. 319–329.

[14] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004, pp. 94–106.

[15] A. Deutsch, "A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations," in *Proc. of the International Conference on Computer Languages (ICCL)*, 1992, pp. 2–13.

[16] X. Zheng and R. Rugina, "Demand-Driven Alias Analysis for C," in *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008, p. 197–208.

[17] D. Yan, G. Xu, and A. Rountev, "Demand-Driven Context-Sensitive Alias Analysis for Java," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 155—165.

[18] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications," in *Proc. of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013, pp. 210–225.

[19] S. Arzt and E. Bodden, "StubDroid: Automatic Inference of Precise Date-flow Summaries for the Android Framework," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2016, pp. 725–735.

[20] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2019, pp. 128–139.

[21] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing Crashes in Android Apps," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2018, pp. 187–198.

[22] "Mandoline," https://resess.github.io/PaperAppendices/Mandoline/, 2020.

[23] F. E. Allen, "Control Flow Analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, p. 1–19, 1970.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[25] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 1999, pp. 1–11.

[26] M. Kamkar, N. Shahmehri, and P. Fritzson, "Interprocedural Dynamic Slicing," in *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, 1992, pp. 370–384.

[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 259––269.

[28] J. Cartucho, "Record and Replay Touchscreen Events on Android," https://github.com/Cartucho/android-touch-record-replay, (Last accessed: January 2021).

[29] T. Azim, A. Alavi, I. Neamtiu, and R. Gupta, "AndroidSlicer," https://github.com/ucr-riple/AndroidSlicer, (Last accessed: January 2021).

[30] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating Source Line Level Energy Information For Android Applications," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 78–89.

[31] E. Duesterwald, R. Gupta, and M. L. Soffa, "Distributed Slicing and Partial Re-execution for Distributed Programs," in *Proc. of Languages and Compilers for Parallel Computing*, 1993, pp. 497–511.

[32] R. Gupta, M. L. Soffa, and J. Howard, "Hybrid Slicing: Integrating Dynamic Information with Static Analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 4, pp. 370–397, 1997.

[33] S. Tallam, C. Tian, R. Gupta, and X. Zhang, "Enabling Tracing Of Long-Running Multithreaded Programs via Dynamic Execution Reduction," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2007, p. 207–218.

[34] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing Droids: Program Slicing for Smali Code," in *Proc. of the Annual ACM Symposium on Applied Computing (SAC)*, 2013, pp. 1844–1851.

[35] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques." in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[36] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2016, pp. 22:1–22:26.

[37] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proc. of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1995, pp. 49–61.